Data Structures & Algorithms

in C++

SECOND EDITION

MICHAEL T. GOODRICH • ROBERTO TAMASSIA • DAVID MOUNT

This page intentionally left blank

Data Structures and Algorithms in C++

Second Edition

This page intentionally left blank

Data Structures and Algorithms in C++

Second Edition

Michael T. Goodrich
Department of Computer Science
University of California, Irvine

Roberto Tamassia
Department of Computer Science
Brown University

David M. Mount
Department of Computer Science
University of Maryland

To Karen, Paul, Anna, and Jack
– Michael T. Goodrich

To Isabel
– Roberto Tamassia

To Jeanine
– David M. Mount

This page intentionally left blank

Preface

This second edition of *Data Structures and Algorithms in* C++ is designed to provide an introduction to data structures and algorithms, including their design, analysis, and implementation. In terms of curricula based on the **IEEE/ACM 2001 Computing Curriculum**, this book is appropriate for use in the courses CS102 (I/O/B versions), CS103 (I/O/B versions), CS111 (A version), and CS112 (A/I/O/F/H versions). We discuss its use for such courses in more detail later in this preface.

The major changes in the second edition are the following:

- We added more examples of data structure and algorithm analysis.
- We enhanced consistency with the C++ Standard Template Library (STL).
- We incorporated STL data structures into many of our data structures.
- We added a chapter on arrays, linked lists, and iterators (Chapter 3).
- We added a chapter on memory management and B-trees (Chapter 14).
- We enhanced the discussion of algorithmic design techniques, like dynamic programming and the greedy method.
- We simplified and reorganized the presentation of code fragments.
- We have introduced STL-style iterators into our container classes, and have presented C++ implementations for these iterators, even for complex structures such as hash tables and binary search trees.
- We have modified our priority-queue interface to use STL-style comparator objects.
- We expanded and revised exercises, continuing our approach of dividing them into reinforcement, creativity, and project exercises.

This book is related to the following books:

- M.T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, John Wiley & Sons, Inc. This book has a similar overall structure to the present book, but uses Java as the underlying language (with some modest, but necessary pedagogical differences required by this approach).
- M.T. Goodrich and R. Tamassia, *Algorithm Design: Foundations, Analysis, and Internet Examples*, John Wiley & Sons, Inc. This is a textbook for a more advanced algorithms and data structures course, such as CS210 (T/W/C/S versions) in the IEEE/ACM 2001 curriculum.

While this book retains the same pedagogical approach and general structure as *Data Structures and Algorithms in Java*, the code fragments have been completely redesigned. We have been careful to make full use of C++'s capabilities and design code in a manner that is consistent with modern C++ usage. In particular, whenever appropriate, we make extensive use of C++ elements that are not part of Java, including the C++ Standard Template Library (STL), C++ memory allocation

and deallocation (and the associated issues of destructors), virtual functions, stream input and output, operator overloading, and C++'s safe run-time casting.

Use as a Textbook

The design and analysis of efficient data structures has long been recognized as a vital subject in computing, because the study of data structures is part of the core of every collegiate computer science and computer engineering major program we are familiar with. Typically, the introductory courses are presented as a two- or three-course sequence. Elementary data structures are often briefly introduced in the first programming course or in an introduction to computer science course and this is followed by a more in-depth introduction to data structures in the courses that follow after this. Furthermore, this course sequence is typically followed at a later point in the curriculum by a more in-depth study of data structures and algorithms. We feel that the central role of data structure design and analysis in the curriculum is fully justified, given the importance of efficient data structures in most software systems, including the Web, operating systems, databases, compilers, and scientific simulation systems.

With the emergence of the object-oriented paradigm as the framework of choice for building robust and reusable software, we have tried to take a consistent object-oriented viewpoint throughout this text. One of the main ideas behind the object-oriented approach is that data should be presented as being encapsulated with the methods that access and modify them. That is, rather than simply viewing data as a collection of bytes and addresses, we think of data objects as instances of an ***abstract data type*** (***ADT***), which includes a repertoire of methods for performing operations on data objects of this type. Likewise, object-oriented solutions are often organized utilizing common ***design patterns***, which facilitate software reuse and robustness. Thus, we present each data structure using ADTs and their respective implementations and we introduce important design patterns as a way to organize those implementations into classes, methods, and objects.

For most of the ADTs presented in this book, we provide a description of the public interface in C++. Also, concrete data structures realizing the ADTs are discussed and we often give concrete C++ classes implementing these interfaces. We also give C++ implementations of fundamental algorithms, such as sorting and graph searching. Moreover, in addition to providing techniques for using data structures to implement ADTs, we also give sample applications of data structures, such as HTML tag matching and a simple system to maintain a play list for a digital audio system. Due to space limitations, however, we only show code fragments of some of the implementations in this book and make additional source code available on the companion web site.

Online Resources

This book is accompanied by an extensive set of online resources, which can be found at the following web site:

www.wiley.com/college/goodrich

Included on this Web site is a collection of educational aids that augment the topics of this book, for both students and instructors. Students are encouraged to use this site along with the book, to help with exercises and increase understanding of the subject. Instructors are likewise welcome to use the site to help plan, organize, and present their course materials. Because of their added value, some of these online resources are password protected.

For the Student

For all readers, and especially for students, we include the following resources:

- All the C++ source code presented in this book.
- PDF handouts of Powerpoint slides (four-per-page) provided to instructors.
- A database of hints to *all* exercises, indexed by problem number.
- An online study guide, which includes solutions to selected exercises.

The hints should be of considerable use to anyone needing a little help getting started on certain exercises, and the solutions should help anyone wishing to see completed exercises. Students who have purchased a new copy of this book will get password access to the hints and other password-protected online resources at no extra charge. Other readers can purchase password access for a nominal fee.

For the Instructor

For instructors using this book, we include the following additional teaching aids:

- Solutions to over 200 of the book's exercises.
- A database of additional exercises, suitable for quizzes and exams.
- Additional C++ source code.
- Slides in Powerpoint and PDF (one-per-page) format.
- Self-contained, special-topic supplements, including discussions on convex hulls, range trees, and orthogonal segment intersection.

The slides are fully editable, so as to allow an instructor using this book full freedom in customizing his or her presentations. All the online resources are provided at no extra charge to any instructor adopting this book for his or her course.

A Resource for Teaching Data Structures and Algorithms

This book contains many C++-code and pseudo-code fragments, and hundreds of exercises, which are divided into roughly 40% reinforcement exercises, 40% creativity exercises, and 20% programming projects.

This book can be used for the CS2 course, as described in the 1978 ACM Computer Science Curriculum, or in courses CS102 (I/O/B versions), CS103 (I/O/B versions), CS111 (A version), and/or CS112 (A/I/O/F/H versions), as described in the IEEE/ACM 2001 Computing Curriculum, with instructional units as outlined in Table 0.1.

Instructional Unit	*Relevant Material*
PL1. Overview of Programming Languages	Chapters 1 and 2
PL2. Virtual Machines	Sections 14.1.1 and 14.1.2
PL3. Introduction to Language Translation	Section 1.7
PL4. Declarations and Types	Sections 1.1.2, 1.1.3, and 2.2.5
PL5. Abstraction Mechanisms	Sections 2.2.5, 5.1–5.3, 6.1.1, 6.2.1, 6.3, 7.1, 7.3.1, 8.1, 9.1, 9.5, 11.4, and 13.1.1
PL6. Object-Oriented Programming	Chapters 1 and 2 and Sections 6.2.1, 7.3.7, 8.1.2, and 13.3.1
PF1. Fundamental Programming Constructs	Chapters 1 and 2
PF2. Algorithms and Problem-Solving	Sections 1.7 and 4.2
PF3. Fundamental Data Structures	Sections 3.1, 3.2, 5.1–5.3, 6.1–6.3, 7.1, 7.3, 8.1, 8.3, 9.1–9.4, 10.1, and 13.1.1
PF4. Recursion	Section 3.5
SE1. Software Design	Chapter 2 and Sections 6.2.1, 7.3.7, 8.1.2, and 13.3.1
SE2. Using APIs	Sections 2.2.5, 5.1–5.3, 6.1.1, 6.2.1, 6.3, 7.1, 7.3.1, 8.1, 9.1, 9.5, 11.4, and 13.1.1
AL1. Basic Algorithmic Analysis	Chapter 4
AL2. Algorithmic Strategies	Sections 11.1.1, 11.5.1, 12.2, 12.3.1, and 12.4.2
AL3. Fundamental Computing Algorithms	Sections 8.1.5, 8.2.2, 8.3.5, 9.2, and 9.3.1, and Chapters 11, 12, and 13
DS1. Functions, Relations, and Sets	Sections 4.1, 8.1, and 11.4
DS3. Proof Techniques	Sections 4.3, 6.1.3, 7.3.3, 8.3, 10.2–10.5, 11.2.1, 11.3.1, 11.4.3, 13.1.1, 13.3.1, 13.4, and 13.5
DS4. Basics of Counting	Sections 2.2.3 and 11.1.5
DS5. Graphs and Trees	Chapters 7, 8, 10, and 13
DS6. Discrete Probability	Appendix A and Sections 9.2, 9.4.2, 11.2.1, and 11.5

Table 0.1: Material for units in the IEEE/ACM 2001 Computing Curriculum.

Contents and Organization

The chapters for this course are organized to provide a pedagogical path that starts with the basics of C++ programming and object-oriented design. We provide an early discussion of concrete structures, like arrays and linked lists, in order to provide a concrete footing to build upon when constructing other data structures. We then add foundational techniques like recursion and algorithm analysis, and, in the main portion of the book, we present fundamental data structures and algorithms, concluding with a discussion of memory management (that is, the architectural underpinnings of data structures). Specifically, the chapters for this book are organized as follows:

1. **A C++ Primer**
2. **Object-Oriented Design**
3. **Arrays, Linked Lists, and Recursion**
4. **Analysis Tools**
5. **Stacks, Queues, and Deques**
6. **List and Iterator ADTs**
7. **Trees**
8. **Heaps and Priority Queues**
9. **Hash Tables, Maps, and Skip Lists**
10. **Search Trees**
11. **Sorting, Sets, and Selection**
12. **Strings and Dynamic Programming**
13. **Graph Algorithms**
14. **Memory Management and B-Trees**
A. **Useful Mathematical Facts**

A more detailed listing of the contents of this book can be found in the table of contents.

Prerequisites

We have written this book assuming that the reader comes to it with certain knowledge. We assume that the reader is at least vaguely familiar with a high-level programming language, such as C, C++, Python, or Java, and that he or she understands the main constructs from such a high-level language, including:

- Variables and expressions.
- Functions (also known as methods or procedures).
- Decision structures (such as if-statements and switch-statements).
- Iteration structures (for-loops and while-loops).

For readers who are familiar with these concepts, but not with how they are expressed in C++, we provide a primer on the C++ language in Chapter 1. Still, this book is primarily a data structures book, not a C++ book; hence, it does not provide a comprehensive treatment of C++. Nevertheless, we do not assume that the reader is necessarily familiar with object-oriented design or with linked structures, such as linked lists, since these topics are covered in the core chapters of this book.

In terms of mathematical background, we assume the reader is somewhat familiar with topics from high-school mathematics. Even so, in Chapter 4, we discuss the seven most-important functions for algorithm analysis. In fact, sections that use something other than one of these seven functions are considered optional, and are indicated with a star ($\star$). We give a summary of other useful mathematical facts, including elementary probability, in Appendix A.

About the Authors

Professors Goodrich, Tamassia, and Mount are well-recognized researchers in algorithms and data structures, having published many papers in this field, with applications to Internet computing, information visualization, computer security, and geometric computing. They have served as principal investigators in several joint projects sponsored by the National Science Foundation, the Army Research Office, the Office of Naval Research, and the Defense Advanced Research Projects Agency. They are also active in educational technology research.

Michael Goodrich received his Ph.D. in Computer Science from Purdue University in 1987. He is currently a Chancellor's Professor in the Department of Computer Science at University of California, Irvine. Previously, he was a professor at Johns Hopkins University. He is an editor for a number of journals in computer science theory, computational geometry, and graph algorithms. He is an ACM Distinguished Scientist, a Fellow of the American Association for the Advancement of Science (AAAS), a Fulbright Scholar, and a Fellow of the IEEE. He is a recipient of the IEEE Computer Society Technical Achievement Award, the ACM Recognition of Service Award, and the Pond Award for Excellence in Undergraduate Teaching.

Roberto Tamassia received his Ph.D. in Electrical and Computer Engineering from the University of Illinois at Urbana-Champaign in 1988. He is the Plastech Professor of Computer Science and the Chair of the Department of Computer Science at Brown University. He is also the Director of Brown's Center for Geometric Computing. His research interests include information security, cryptography, analysis, design, and implementation of algorithms, graph drawing, and computational geometry. He is an IEEE Fellow and a recipient of the Technical Achievement Award from the IEEE Computer Society for pioneering the field of graph drawing. He is an editor of several journals in geometric and graph algorithms. He previously served on the editorial board of *IEEE Transactions on Computers*.

David Mount received his Ph.D. in Computer Science from Purdue University in 1983. He is currently a professor in the Department of Computer Science at the University of Maryland with a joint appointment in the University of Maryland's Institute for Advanced Computer Studies. He is an associate editor for *ACM Transactions on Mathematical Software* and the *International Journal of Computational Geometry and Applications*. He is the recipient of two ACM Recognition of Service Awards.

In addition to their research accomplishments, the authors also have extensive experience in the classroom. For example, Dr. Goodrich has taught data structures and algorithms courses, including Data Structures as a freshman-sophomore level course and Introduction to Algorithms as an upper-level course. He has earned several teaching awards in this capacity. His teaching style is to involve the students in lively interactive classroom sessions that bring out the intuition and insights behind data structuring and algorithmic techniques. Dr. Tamassia has taught Data Structures and Algorithms as an introductory freshman-level course since 1988. One thing that has set his teaching style apart is his effective use of interactive hypermedia presentations integrated with the Web. Dr. Mount has taught both the Data Structures and the Algorithms courses at the University of Maryland since 1985. He has won a number of teaching awards from Purdue University, the University of Maryland, and the Hong Kong University of Science and Technology. His lecture notes and homework exercises for the courses that he has taught are widely used as supplementary learning material by students and instructors at other universities.

Acknowledgments

We are also grateful to Karen Goodrich, Art Moorshead, Scott Smith, and Ioannis Tollis for their insightful comments.

We are also truly indebted to the outside reviewers and readers for their copious comments, emails, and constructive criticism, which were extremely useful in writing this edition. We specifically thank the following reviewers for their comments and suggestions: Divy Agarwal, University of California, Santa Barbara; Terry Andres, University of Manitoba; Bobby Blumofe, University of Texas, Austin; Michael Clancy, University of California, Berkeley; Larry Davis, University of Maryland; Scott Drysdale, Dartmouth College; Arup Guha, University of Central Florida; Chris Ingram, University of Waterloo; Stan Kwasny, Washington University; Calvin Lin, University of Texas at Austin; John Mark Mercer, McGill University; Laurent Michel, University of Connecticut; Leonard Myers, California Polytechnic State University, San Luis Obispo; David Naumann, Stevens Institute of Technology; Robert Pastel, Michigan Technological University; Bina Ramamurthy, SUNY Buffalo; Ken Slonneger, University of Iowa; C.V. Ravishankar, University of Michigan; Val Tannen, University of Pennsylvania; Paul Van Arragon, Messiah College; and Christopher Wilson, University of Oregon.

We are grateful to our editor, Beth Golub, for her enthusiastic support of this project. The team at Wiley has been great. Many thanks go to Mike Berlin, Lilian Brady, Regina Brooks, Paul Crockett, Richard DeLorenzo, Jen Devine, Simon Durkin, Micheline Frederick, Lisa Gee, Katherine Hepburn, Rachael Leblond, Andre Legaspi, Madelyn Lesure, Frank Lyman, Hope Miller, Bridget Morrisey, Chris Ruel, Ken Santor, Lauren Sapira, Dan Sayre, Diana Smith, Bruce Spatz, Dawn Stanley, Jeri Warner, and Bill Zobrist.

The computing systems and excellent technical support staff in the departments of computer science at Brown University, University of California, Irvine, and University of Maryland gave us reliable working environments. This manuscript was prepared primarily with the LaTeX typesetting package.

Finally, we would like to warmly thank Isabel Cruz, Karen Goodrich, Jeanine Mount, Giuseppe Di Battista, Franco Preparata, Ioannis Tollis, and our parents for providing advice, encouragement, and support at various stages of the preparation of this book. We also thank them for reminding us that there are things in life beyond writing books.

<div align="right">

Michael T. Goodrich
Roberto Tamassia
David M. Mount

</div>

Contents

Chapter

1

A C++ Primer

Contents

1.1   Basic C++ Programming Elements

Building data structures and algorithms requires communicating instructions to a computer, and an excellent way to perform such communication is using a high-level computer language, such as C++. C++ evolved from the programming language C, and has, over time, undergone further evolution and development from its original definition. It has incorporated many features that were not part of C, such as symbolic constants, in-line function substitution, reference types, parametric polymorphism through templates, and exceptions (which are discussed later). As a result, C++ has grown to be a complex programming language. Fortunately, we do not need to know every detail of this sophisticated language in order to use it effectively.

In this chapter and the next, we present a quick tour of the C++ programming language and its features. It would be impossible to present a complete presentation of the language in this short space, however. Since we assume that the reader is already familiar with programming with some other language, such as C or Java, our descriptions are short. This chapter presents the language's basic features, and in the following chapter, we concentrate on those features that are important for object-oriented programming.

C++ is a powerful and flexible programming language, which was designed to build upon the constructs of the C programming language. Thus, with minor exceptions, C++ is a superset of the C programming language. C++ shares C's ability to deal efficiently with hardware at the level of bits, bytes, words, addresses, etc. In addition, C++ adds several enhancements over C (which motivates the name "C++"), with the principal enhancement being the object-oriented concept of a *class*.

A class is a user-defined type that encapsulates many important mechanisms such as guaranteed initialization, implicit type conversion, control of memory management, operator overloading, and polymorphism (which are all important topics that are discussed later in this book). A class also has the ability to hide its underlying data. This allows a class to conceal its implementation details and allows users to conceptualize the class in terms of a well-defined interface. Classes enable programmers to break an application up into small, manageable pieces, or *objects*. The resulting programs are easier to understand and easier to maintain.

1.1.1   A Simple C++ Program

Like many programming languages, creating and running a C++ program requires several steps. First, we create a C++ source file into which we enter the lines of our program. After we save this file, we then run a program, called a *compiler*, which

creates a machine-code interpretation of this program. Another program, called a *linker* (which is typically invoked automatically by the compiler), includes any required library code functions needed and produces the final machine-executable file. In order to run our program, the user requests that the system execute this file.

Let us consider a very simple program to illustrate some of the language's basic elements. Don't worry if some elements in this example are not fully explained. We discuss them in greater depth later in this chapter. This program inputs two integers, which are stored in the variables x and y. It then computes their sum and stores the result in a variable sum, and finally it outputs this sum. (The line numbers are not part of the program; they are just for our reference.)

```
 1  #include <cstdlib>
 2  #include <iostream>
 3  /* This program inputs two numbers x and y and outputs their sum */
 4  int main( ) {
 5     int x, y;
 6     std::cout << "Please enter two numbers: ";
 7     std::cin >> x >> y;                    // input x and y
 8     int sum = x + y;                       // compute their sum
 9     std::cout << "Their sum is " << sum << std::endl;
10     return EXIT_SUCCESS;                   // terminate successfully
11  }
```

A few things about this C++ program should be fairly obvious. First, comments are indicated with two slashes (//). Each such comment extends to the end of the line. Longer block comments are enclosed between /* and */. Block comments may extend over multiple lines. The quantities manipulated by this program are stored in three integer variables, x, y, and sum. The operators ">>" and "<<" are used for input and output, respectively.

Program Elements

Let us consider the elements of the above program in greater detail. Lines 1 and 2 input the two *header files*, "cstdlib" and "iostream." Header files are used to provide special declarations and definitions, which are of use to the program. The first provides some standard system definitions, and the second provides definitions needed for input and output.

The initial entry point for C++ programs is the function main. The statement "int main( )" on line 4 declares main to be a function that takes no arguments and returns an integer result. (In general, the main function may be called with the command-line arguments, but we don't discuss this.) The *function body* is given within curly braces ({...}), which start on line 4 and end on line 11. The program terminates when the return statement on line 10 is executed.

By convention, the function main returns the value zero to indicate success and returns a nonzero value to indicate failure. The include file cstdlib defines the constant EXIT_SUCCESS to be 0. Thus, the return statement on line 10 returns 0, indicating a successful termination.

The statement on line 6 prints a string using the output operator ("<<"). The statement on line 7 inputs the values of the variables x and y using the input operator (">>"). These variable values could be supplied, for example, by the person running our program. The name std::cout indicates that output is to be sent to the ***standard output stream***. There are two other important I/O streams in C++: ***standard input*** is where input is typically read, and ***standard error*** is where error output is written. These are denoted std::cin and std::cerr, respectively.

The prefix "std::" indicates that these objects are from the system's ***standard library***. We should include this prefix when referring to objects from the standard library. Nonetheless, it is possible to inform the compiler that we wish to use objects from the standard library—and so omit this prefix—by utilizing the "**using**" statement as shown below.

```
#include <iostream>
using namespace std;                      // makes std:: available
// . . .
cout << "Please enter two numbers: ";    // (std:: is not needed)
cin >> x >> y;
```

We discuss the **using** statement later in Section 1.1.4. In order to keep our examples short, we often omit the **include** and **using** statements when displaying C++ code. We also use " //..." to indicate that some code has been omitted.

Returning to our simple example C++ program, we note that the statement on line 9 outputs the value of the variable sum, which in this case stores the computed sum of x and y. By default, the output statement does not produce an end of line. The special object std::endl generates a special end-of-line character. Another way to generate an end of line is to output the ***newline character***, '\n'.

If run interactively, that is, with the user inputing values when requested to do so, this program's output would appear as shown below. The user's input is indicated below in blue.

```
Please enter two numbers:  7 35
Their sum is 42
```

1.1.2 Fundamental Types

We continue our exploration of C++ by discussing the language's basic data types and how these types are represented as constants and variables. The fundamental

types are the basic building blocks from which more complex types are constructed. They include the following.

bool	Boolean value, either **true** or **false**
char	character
short	short integer
int	integer
long	long integer
float	single-precision floating-point number
double	double-precision floating-point number

There is also an enumeration, or **enum**, type to represent a set of discrete values. Together, enumerations and the types **bool**, **char**, and **int** are called *integral types*. Finally, there is a special type **void**, which explicitly indicates the absence of any type information. We now discuss each of these types in greater detail.

Characters

A **char** variable holds a single character. A **char** in C++ is typically 8-bits, but the exact number of bits used for a **char** variable is dependent on the particular implementation. By allowing different implementations to define the meaning of basic types, such as **char**, C++ can tailor its generated code to each machine architecture and so achieve maximum efficiency. This flexibility can be a source of frustration for programmers who want to write machine-independent programs, however.

A *literal* is a constant value appearing in a program. Character literals are enclosed in single quotes, as in 'a', 'Q', and '+'. A backslash (\ ) is used to specify a number of special character literals as shown below.

'\n'	newline	'\t'	tab
'\b'	backspace	'\0'	null
'\''	single quote	'\"'	double quote
'\\'	backslash		

The null character, '\0', is sometimes used to indicate the end of a string of characters. Every character is associated with an integer code. The function int(ch) returns the integer value associated with a character variable ch.

Integers

An **int** variable holds an integer. Integers come in three sizes: **short int**, (plain) **int**, and **long int**. The terms "**short**" and "**long**" are synonyms for "**short int**" and "**long int**," respectively. Decimal numbers such as 0, 25, 98765, and -3 are of type **int**. The suffix "l" or "L" can be added to indicate a long integer, as in 123456789L. Octal (base 8) constants are specified by prefixing the number with the zero digit, and hexadecimal (base 16) constants can be specified by prefixing the number with

"0x." For example, the literals 256, 0400, and 0x100 all represent the integer value 256 (in decimal).

When declaring a variable, we have the option of providing a *definition*, or initial value. If no definition is given, the initial value is unpredictable, so it is important that each variable be assigned a value before being used. Variable names may consist of any combination of letters, digits, or the underscore (_) character, but the first character cannot be a digit. Here are some examples of declarations of integral variables.

```
short n;                                // n's value is undefined
int   octalNumber = 0400;                // 400 (base 8) = 256 (base 10)
char newline_character = '\n';
long BIGnumber = 314159265L;
short _aSTRANGE__1234_variABlE_NaMe;
```

Although it is legal to start a variable name with an underscore, it is best to avoid this practice, since some C++ compilers use this convention for defining their own internal identifiers.

C++ does not specify the exact number of bits in each type, but a **short** is at least 16 bits, and a **long** is at least 32 bits. In fact, there is no requirement that **long** be strictly longer than **short** (but it cannot be shorter!). Given a type T, the expression **sizeof**(T) returns the size of type T, expressed as some number of multiples of the size of **char**. For example, on typical systems, a **char** is 8 bits long, and an **int** is 32 bits long, and hence **sizeof**(int) is 4.

Enumerations

An enumeration is a user-defined type that can hold any of a set of discrete values. Once defined, enumerations behave much like an integer type. A common use of enumerations is to provide meaningful names to a set of related values. Each element of an enumeration is associated with an integer value. By default, these values count up from 0, but it is also possible to define explicit constant values as shown below.

```
enum Day { SUN, MON, TUE, WED, THU, FRI, SAT };
enum Mood { HAPPY = 3, SAD = 1, ANXIOUS = 4, SLEEPY = 2 };

Day today = THU;            // today may be any of MON … SAT
Mood myMood = SLEEPY;    // myMood may be HAPPY, …, SLEEPY
```

Since we did not specify values, SUN would be associated with 0, MON with 1, and so on. As a hint to the reader, we write enumeration names and other constants with all capital letters.

Floating Point

A variable of type **float** holds a single-precision floating-point number, and a variable of type **double** holds a double-precision floating-point number. As it does with integers, C++ leaves undefined the exact number of bits in each of the floating point types. By default, floating point literals, such as 3.14159 and -1234.567 are of type **double**. Scientific or exponential notation may by specified using either "e" or "E" to separate the mantissa from the exponent, as in 3.14E5, which means 3.14×10^5. To force a literal to be a **float**, add the suffix "f" or "F," as in 2.0f or 1.234e-3F.

1.1.3 Pointers, Arrays, and Structures

We next discuss how to combine fundamental types to form more complex ones.

Pointers

Each program variable is stored in the computer's memory at some location, or *address*. A *pointer* is a variable that holds the value of such an address. Given a type T, the type T* denotes a pointer to a variable of type T. For example, **int*** denotes a pointer to an integer.

Two essential operators are used to manipulate pointers. The first returns the address of an object in memory, and the second returns the contents of a given address. In C++ the first task is performed by the *address-of* operator, &. For example if x is an integer variable in your program &x is the address of x in memory. Accessing an object's value from its address is called *dereferencing*. This is done using the * operator. For example, if we were to declare q to be a pointer to an integer (that is, int*) and then set q = &x, we could access x's value with *q. Assigning an integer value to *q effectively changes the value of x.

Consider, for example, the code fragment below. The variable p is declared to be a pointer to a **char**, and is initialized to point to the variable ch. Thus, *p is another way of referring to ch. Observe that when the value of ch is changed, the value of *p changes as well.

```
char ch = 'Q';
char* p = &ch;            // p holds the address of ch
cout << *p;               // outputs the character 'Q'
ch = 'Z';                 // ch now holds 'Z'
cout << *p;               // outputs the character 'Z'
*p = 'X';                 // ch now holds 'X'
cout << ch;               // outputs the character 'X'
```

We shall see that pointers are very useful when building data structures where objects are linked to one another through the use of pointers. Pointers need not point

only to fundamental types, such as **char** and **int**—they may also point to complex types and even to functions. Indeed, the popularity of C++ stems in part from its ability to handle low-level entities like pointers.

It is useful to have a pointer value that points to nothing, that is, a ***null pointer***. By convention, such a pointer is assigned the value zero. An attempt to dereference a null pointer results in a run-time error. All C++ implementations define a special symbol NULL, which is equal to zero. This definition is activated by inserting the statement "#include <cstdlib>" in the beginning of a program file.

We mentioned earlier that the special type **void** is used to indicate no type information at all. Although we cannot declare a variable to be of type **void**, we can declare a pointer to be of type **void***. Such a pointer can point to a variable of ***any*** type. Since the compiler is unable to check the correctness of such references, the use of **void*** pointers is strongly discouraged, except in unusual cases where direct access to the computer's memory is needed.

Beware when declaring two or more pointers on the same line. The * operator binds with the variable name, not with the type name. Consider the following misleading declaration.

Caution

```
int* x, y, z;            // same as: int* x; int y; int z;
```

This declares one pointer variable x, but the other two variables are plain integers. The simplest way to avoid this confusion is to declare one variable per statement.

Arrays

An ***array*** is a collection of elements of the same type. Given any type T and a constant N, a variable of type T[N] holds an array of N elements, each of type T. Each element of the array is referenced by its ***index***, that is, a number from 0 to $N - 1$. The following statements declare two arrays; one holds three doubles and the other holds 10 double pointers.

```
double f[5];            // array of 5 doubles: f[0], ..., f[4]
int m[10];              // array of 10 ints: m[0], ..., m[9]
f[4] = 2.5;
m[2] = 4;
cout << f[m[2]];        // outputs f[4], which is 2.5
```

Once declared, it is not possible to increase the number of elements in an array. Also, C++ provides no built-in run-time checking for array subscripting out of bounds. This decision is consistent with C++'s general philosophy of not introducing any feature that would slow the execution of a program. Indexing an array outside of its declared bounds is a common programming error. Such an error often occurs "silently," and only much later are its effects noticed. In Section 1.5.5,

we see that the vector type of the C++ Standard Template Library (STL) provides many of the capabilities of a more complete array type, including run-time index checking and the ability to dynamically change the array's size.

A two-dimensional array is implemented as an "array of arrays." For example "int A[15][30]" declares A to be an array of 30 objects, each of which is an array of 15 integers. An element in such an array is indexed as A[i][j], where i is in the range 0 to 14 and j is in the range 0 to 29.

When declaring an array, we can initialize its values by enclosing the elements in curly braces ({...}). When doing so, we do not have to specify the size of the array, since the compiler can figure this out.

```
int a[]  = {10, 11, 12, 13};      // declares and initializes a[4]
bool b[] = {false, true};         // declares and initializes b[2]
char c[] = {'c', 'a', 't'};       // declares and initializes c[3]
```

Just as it is possible to declare an array of integers, it is possible to declare an array of pointers to integers. For example, int* r[17] declares an array r consisting of 17 pointers to objects of type **int**. Once initialized, we can dereference an element of this array using the * operator, for example, *r[16] is the value of the integer pointed to by the last element of this array.

Pointers and Arrays

There is an interesting connection between arrays and pointers, which C++ inherited from the C programming language—the name of an array is equivalent to a pointer to the array's initial element and vice versa. In the example below, c is an array of characters, and p and q are pointers to the first element of c. They all behave essentially the same, however.

```
char c[] = {'c', 'a', 't'};
char* p = c;                      // p points to c[0]
char* q = &c[0];                  // q also points to c[0]
cout << c[2] << p[2] << q[2];     // outputs "ttt"
```

This equivalence between array names and pointers can be confusing, but it helps to explain many of C++'s apparent mysteries. For example, given two arrays c and d, the comparison (c == d) does not test whether the contents of the two arrays are equal. Rather it compares the addresses of their initial elements, which is probably not what the programmer had in mind. If there is a need to perform operations on entire arrays (such as copying one array to another) it is a good idea to use the vector class, which is part of C++'s Standard Template Library. We discuss these concepts in Section 1.5.5.

Strings

A string literal, such as `"Hello World"`, is represented as a fixed-length array of characters that ends with the null character. Character strings represented in this way are called ***C-style strings***, since they were inherited from C. Unfortunately, this representation alone does not provide many string operations, such as concatenation and comparison. It also possesses all the peculiarities of C++ arrays, as mentioned earlier.

For this reason, C++ provides a `string` type as part of its Standard Template Library (STL). When we need to distinguish, we call these ***STL strings***. In order to use STL strings it is necessary to include the header file `<string>`. Since STL strings are part of the standard namespace (see Section 1.1.4), their full name is std::string. By adding the statement "**using** std::string," we inform the compiler that we want to access this definition directly, so we can omit the "std::" prefix. STL strings may be concatenated using the $+$ operator, they may be compared with each other using lexicographic (or dictionary) order, and they may be input and output using the `>>` and `<<` operators, respectively. For example:

```
#include <string>
using std::string;
// ...
string s = "to be";
string t = "not " + s;              // t = "not to be"
string u = s + " or " + t;          // u = "to be or not to be"
if (s > t)                          // true: "to be" > "not to be"
   cout << u;                       // outputs "to be or not to be"
```

There are other STL string operations, as well. For example, we can append one string to another using the `+=` operator. Also, strings may be indexed like arrays and the number of characters in a string s is given by s.size(). Since some library functions require the old C-style strings, there is a conversion function s.c_str(), which returns a pointer to a C-style string. Here are some examples:

```
string s = "John";                  // s = "John"
int i = s.size();                   // i = 4
char c = s[3];                      // c = 'n'
s += " Smith";                      // now s = "John Smith"
```

The C++ STL provides many other string operators including operators for extracting, searching for, and replacing substrings. We discuss some of these in Section 1.5.5.

C-Style Structures

A ***structure*** is useful for storing an aggregation of elements. Unlike an array, the elements of a structure may be of different types. Each ***member***, or ***field***, of a

structure is referred to by a given name. For example, consider the following structure for storing information about an airline passenger. The structure includes the passenger's name, meal preference, and information as to whether this passenger is in the frequent flyer program. We create an enumerated type to handle meal preferences.

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

struct Passenger {
    string      name;          // passenger name
    MealType    mealPref;      // meal preference
    bool        isFreqFlyer;   // in the frequent flyer program?
    string      freqFlyerNo;   // the passenger's freq. flyer number
};
```

This defines a new type called Passenger. Let us declare and initialize a variable named "pass" of this type.

```
Passenger pass = { "John Smith", VEGETARIAN, true, "293145" };
```

The individual members of the structure are accessed using the ***member selection operator***, which has the form struct_name.member. For example, we could change some of the above fields as follows.

```
pass.name = "Pocahontas";     // change name
pass.mealPref = REGULAR;      // change meal preference
```

Structures of the same type may be assigned to one another. For example, if p1 and p2 are of type Passenger, then p2 = p1 copies the elements of p1 to p2.

What we have discussed so far might be called a ***C-style structure***. C++ provides a much more powerful and flexible construct called a class, in which both data and functions can be combined. We discuss classes in Section 1.5.

Pointers, Dynamic Memory, and the "new" Operator

We often find it useful in data structures to create objects dynamically as the need arises. The C++ run-time system reserves a large block of memory called the ***free store***, for this reason. (This memory is also sometimes called ***heap memory***, but this should not be confused with the heap data structure, which is discussed in Chapter 8.) The operator **new** dynamically allocates the correct amount of storage for an object of a given type from the free store and returns a pointer to this object. That is, the value of this pointer is the address where this object resides in memory. Indeed, C++ allows for pointer variables to any data type, even to other pointers or to individual cells in an array.

For example, suppose that in our airline system we encounter a new passenger. We would like to dynamically create a new instance using the **new** operator. Let p be a pointer to a Passenger structure. This implies that *p refers to the actual structure; hence, we could access one of its members, say the mealPref field, using the expression (*p).mealPref. Because complex objects like structures are often allocated dynamically, C++ provides a shorter way to access members using the "->" operator.

pointer_name->member   is equivalent to   (*pointer_name).member

For example, we could allocate a new passenger object and initialize its members as follows.

```
Passenger *p;
// . . .
p = new Passenger;              // p points to the new Passenger
p->name = "Pocahontas";        // set the structure members
p->mealPref = REGULAR;
p->isFreqFlyer = false;
p->freqFlyerNo = "NONE";
```

It would be natural to wonder whether we can initialize the members using the curly brace ({...}) notation used above. The answer is no, but we will see another more convenient way of initializing members when we discuss classes and constructors in Section 1.5.2.

This new passenger object continues to exist in the free store until it is explicitly deleted—a process that is done using the **delete** operator, which destroys the object and returns its space to the free store.

```
delete p;                      // destroy the object p points to
```

The **delete** operator should only be applied to objects that have been allocated through **new**. Since the object at p's address was allocated using the **new** operator, the C++ run-time system knows how much memory to deallocate for this **delete** statement. Unlike some programming languages such as Java, C++ does not provide automatic *garbage collection*. This means that C++ programmers have the responsibility of explicitly deleting all dynamically allocated objects.

Arrays can also be allocated with **new**. When this is done, the system allocator returns a pointer to the first element of the array. Thus, a dynamically allocated array with elements of type T would be declared being of type *T. Arrays allocated in this manner cannot be deallocated using the standard **delete** operator. Instead, the operator delete[ ] is used. Here is an example that allocates a character buffer of 500 elements, and then later deallocates it.

```
char* buffer = new char[500];   // allocate a buffer of 500 chars
buffer[3] = 'a';                // elements are still accessed using []
delete [] buffer;               // delete the buffer
```

Memory Leaks

Failure to delete dynamically allocated objects can cause problems. If we were to change the (address) value of p without first deleting the structure to which it points, there would be no way for us to access this object. It would continue to exist for the lifetime of the program, using up space that could otherwise be used for other allocated objects. Having such inaccessible objects in dynamic memory is called a ***memory leak***. We should strongly avoid memory leaks, especially in programs that do a great deal of memory allocation and deallocation. A program with memory leaks can run out of usable memory even when there is a sufficient amount of memory present. An important rule for a disciplined C++ programmer is the following:

<div style="border-left: 4px solid navy; padding-left: 8px;">Remember</div>

> If an object is allocated with **new**, it should eventually be deallocated with **delete**.

References

Pointers provide one way to refer indirectly to an object. Another way is through references. A ***reference*** is simply an alternative name for an object. Given a type T, the notation T& indicates a reference to an object of type T. Unlike pointers, which can be NULL, a reference in C++ must refer to an actual variable. When a reference is declared, its value must be initialized. Afterwards, any access to the reference is treated exactly as if it is an access to the underlying object.

```
string author = "Samuel Clemens";
string& penName = author;        // penName is an alias for author
penName = "Mark Twain";          // now author = "Mark Twain"
cout << author;                  // outputs "Mark Twain"
```

References are most often used for passing function arguments and are also often used for returning results from functions. These uses are discussed later.

1.1.4 Named Constants, Scope, and Namespaces

We can easily name variables without concern for naming conflicts in small problems. It is much harder for us to avoid conflicts in large software systems, which may consist of hundreds of files written by many different programmers. C++ has a number of mechanisms that aid in providing names and limiting their scope.

Constants and Typedef

Good programmers commonly like to associate names with constant quantities. By adding the keyword **const** to a declaration, we indicate that the value of the associated object cannot be changed. Constants may be used virtually anywhere that literals can be used, for example, in an array declaration. As a hint to the reader, we will use all capital letters when naming constants.

```
const double PI        = 3.14159265;
const int    CUT_OFF[] = {90, 80, 70, 60};
const int    N_DAYS    = 7;
const int    N_HOURS   = 24*N_DAYS; // using a constant expression
int counter[N_HOURS];                       // an array of 168 ints
```

Note that enumerations (see Section 1.1.2) provide another convenient way to define integer-valued constants, especially within structures and classes.

In addition to associating names with constants, it is often useful to associate a name with a type. This association can be done with a **typedef** declaration. Rather than declaring a variable, a **typedef** defines a new type name.

```
typedef char* BufferPtr;            // type BufferPtr is a pointer to char
typedef double Coordinate;          // type Coordinate is a double

BufferPtr p;                        // p is a pointer to char
Coordinate x, y;                    // x and y are of type double
```

By using **typedef** we can provide shorter or more meaningful synonyms for various types. The type name Coordinate provides more of a hint to the reader of the meaning of variables x and y than does **double**. Also, if later we decide to change our coordinate representation to **int**, we need only change the **typedef** statement. We will follow the convention of indicating user-defined types by capitalizing the first character of their names.

Local and Global Scopes

When a group of C++ statements are enclosed in curly braces ({...}), they define a *block*. Variables and types that are declared within a block are only accessible from within the block. They are said to be *local* to the block. Blocks can be nested within other blocks. In C++, a variable may be declared outside of any block. Such a variable is *global*, in the sense that it is accessible from everywhere in the program. The portions of a program from which a given name is accessible are called its *scope*.

Two variables of the same name may be defined within nested blocks. When this happens, the variable of the inner block becomes active until leaving the block.

Thus a local variable "hides" any global variables of the same name as shown in the following example.

```cpp
const int Cat = 1;                      // global Cat

int main() {
  const int Cat = 2;                    // this Cat is local to main
  cout << Cat;                          // outputs 2 (local Cat)
  return EXIT_SUCCESS;
}

int dog = Cat;                          // dog = 1 (from the global Cat)
```

Namespaces

Global variables present many problems in large software systems because they can be accessed and possibly modified anywhere in the program. They also can lead to programming errors, since an important global variable may be hidden by a local variable of the same name. As a result, it is best to avoid global variables. We may not be able to avoid globals entirely, however. For example, when we perform output, we actually use the system's global standard output stream object, cout. If we were to define a variable with the same name, then the system's cout stream would be inaccessible.

A *namespace* is a mechanism that allows a group of related names to be defined in one place. This helps organize global objects into natural groups and minimizes the problems of globals. For example, the following declares a namespace myglobals containing two variables, cat and dog.

```cpp
namespace myglobals {
  int cat;
  string dog = "bow wow";
}
```

Namespaces may generally contain definitions of more complex objects, including types, classes, and functions. We can access an object x in namespace group, using the notation group::x, which is called its *fully qualified name*. For example, myglobals::cat refers to the copy of variable cat in the myglobals namespace.

We have already seen an example of a namespace. Many standard system objects, such as the standard input and output streams cin and cout, are defined in a system namespace called std. Their fully qualified names are std::cin and std::cout, respectively.

The Using Statement

If we are repeatedly using variables from the same namespace, it is possible to avoid entering namespace specifiers by telling the system that we want to "use" a particular specifier. We communicate this desire by utilizing the **using** statement, which makes some or all of the names from the namespace accessible, without explicitly providing the specifier. This statement has two forms that allow us to list individual names or to make every name in the namespace accessible as shown below.

```
using std::string;                // makes just std::string accessible
using std::cout;                  // makes just std::cout accessible

using namespace myglobals;        // makes all of myglobals accessible
```

1.2 Expressions

An *expression* combines variables and literals with operators to create new values. In the following discussion, we group operators according to the types of objects they may be applied to. Throughout, we use var to denote a variable or anything to which a value may be assigned. (In official C++ jargon, this is called an *lvalue*.) We use exp to denote an expression and type to denote a type.

Member Selection and Indexing

Some operators access a member of a structure, class, or array. We let class_name denote the name of a structure or class; pointer denotes a pointer to a structure or class and array denotes an array or a pointer to the first element of an array.

class_name . member	class/structure member selection
pointer −> member	class/structure member selection
array [exp]	array subscripting

Arithmetic Operators

The following are the binary arithmetic operators:

exp + exp	addition
exp − exp	subtraction
exp * exp	multiplication
exp / exp	division
exp % exp	modulo (remainder)

There are also unary minus (−x) and unary plus (+x) operations. Division between two integer operands results in an integer result by truncation, even if the

result is being assigned to a floating point variable. The modulo operator n%m yields the remainder that would result from the integer division n/m.

Increment and Decrement Operators

The ***post-increment*** operator returns a variable's value and then increments it by 1. The post-decrement operator is analogous but decreases the value by 1. The ***pre-increment*** operator first increments the variables and then returns the value.

$$
\begin{array}{ll}
\text{var } ++ & \text{post increment} \\
\text{var } -- & \text{post decrement} \\
++ \text{ var} & \text{pre increment} \\
-- \text{ var} & \text{pre decrement}
\end{array}
$$

The following code fragment illustrates the increment and decrement operators.

```
int a[] = {0, 1, 2, 3};
int i = 2;
int j = i++;              // j = 2 and now i = 3
int k = −−i;              // now i = 2 and k = 2
cout << a[k++];           // a[2] (= 2) is output; now k = 3
```

Relational and Logical Operators

C++ provides the usual comparison operators.

$$
\begin{array}{ll}
\text{exp} < \text{exp} & \text{less than} \\
\text{exp} > \text{exp} & \text{greater than} \\
\text{exp} <= \text{exp} & \text{less than or equal} \\
\text{exp} >= \text{exp} & \text{greater than or equal} \\
\text{exp} == \text{exp} & \text{equal to} \\
\text{exp} \mathrel{!=} \text{exp} & \text{not equal to}
\end{array}
$$

These return a Boolean result—either **true** or **false**. Comparisons can be made between numbers, characters, and STL strings (but not C-style strings). Pointers can be compared as well, but it is usually only meaningful to test whether pointers are equal or not equal (since their values are memory addresses).

The following logical operators are also provided.

$$
\begin{array}{ll}
! \text{ exp} & \text{logical not} \\
\text{exp \&\& exp} & \text{logical and} \\
\text{exp} \mathbin{||} \text{exp} & \text{logical or}
\end{array}
$$

The operators && and || evaluate sequentially from left to right. If the left operand of && is false, the entire result is false, and the right operand is not evaluated. The || operator is analogous, but evaluation stops if the left operand is true.

This "short circuiting" is quite useful in evaluating a chain of conditional expressions where the left condition guards against an error committed by the right

condition. For example, the following code first tests that a Passenger pointer p is non-null before accessing it. It would result in an error if the execution were not stopped if the first condition is not satisfied.

```
if ((p != NULL) && p->isFreqFlyer) ...
```

Bitwise Operators

The following operators act on the representations of numbers as binary bit strings. They can be applied to any integer type, and the result is an integer type.

˜ exp	bitwise complement
exp & exp	bitwise and
exp ˆ exp	bitwise exclusive-or
exp | exp	bitwise or
exp1 << exp2	shift exp1 left by exp2 bits
exp1 >> exp2	shift exp1 right by exp2 bits

The left shift operator always fills with zeros. How the right shift fills depends on a variable's type. In C++ integer variables are "signed" quantities by default, but they may be declared as being "unsigned," as in "**unsigned int** x." If the left operand of a right shift is unsigned, the shift fills with zeros and otherwise the right shift fills with the number's sign bit (0 for positive numbers and 1 for negative numbers). Note that the input (>>) and output (<<) operators are not in this group. They are discussed later.

Assignment Operators

In addition to the familiar assignment operator (=), C++ includes a special form for each of the arithmetic binary operators (+, −, *, /, %) and each of the bit-wise binary operators (&, |, ˆ, <<, >>), that combines a binary operation with assignment. For example, the statement "n += 2" means "n = n + 2." Some examples are shown below.

```
int     i = 10;
int     j = 5;
string  s = "yes";
i    −=   4;              // i = i - 4 = 6
j    *=  −2;              // j = j * (-2) = -10
s    +=   " or no";      // s = s + " or no" = "yes or no"
```

These assignment operators not only provide notational convenience, but they can be more efficient to execute as well. For example, in the string concatenation example above, the new text can just be appended to s without the need to generate a temporary string to hold the intermediate result.

Take care when performing assignments between aggregate objects (arrays, strings, and structures). Typically the programmer intends such an assignment to copy the contents of one object to the other. This works for STL strings and C-style structures (provided they have the same type). However, as discussed earlier, C-style strings and arrays cannot be copied merely through a single assignment statement.

Other Operators

Here are some other useful operators.

class_name :: member	class scope resolution
namespace_name :: member	namespace resolution
bool_exp ? true_exp : false_exp	conditional expression

We have seen the namespace resolution operator in Section 1.1.4. The conditional expression is a variant of "if-then-else" for expressions. If bool_exp evaluates to true, the value of true_exp is returned, and otherwise the value of false_exp is returned.

The following example shows how to use this to return the minimum of two numbers, x and y.

```
smaller = (x < y ? x : y);          // smaller = min(x,y)
```

We also have the following operations on input/output streams.

stream >> var	stream input
stream << exp	stream output

Although they look like the bitwise shift operators, the input (>>) and output (<<) stream operators are quite different. They are examples of C++'s powerful capability, called *operator overloading*, which are discussed in Section 1.4.2. These operators are not an intrinsic part of C++, but are provided by including the file <iostream>. We refer the reader to the references given in the chapter notes for more information on input and output in C++.

The above discussion provides a somewhat incomplete list of all the C++ operators, but it nevertheless covers the most common ones. Later we introduce others, including casting operators.

Operator Precedence

Operators in C++ are assigned a *precedence* that determines the order in which operations are performed in the absence of parentheses. In Table 1.1, we show the precedence of some of the more common C++ operators, with the highest listed first. Unless parentheses are used, operators are evaluated in order from highest

to lowest. For example, the expression $0 < 4 + x * 3$ would be evaluated as if it were parenthesized as $0 < (4 + (x * 3))$. If p is an array of pointers, then *p[2] is equivalent to *(p[2]). Except for && and ||, which guarantee left-to-right evaluation, the order of evaluation of subexpressions is dependent on the implementation. Since these rules are complex, it is a good idea to add parentheses to complex expressions to make your intent clear to someone reading your program.

Operator Precedences

Type	*Operators*
scope resolution	namespace_name :: member
selection/subscripting function call postfix operators	class_name.member pointer−>member array[exp] function(args) var++ var−−
prefix operators dereference/address	++var −−var +exp −exp ˜exp !exp *pointer &var
multiplication/division	* / %
addition/subtraction	+ −
shift	<< >>
comparison	< <= > >=
equality	== !=
bitwise and	&
bitwise exclusive-or	ˆ
bitwise or		
logical and	&&
logical or	||
conditional	bool_exp ? true_exp : false_exp
assignment	= += −= *= /= %= >>= <<= &= ˆ= |=

Table 1.1: The C++ precedence rules. The notation "**exp**" denotes any expression.

1.2.1   Changing Types through Casting

Casting is an operation that allows us to change the type of a variable. In essence, we can take a variable of one type and *cast* it into an equivalent variable of another type. Casting is useful in many situations. There are two fundamental types of casting that can be done in C++. We can either cast with respect to the fundamental types or we can cast with respect to class objects and pointers. We discuss casting with fundamental types here, and we consider casting with objects in Section 2.2.4. We begin by introducing the traditional way of casting in C++, and later we present C++'s newer casting operators.

Traditional C-Style Casting

Let exp be some expression, and let T be a type. To cast the value of the expression to type T we can use the notation "(T)exp." We call this a ***C-style cast***. If the desired type is a type name (as opposed to a type expression), there is an alternate ***functional-style cast***. This has the form "T(exp)." Some examples are shown below. In both cases, the integer value 14 is cast to a double value 14.0.

```
int     cat = 14;
double  dog = (double) cat;              // traditional C-style cast
double  pig = double(cat);               // C++ functional cast
```

Both forms of casting are legal, but some authors prefer the functional-style cast.

Casting to a type of higher precision or size is often needed in forming expressions. The results of certain binary operators depend on the variable types involved. For example, division between integers always produces an integer result by truncating the fractional part. If a floating-point result is desired, we must cast the operands ***before*** performing the operation as shown below.

```
int     i1  = 18;
int     i2  = 16;
double  dv1 = i1 / i2;                    // dv1 has value 1.0
double  dv2 = double(i1) / double(i2);    // dv2 has value 1.125
double  dv3 = double( i1 / i2 );          // dv3 has value 1.0
```

When i1 and i2 are cast to doubles, double-precision division is performed. When i1 and i2 are not cast, truncated integer division is performed. In the case of dv3, the cast is performed after the integer division, so precision is still lost.

Explicit Cast Operators

Casting operations can vary from harmless to dangerous, depending on how similar the two types are and whether information is lost. For example, casting a **short** to an **int** is harmless, since no information is lost. Casting from a **double** to an **int** is more dangerous because the fractional part of the number is lost. Casting from a **double*** to **char*** is dangerous because the meaning of converting such a pointer will likely vary from machine to machine. One important element of good software design is that programs be ***portable***, meaning that they behave the same on different machines.

For this reason, C++ provides a number of casting operators that make the safety of the cast much more explicit. These are called the **static_cast**, **dynamic_cast**, **const_cast**, and **reinterpret_cast**. We discuss only the **static_cast** here and consider the others as the need arises.

Static Casting

Static casting is used when a conversion is made between two related types, for example numbers to numbers or pointers to pointers. Its syntax is given below.

> **static_cast** < *desired_type* > (*expression*)

The most common use is for conversions between numeric types. Some of these conversions may involve the loss of information, for example a conversion from a **double** to an **int**. This conversion is done by truncating the fractional part (not rounding). For example, consider the following:

```
double  d1 = 3.2;
double  d2 = 3.9999;
int     i1 = static_cast<int>(d1);      // i1 has value 3
int     i2 = static_cast<int>(d2);      // i2 has value 3
```

This type of casting is more verbose than the C-style and functional-style casts shown earlier. But this form is appropriate, because it serves as a visible warning to the programmer that a potentially unsafe operation is taking place. In our examples in this book, we use the functional style for safe casts (such as integer to double) and these newer cast operators for all other casts. Some older C++ compilers may not support the newer cast operators, but then the traditional C-style and functional-style casts can be used instead.

Implicit Casting

There are many instances where the programmer has not requested an *explicit cast*, but a change of types is required. In many of these cases, C++ performs an *implicit cast*. That is, the compiler automatically inserts a cast into the machine-generated code. For example, when numbers of different types are involved in an operation, the compiler automatically casts to the stronger type. C++ allows an assignment that implicitly loses information, but the compiler usually issues a warning message.

```
int    i  = 3;
double d  = 4.8;
double d3 = i / d;       // d3 = 0.625 = double(i)/d
int    i3 = d3;          // i3 = 0 = int(d3)
                         // Warning! Assignment may lose information
```

A general rule with casting is to "play it safe." If a compiler's behavior regarding the implicit casting of a value is uncertain, then we are safest in using an explicit cast. Doing so makes our intentions clear.

1.3   Control Flow

Control flow in C++ is similar to that of other high-level languages. We review the basic structure and syntax of control flow in C++ in this section, including method returns, if statements, switch statements, loops, and restricted forms of "jumps" (the **break** and **continue** statements).

If Statement

Every programming language includes a way of making choices, and C++ is no exception. The most common method of making choices in a C++ program is through the use of an *if statement*. The syntax of an *if statement* in C++ is shown below, together with a small example.

```
if ( condition )
    true_statement
else if ( condition )
    else_if_statement
else
    else_statement
```

Each of the conditions should return a Boolean result. Each statement can either be a single statement or a block of statements enclosed in braces ({...}). The "**else if**" and "**else**" parts are optional, and any number of else-if parts may be given. The conditions are tested one by one, and the statement associated with the first true condition is executed. All the other statements are skipped. Here is a simple example.

```
if ( snowLevel < 2 ) {
    goToClass();                // do this if snow level is less than 2
    comeHome();
}
else if ( snowLevel < 5 )
    haveSnowballFight();        // if level is at least 2 but less than 5
else if ( snowLevel < 10 )
    goSkiing();                 // if level is at least 5 but less than 10
else
    stayAtHome();               // if snow level is 10 or more
```

Switch Statement

A *switch statement* provides an efficient way to distinguish between many different options according to the value of an integral type. In the following example, a

single character is input, and based on the character's value, an appropriate editing function is called. The comments explain the equivalent if-then-else structure, but the compiler is free to select the most efficient way to execute the statement.

```cpp
char command;
cin >> command;              // input command character
switch (command) {           // switch based on command value
  case 'I' :                 // if (command == 'I')
    editInsert();
    break;
  case 'D' :                 // else if (command == 'D')
    editDelete();
    break;
  case 'R' :                 // else if (command == 'R')
    editReplace();
    break;
  default :                  // else
    cout << "Unrecognized command\n";
    break;
}
```

The argument of the **switch** can be any integral type or enumeration. The "**default**" case is executed if none of the cases equals the switch argument.

Each case in a switch statement should be terminated with a **break** statement, which, when executed, exits the switch statement. Otherwise, the flow of control "falls through" to the next case.

While and Do-While Loops

C++ has two kinds of conditional loops for iterating over a set of statements as long as some specified condition holds. These two loops are the standard *while loop* and the *do-while loop*. One loop tests a Boolean condition before performing an iteration of the loop body and the other tests a condition after. Let us consider the while loop first.

while (*condition*)
    loop_body_statement

At the beginning of each iteration, the loop tests the Boolean expression and then executes the loop body only if this expression evaluates to true. The loop body statement can also be a block of statements.

Consider the following example. It computes the sum of the elements of an array, until encountering the first negative value. Note the use of the += operator to increment the value of sum and the ++ operator which increments i after accessing

the current array element.

```
int a[100];
// ...
int i = 0;
int sum = 0;
while (i < 100 && a[i] >= 0) {
  sum += a[i++];
}
```

The do-while loop is similar to the while loop in that the condition is tested at the end of the loop execution rather than before. It has the following syntax.

```
do
    loop_body_statement
while ( condition )
```

For Loop

Many loops involve three common elements: an initialization, a condition under which to continue execution, and an increment to be performed after each execution of the loop's body. A *for loop* conveniently encapsulates these three elements.

```
for ( initialization ; condition ; increment )
    loop_body_statement
```

The *initialization* indicates what is to be done before starting the loop. Typically, this involves declaring and initializing a loop-control variable or counter. Next, the *condition* gives a Boolean expression to be tested in order for the loop to continue execution. It is evaluated before executing the loop body. When the condition evaluates to **false**, execution jumps to the next statement after the for loop. Finally, the *increment* specifies what changes are to be made at the end of each execution of the loop body. Typically, this involves incrementing or decrementing the value of the loop-control variable.

Here is a simple example, which prints the positive elements of an array, one per line. Recall that '\n' generates a newline character.

```
const int NUM_ELEMENTS = 100;
double b[NUM_ELEMENTS];
// ...
for (int i = 0; i < NUM_ELEMENTS; i++) {
  if (b[i] > 0)
    cout << b[i] << '\n';
}
```

In this example, the loop variable i was declared as int i = 0. Before each iteration, the loop tests the condition "i < NUM_ELEMENTS" and executes the loop body

only if this is true. Finally, at the end of each iteration the loop uses the statement i++ to increment the loop variable i before testing the condition again. Although the loop variable is declared outside the curly braces of the for loop, the compiler treats it as if it were a local variable within the loop. This implies that its value is not accessible outside the loop.

Break and Continue Statements

C++ provides statements to change control flow, including the **break**, **continue**, and **return** statements. We discuss the first two here, and leave the **return** statement for later. A break statement is used to "break" out of a loop or switch statement. When it is executed, it causes the flow of control to immediately exit the innermost switch statement or loop (for loop, while loop, or do-while loop). The break statement is useful when the condition for terminating the loop is determined inside the loop. For example, in an input loop, termination often depends on a specific value that has been input. The following example provides a different implementation of an earlier example, which sums the elements of an array until finding the first negative value.

```
int a[100];
// ...
int sum = 0;
for (int i = 0; i < 100; i++) {
  if (a[i] < 0) break;
  sum += a[i];
}
```

The other statement that is often useful for altering loop behavior is the **continue** statement. The continue statement can only be used inside loops (**for**, **while**, and **do-while**). The continue statement causes the execution to skip to the end of the loop, ready to start a new iteration.

1.4   Functions

A *function* is a chunk of code that can be called to perform some well-defined task, such as calculating the area of a rectangle, computing the weekly withholding tax for a company employee, or sorting a list of names in ascending order. In order to define a function, we need to provide the following information to the compiler:

Return type. This specifies the type of value or object that is returned by the function. For example, a function that computes the area of a rectangle might return a value of type **double**. A function is not required to return a value. For example, it may simply produce some output or modify some data structure.

If so, the return type is **void**. A function that returns no value is sometimes called a *procedure*.

Function name. This indicates the name that is given to the function. Ideally, the function's name should provide a hint to the reader as to what the function does.

Argument list. This serves as a list of placeholders for the values that will be passed into the function. The actual values will be provided when the function is invoked. For example, a function that computes the area of a polygon might take four **double** arguments; the *x*- and *y*-coordinates of the rectangle's lower left corner and the *x*- and *y*-coordinates of the rectangle's upper right corner. The argument list is given as a comma-separated list enclosed in parentheses, where each entry consists of the name of the argument and its type. A function may have any number of arguments, and the argument list may even be empty.

Function body. This is a collection of C++ statements that define the actual computations to be performed by the function. This is enclosed within curly braces. If the function returns a value, the body will typically end with a **return** statement, which specifies the final function value.

Function specifications in C++ typically involve two steps, declaration and definition. A function is *declared*, by specifying three things: the function's return type, its name, and its argument list. The declaration makes the compiler aware of the function's existence, and allows the compiler to verify that the function is being used correctly. This three-part combination of return type, function name, and argument types is called the function's *signature* or *prototype*.

For example, suppose that we wanted to create a function, called evenSum, that is given two arguments, an integer array a and its length n. It determines whether the sum of array values is even, and if so it returns the value **true**. Otherwise, it returns **false**. Thus, it has a return value of type **bool**. The function could be declared as follows:

```
bool evenSum(int a[], int n);                    // function declaration
```

Second, the function is *defined*. The definition consists both of the function's signature and the function body. The reason for distinguishing between the declaration and definition involves the manner in which large C++ programs are written. They are typically spread over many different files. The function declaration must appear in every file that invokes the function, but the definition must appear only

once. Here is how our evenSum function might be defined.

```
bool evenSum(int a[], int n) {            // function definition
  int sum = 0;
  for (int i = 0; i < n; i++)             // sum the array elements
    sum += a[i];
  return (sum % 2) == 0;                  // returns true if sum is even
}
```

The expression in the **return** statement may take a minute to understand. We use the mod operator (%) to compute the remainder when sum is divided by 2. If the sum is even, the remainder is 0, and hence the expression "(sum % 2) == 0" evaluates to **true**. Otherwise, it evaluates to **false**, which is exactly what we want.

To complete the example, let us provide a simple main program, which first declares the function, and then invokes it on an actual array.

```
bool evenSum(int a[], int n);             // function declaration

int main() {
  int list[] = {4, 2, 7, 8, 5, 1};
  bool result = evenSum(list, 6);         // invoke the function
  if (result)    cout << "the sum is even\n";
  else           cout << "the sum is odd\n";
  return EXIT_SUCCESS;
}
```

Let us consider this example in greater detail. The names "a" and "n" in the function definition are called *formal arguments* since they serve merely as place-holders. The variable "list" and literal "6" in the function call in the main program are the *actual arguments*. Thus, each reference to "a" in the function body is translated into a reference to the actual array "list." Similarly, each reference to "n" can be thought of as taking on the actual value 6 in the function body. The types of the actual arguments must agree with the corresponding formal arguments. Exact type agreement is not always necessary, however, for the compiler may perform implicit type conversions in some cases, such as casting a **short** actual argument to match an **int** formal argument.

When we refer to function names throughout this book, we often include a pair of parentheses following the name. This makes it easier to distinguish function names from variable names. For example, we would refer to the above function as evenSum.

1.4.1   Argument Passing

By default, arguments in C++ programs are passed *by value*. When arguments are passed by value, the system makes a copy of the variable to be passed to the

function. In the above example, the formal argument "n" is initialized to the actual value 6 when the function is called. This implies that modifications made to a formal argument in the function do not alter the actual argument.

Sometimes it is useful for the function to modify one of its arguments. To do so, we can explicitly define a formal argument to be a ***reference type*** (as introduced in Section 1.1.3). When we do this, any modifications made to an argument in the function modifies the corresponding actual argument. This is called passing the argument ***by reference***. An example is shown below, where one argument is passed by value and the other is passed by reference.

```
void f(int value, int& ref) {          // one value and one reference
  value++;                             // no effect on the actual argument
  ref++;                               // modifies the actual argument
  cout << value << endl;               // outputs 2
  cout << ref  << endl;                // outputs 6
}

int main() {
  int cat = 1;
  int dog = 5;
  f(cat, dog);                         // pass cat by value, dog by ref
  cout << cat << endl;                 // outputs 1
  cout << dog << endl;                 // outputs 6
  return EXIT_SUCCESS;
}
```

Observe that altering the value argument had no effect on the actual argument, whereas modifying the reference argument did.

Modifying function arguments is felt to be a rather sneaky way of passing information back from a function, especially if the function returns a nonvoid value. Another way to modify an argument is to pass the address of the argument, rather than the argument itself. Even though a pointer is passed by value (and, hence, the address of where it is pointing cannot be changed), we can access the pointer and modify the variables to which it points. Reference arguments achieve essentially the same result with less notational burden.

Constant References as Arguments

There is a good reason for choosing to pass structure and class arguments by reference. In particular, passing a large structure or class by value results in a copy being made of the entire structure. All this copying may be quite inefficient for large structures and classes. Passing such an argument by reference is much more efficient, since only the address of the structure need be passed.

Since most function arguments are not modified, an even better practice is to pass an argument as a "constant reference." Such a declaration informs the compiler

that, even though the argument is being passed by reference, the function cannot alter its value. Furthermore, the function is not allowed to pass the argument to another function that might modify its value. Here is an example using the Passenger structure, which we defined earlier in Section 1.1.3. The attempt to modify the argument would result in a compiler error message.

```
void someFunction(const Passenger& pass) {
  pass.name = "new name";            // ILLEGAL! pass is declared const
}
```

When writing small programs, we can easily avoid modifying the arguments that are passed by reference for the sake of efficiency. But in large programs, which may be distributed over many files, enforcing this rule is much harder. Fortunately, passing class and structure arguments as a constant reference allows the compiler to do the checking for us. Henceforth, when we pass a class or structure as an argument, we typically pass it as a reference, usually a constant reference.

Array Arguments

We have discussed passing large structures and classes by reference, but what about large arrays? Would passing an array by value result in making a copy of the entire array? The answer is no. When an array is passed to a function, it is converted to a pointer to its initial element. That is, an object of type T[ ] is converted to type T*. Thus, an assignment to an element of an array within a function does modify the actual array contents. In short, arrays are not passed by value.

By the same token, it is not meaningful to pass an array back as the result of a function call. Essentially, an attempt to do so will only pass a pointer to the array's initial element. If returning an array is our goal, then we should either explicitly return a pointer or consider returning an object of type vector from the C++ Standard Template Library.

1.4.2   Overloading and Inlining

Overloading means defining two or more functions or operators that have the same name, but whose effect depends on the types of their actual arguments.

Function Overloading

Function overloading occurs when two or more functions are defined with the same name but with different argument lists. Such definitions are useful in situations where we desire two functions that achieve essentially the same purpose, but do it with different types of arguments.

One convenient application of function overloading is in writing procedures that print their arguments. In particular, a function that prints an integer would be different from a function that prints a Passenger structure from Section 1.1.3, but both could use the same name, print, as shown in the following example.

```
void print(int x)                          // print an integer
  { cout << x; }

void print(const Passenger& pass) {   // print a Passenger
  cout << pass.name << " " << pass.mealPref;
  if (pass.isFreqFlyer)
    cout << " " << pass.freqFlyerNo;
}
```

When the print function is used, the compiler considers the types of the actual argument and invokes the appropriate function, that is, the one with signature closest to the actual arguments.

Operator Overloading

C++ also allows overloading of operators, such as +, *, +=, and <<. Not surprisingly, such a definition is called *operator overloading*. Suppose we would like to write an equality test for two Passenger objects. We can denote this in a natural way by overloading the == operator as shown below.

```
bool operator==(const Passenger& x, const Passenger& y) {
  return x.name          == y.name
      && x.mealPref       == y.mealPref
      && x.isFreqFlyer    == y.isFreqFlyer
      && x.freqFlyerNo    == y.freqFlyerNo;
}
```

This definition is similar to a function definition, but in place of a function name we use "operator==." In general, the == is replaced by whatever operator is being defined. For binary operators we have two arguments, and for unary operators we have just one.

There are several useful applications of function and operator overloading. For example, overloading the == operator allows us to naturally test for the equality of two objects, p1 and p2, with the expression "p1==p2." Another useful application of operator overloading is for defining input and output operators for classes and structures. Here is how to define an output operator for our Passenger structure. The type ostream is the system's output stream type. The standard output, cout is

of this type.

```
ostream& operator<<(ostream& out, const Passenger& pass) {
  out << pass.name << " " << pass.mealPref;
  if (pass.isFreqFlyer) {
    out << " " << pass.freqFlyerNo;
  }
  return out;
}
```

The output in this case is not very pretty, but we could easily modify our output operator to produce nicer formatting.

There is much more that could be said about function and operator overloading, and indeed C++ functions in general. We refer the reader to a more complete reference on C++ for this information.

Operator overloading is a powerful mechanism, but it is easily abused. It can be very confusing for someone reading your program to find that familiar operations such as "+" and "/" have been assigned new and possibly confusing meanings. Good programmers usually restrict operator overloading to certain general purpose operators such as " <<" (output), "=" (assignment), "==" (equality), "[ ]" (indexing, for sequences).

In-line Functions

Very short functions may be defined to be "**inline**." This is a hint to the compiler it should simply expand the function code in place, rather than using the system's call-return mechanism. As a rule of thumb, in-line functions should be very short (at most a few lines) and should not involve any loops or conditionals. Here is an example, which returns the minimum of two integers.

```
inline int min(int x, int y) { return (x < y ? x : y); }
```

1.5 Classes

The concept of a *class* is fundamental to C++, since it provides a way to define new user-defined types, complete with associated functions and operators. By restricting access to certain class members, it is possible to separate out the properties that are essential to a class's correct use from the details needed for its implementation. Classes are fundamental to programming that uses an object-oriented approach, which is a programming paradigm we discuss in the next chapter.

1.5.1 Class Structure

A class consists of *members*. Members that are variables or constants are *data members* (also called *member variables*) and members that are functions are called *member functions* (also called *methods*). Data members may be of any type, and may even be classes themselves, or pointers or references to classes. Member functions typically act on the member variables, and so define the behavior of the class.

We begin with a simple example, called Counter. It implements a simple counter stored in the member variable count. It provides three member functions. The first member function, called Counter, initializes the counter. The second, called getCount, returns the counter's current value. The third, called increaseBy, increases the counter's value.

```cpp
class Counter {                       // a simple counter
public:
  Counter();                          // initialization
  int getCount();                     // get the current count
  void increaseBy(int x);             // add x to the count
private:
  int count;                          // the counter's value
};
```

Let's explore this class definition in a bit more detail. Observe that the class definition is separated into two parts by the keywords **public** and **private**. The public section defines the class's *public interface*. These are the entities that users of the class are allowed to access. In this case, the public interface has the three member functions (Counter, getCount, and increaseBy). In contrast, the private section declares entities that cannot be accessed by users of the class. We say more about these two parts below.

So far, we have only declared the member functions of class Counter. Next, we present the definitions of these member functions. In order to make clear to the compiler that we are defining member functions of Counter (as opposed to member functions of some other class), we precede each function name with the scoping specifier "Counter::".

```cpp
Counter::Counter()                    // constructor
  { count = 0; }
int Counter::getCount()               // get current count
  { return count; }
void Counter::increaseBy(int x)       // add x to the count
  { count += x; }
```

The first of these functions has the same name as the class itself. This is a special member function called a *constructor*. A constructor's job is to initialize the values

of the class's member variables. The function getCount is commonly referred to as a "getter" function. Such functions provide access to the private members of the class.

Here is an example how we might use our simple class. We declare a new object of type Counter, called ctr. This implicitly invokes the class's constructor, and thus initializes the counter's value to 0. To invoke one of the member functions, we use the notation ctr.function_name().

```
Counter ctr;                        // an instance of Counter
cout << ctr.getCount() << endl;     // prints the initial value (0)
ctr.increaseBy(3);                  // increase by 3
cout << ctr.getCount() << endl;     // prints 3
ctr.increaseBy(5);                  // increase by 5
cout << ctr.getCount() << endl;     // prints 8
```

Access Control

One important feature of classes is the notion of *access control*. Members may be declared to be *public*, which means that they are accessible from outside the class, or *private*, which means that they are accessible only from within the class. (We discuss two exceptions to this later: protected access and friend functions.) In the previous example, we could not directly access the private member count from outside the class definition.

```
Counter ctr;                        // ctr is an instance of Counter
// ...
cout << ctr.count << endl;          // ILLEGAL - count is private
```

Why bother declaring members to be private? We discuss the reasons in detail in Chapter 2 when we discuss object-oriented programming. For now, suffice it to say that it stems from the desire to present users with a clean (public) interface from which to use the class, without bothering them with the internal (private) details of its implementation. All external access to class objects takes place through the public members, or the *public interface* as it is called. The syntax for a class is as follows.

```
class ⟨ class_name ⟩ {
public:
    public_members
private:
    private_members
};
```

Note that if no *access specifier* is given, the default is **private** for classes and **public** for structures. (There is a third specifier, called **protected**, which is discussed later in the book.) There is no required order between the private and public

sections, and in fact, it is possible to switch back and forth between them. Most C++ style manuals recommend that public members be presented first, since these are the elements of the class that are relevant to a programmer who wishes to use the class. We sometimes violate this convention in this book, particularly when we want to emphasize the private members.

Member Functions

Let us return to the Passenger structure, which was introduced earlier in Section 1.1.3, but this time we define it using a class structure. We provide the same member variables as earlier, but they are now private members. To this we add a few member functions. For this short example, we provide just a few of many possible member functions. The first member function is a constructor. Its job is to guarantee that each instance of the class is properly initialized. Notice that the constructor does not have a return type. The member function isFrequentFlyer tests whether the passenger is a frequent flyer, and the member function makeFrequent-Flyer makes a passenger a frequent flyer and assigns a frequent flyer number. This is only a partial definition, and a number of member functions have been omitted. As usual we use " //..." to indicate omitted code.

```cpp
class Passenger {                       // Passenger (as a class)
public:
  Passenger();                          // constructor
  bool isFrequentFlyer() const;         // is this a frequent flyer?
                                        // make this a frequent flyer
  void makeFrequentFlyer(const string& newFreqFlyerNo);
  // ... other member functions
private:
  string     name;                      // passenger name
  MealType   mealPref;                  // meal preference
  bool       isFreqFlyer;               // is a frequent flyer?
  string     freqFlyerNo;               // frequent flyer number
};
```

Class member functions can be placed in two major categories: ***accessor functions***, which only read class data, and ***update functions***, which may alter class data. The keyword "**const**" indicates that the member function isFrequentFlyer is an accessor. This informs the user of the class that this function will not change the object contents. It also allows the compiler to catch a potential error should we inadvertently attempt to modify any class member variables.

We have declared two member functions, but we still need to define them. Member functions may either be defined inside or outside the class body. Most C++ style manuals recommend defining all member functions outside the class, in

order to present a clean public interface in the class's definition. As we saw above in the Counter example, when a member function is defined outside the class body, it is necessary to specify which class it belongs to, which is done by preceding the function name with the scoping specifier class_name::member_name.

```
bool Passenger::isFrequentFlyer() const {
  return isFreqFlyer;
}
void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
  isFreqFlyer = true;
  freqFlyerNo = newFreqFlyerNo;
}
```

Notice that when we are within the body of a member function, the member variables (such as isFreqFlyer and freqFlyerNo) are given without reference to a particular object. These functions will be invoked on a particular Passenger object. For example, let pass be a variable of type Passenger. We may invoke these public member functions on pass using the same member selection operator we introduced with structures as shown below. Only public members may be accessed in this way.

```
Passenger pass;                        // pass is a Passenger
// ...
if ( !pass.isFrequentFlyer() ) {       // not already a frequent flyer?
  pass.makeFrequentFlyer("392953");    // set pass's freq flyer number
}
pass.name = "Joe Blow";                // ILLEGAL! name is private
```

In-Class Function Definitions

In the above examples, we have shown member functions being defined outside of the class body. We can also define members within the class body. When a member function is defined within a class it is compiled ***in line*** (recall Section 1.4.2). As with in-line functions, in-class function definitions should be reserved for short functions that do not involve loops or conditionals. Here is an example of how the isFrequentFlyer member function would be defined from within the class.

```
class Passenger {
public:
  // ...
  bool isFrequentFlyer() const { return isFreqFlyer; }
  // ...
};
```

1.5.2 Constructors and Destructors

The above declaration of the class variable pass suffers from the shortcoming that we have not initialized any of its classes members. An important aspect of classes is the capability to initialize a class's member data. A ***constructor*** is a special member function whose task is to perform such an initialization. It is invoked when a new class object comes into existence. There is an analogous ***destructor*** member function that is called when a class object goes out of existence.

Constructors

A constructor member function's name is the same as the class, and it has no return type. Because objects may be initialized in different ways, it is natural to define different constructors and rely on function overloading to determine which one is to be called.

Returning to our Passenger class, let us define three constructors. The first constructor has no arguments. Such a constructor is called a ***default constructor***, since it is used in the absence of any initialization information. The second constructor is given the values of the member variables to initialize. The third constructor is given a Passenger reference from which to copy information. This is called a ***copy constructor***.

```
class Passenger {
private:
  // ...
public:
  Passenger();                              // default constructor
  Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
  Passenger(const Passenger& pass);  // copy constructor
  // ...
};
```

Look carefully at the second constructor. The notation ffn="NONE" indicates that the argument for ffn is a ***default argument***. That is, an actual argument need not be given, and if so, the value "NONE" is used instead. If a newly created passenger is not a frequent flyer, we simply omit this argument. The constructor tests for this special value and sets things up accordingly. Default arguments can be assigned any legal value and can be used for more than one argument. It is often useful to define default values for all the arguments of a constructor. Such a constructor is the default constructor because it is called if no arguments are given. Default arguments can be used with any function (not just constructors). The associated constructor definitions are shown below. Note that the default argument is given in

the declaration, but not in the definition.

```
Passenger::Passenger() {                    // default constructor
  name = "--NO NAME--";
  mealPref = NO_PREF;
  isFreqFlyer = false;
  freqFlyerNo = "NONE";
}
                                            // constructor given member values
Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
  name = nm;
  mealPref = mp;
  isFreqFlyer = (ffn != "NONE");        // true only if ffn given
  freqFlyerNo = ffn;
}
                                            // copy constructor
Passenger::Passenger(const Passenger& pass) {
  name = pass.name;
  mealPref = pass.mealPref;
  isFreqFlyer = pass.isFreqFlyer;
  freqFlyerNo = pass.freqFlyerNo;
}
```

Here are some examples of how the constructors above can be invoked to define Passenger objects. Note that in the cases of p3 and pp2 we have omitted the frequent flyer number.

```
Passenger p1;                                   // default constructor
Passenger p2("John Smith", VEGETARIAN, 293145); // 2nd constructor
Passenger p3("Pocahontas", REGULAR);        // not a frequent flyer
Passenger p4(p3);                           // copied from p3
Passenger p5 = p2;                          // copied from p2
Passenger* pp1 = new Passenger;             // default constructor
Passenger* pp2 = new Passenger("Joe Blow", NO_PREF); // 2nd constr.
Passenger pa[20];                           // uses the default constructor
```

Although they look different, the declarations for p4 and p5 both call the copy constructor. These declarations take advantage of a bit of notational magic, which C++ provides to make copy constructors look more like the type definitions we have seen so far. The declarations for pp1 and pp2 create new passenger objects from the free store, and return a pointer to each. The declaration of pa declares an array of Passenger. The individual members of the array are always initialized from the default constructor.

Initializing Class Members with Initializer Lists

There is a subtlety that we glossed over in our presentations of the constructors. Recall that a string is a class in the standard template library. Our initialization using "name=nm" above relied on the fact that the string class has an assignment operator defined for it. If the type of name is a class without an assignment operator, this type of initialization might not be possible. In order to deal with the issue of initializing member variables that are themselves classes, C++ provides an alternate method of initialization called an ***initializer list***. This list is placed between the constructor's argument list and its body. It consists of a colon (:) followed by a comma-separated list of the form member_name(initial_value). To illustrate the feature, let us rewrite the second Passenger constructor so that its first three members are initialized by an initializer list. The initializer list is executed before the body of the constructor.

```
                    // constructor using an initializer list
    Passenger::Passenger(const string& nm, MealType mp, string ffn)
       : name(nm), mealPref(mp), isFreqFlyer(ffn != "NONE")
       { freqFlyerNo = ffn; }
```

Destructors

A constructor is called when a class object comes into existence. A ***destructor*** is a member function that is automatically called when a class object ceases to exist. If a class object comes into existence dynamically using the **new** operator, the destructor will be called when this object is destroyed using the **delete** operator. If a class object comes into existence because it is a local variable in a function that has been called, the destructor will be called when the function returns. The destructor for a class T is denoted ˜T. It takes no arguments and has no return type. Destructors are needed when classes allocate resources, such as memory, from the system. When the object ceases to exist, it is the responsibility of the destructor to return these resources to the system.

Let us consider a class Vect, shown in the following code fragment, which stores a vector by dynamically allocating an array of integers. The dynamic array is referenced by the member variable data. (Recall from Section 1.1.3 that a dynamically allocated array is represented by a pointer to its initial element.) The member variable size stores the number of elements in the vector. The constructor for this class allocates an array of the desired size. In order to return this space to the system when a Vect object is removed, we need to provide a destructor to deallocate this space. (Recall that when an array is deleted we use "delete[ ]," rather than "**delete**.")

```
class Vect {                          // a vector class
public:
  Vect(int n);                        // constructor, given size
  ~Vect();                            // destructor
  // ... other public members omitted
private:
  int*        data;                   // an array holding the vector
  int         size;                   // number of array entries
};

Vect::Vect(int n) {                   // constructor
  size = n;
  data = new int[n];                  // allocate array
}

Vect::~Vect() {                       // destructor
  delete [] data;                     // free the allocated array
}
```

We are not strictly required by C++ to provide our own destructor. Nonetheless, if our class allocates memory, we should write a destructor to free this memory. If we did not provide the destructor in the example above, the deletion of an object of type Vect would cause a memory leak. (Recall from Section 1.1.3 that this is an inaccessible block of memory that cannot be removed). The job of explicitly deallocating objects that were allocated is one of the chores that C++ programmers must endure.

1.5.3   Classes and Memory Allocation

When a class performs memory allocation using **new**, care must be taken to avoid a number of common programming errors. We have shown above that failure to deallocate storage in a class's destructor can result in memory leaks. A somewhat more insidious problem occurs when classes that allocate memory fail to provide a copy constructor or an assignment operator. Consider the following example, using our Vect class.

```
Vect a(100);                          // a is a vector of size 100
Vect b = a;                           // initialize b from a (DANGER!)
Vect c;                               // c is a vector (default size 10)
c = a;                                // assign a to c (DANGER!)
```

It would seem that we have just created three separate vectors, all of size 100, but have we? In reality all three of these vectors share the same 100-element array. Let us see why this has occurred.

The declaration of object a invokes the vector constructor, which allocates an array of 100 integers and a.data points to this array. The declaration "Vect b=a" initializes b from a. Since we provided no copy constructor in Vect, the system uses its default, which simply copies each member of a to b. In particular it sets "b.data=a.data." Notice that this does not copy the contents of the array; rather it copies the pointer to the array's initial element. This default action is sometimes called a *shallow copy*.

The declaration of c invokes the constructor with a default argument value of 10, and hence allocates an array of 10 elements in the free store. Because we have not provided an assignment operator, the statement "c=a," also does a shallow copy of a to c. Only pointers are copied, not array contents. Worse yet, we have lost the pointer to c's original 10-element array, thus creating a memory leak.

Now, a, b, and c all have members that point to the same array in the free store. If the contents of the arrays of one of the three were to change, the other two would mysteriously change as well. Worse yet, if one of the three were to be deleted before the others (for example, if this variable was declared in a nested block), the destructor would delete the shared array. When either of the other two attempts to access the now deleted array, the results would be disastrous. In short, there are many problems here.

Fortunately, there is a simple fix for all of these problems. The problems arose because we allocated memory and we used the system's default copy constructor and assignment operator. If a class allocates memory, you should provide a copy constructor and assignment operator to allocate new memory for making copies. A copy constructor for a class T is typically declared to take a single argument, which is a constant reference to an object of the same class, that is, T(const T& t). As shown in the code fragment below, it copies each of the data members from one class to the other while allocating memory for any dynamic members.

```
Vect::Vect(const Vect& a) {          // copy constructor from a
  size = a.size;                     // copy sizes
  data = new int[size];              // allocate new array
  for (int i = 0; i < size; i++) {   // copy the vector contents
    data[i] = a.data[i];
  }
}
```

The assignment operator is handled by overloading the = operator as shown in the next code fragment. The argument "a" plays the role of the object on the right side of the assignment operator. The assignment operator deletes the existing array storage, allocates a new array of the proper size, and copies elements into this new array. The if statement checks against the possibility of self assignment. (This can sometimes happen when different variables reference the same object.) We perform this check using the keyword **this**. For any instance of a class object,

"**this**" is defined to be the address of this instance. If **this** equals the address of a, then this is a case of self assignment, and we ignore the operation. Otherwise, we deallocate the existing array, allocate a new array, and copy the contents over.

```
Vect& Vect::operator=(const Vect& a) {   // assignment operator from a
  if (this != &a) {                       // avoid self-assignment
    delete [] data;                       // delete old array
    size = a.size;                         // set new size
    data = new int[size];                 // allocate new array
    for (int i=0; i < size; i++) {        // copy the vector contents
      data[i] = a.data[i];
    }
  }
  return *this;
}
```

Notice that in the last line of the assignment operator we return a reference to the current object with the statement "return *this." Such an approach is useful for assignment operators, since it allows us to chain together assignments, as in "a=b=c." The assignment "b=c" invokes the assignment operator, copying variable c to b and then returns a reference to b. This result is then assigned to variable a.

The only other changes needed to complete the job would be to add the appropriate function declarations to the Vect class. By using the copy constructor and assignment operator, we avoid the above memory leak and the dangerous shared array. The lessons of the last two sections can be summarized in the following rule.

Remember

> Every class that allocates its own objects using **new** should:
> - Define a ***destructor*** to free any allocated objects.
> - Define a ***copy constructor***, which allocates its own new member storage and copies the contents of member variables.
> - Define an ***assignment operator***, which deallocates old storage, allocates new storage, and copies all member variables.

Some programmers recommend that these functions be included for every class, even if memory is not allocated, but we are not so fastidious. In rare instances, we may want to forbid users from using one or more of these operations. For example, we may not want a huge data structure to be copied inadvertently. In this case, we can define empty copy constructors and assignment functions and make them private members of the class.

1.5.4 Class Friends and Class Members

Complex data structures typically involve the interaction of many different classes. In such cases, there are often issues coordinating the actions of these classes to allow sharing of information. We discuss some of these issues in this section.

We said private members of a class may only be accessed from within the class, but there is an exception to this. Specifically, we can declare a function as a *friend*, which means that this function may access the class's private data. There are a number of reasons for defining friend functions. One is that syntax requirements may forbid us from defining a member function. For example, consider a class SomeClass. Suppose that we want to define an overloaded output operator for this class, and this output operator needs access to private member data. To handle this, the class declares that the output operator is a friend of the class as shown below.

```
class SomeClass {
private:
  int secret;
public:
  // ...                              // give << operator access to secret
  friend ostream& operator<<(ostream& out, const SomeClass& x);
};

ostream& operator<<(ostream& out, const SomeClass& x)
  { cout << x.secret; }
```

Another time when it is appropriate to use friends is when two different classes are closely related. For example, Code Fragment 1.1 shows two cooperating classes Vector and Matrix. The former stores a three-dimensional vector and the latter stores a 3×3 matrix. In this code fragment, we show just one example of the usefulness of class friendship. The class Vector stores is coordinates in a private array, called coord. The Matrix class defines a function that multiplies a matrix times a vector. Because coord is a private member of Vector, members of the class Matrix would not have access to coord. However, because Vector has declared Matrix to be a friend, class Matrix can access all the private members of class Vector.

The ability to declare friendship relationships between classes is useful, but the extensive use of friends often indicates a poor class structure design. For example, a better solution would be to have class Vector define a public subscripting operator. Then the multiply function could use this public member to access the vector class, rather than access private member data.

Note that "friendship" is not transitive. For example, if a new class Tensor was made a friend of Matrix, Tensor would not be a friend of Vector, unless class Vector were to explicitly declare it to be so.

```
class Vector {                              // a 3-element vector
public: // ... public members omitted
private:
  double coord[3];                          // storage for coordinates
  friend class Matrix;                      // give Matrix access to coord
};

class Matrix {                              // a 3x3 matrix
public:
  Vector multiply(const Vector& v);         // multiply by vector v
  // ... other public members omitted
private:
  double a[3][3];                           // matrix entries
};

Vector Matrix::multiply(const Vector& v) {  // multiply by vector v
  Vector w;
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      w.coord[i] += a[i][j] * v.coord[j];   // access to coord allowed
  return w;
}
```

Code Fragment 1.1: An example of class friendship.

Nesting Classes and Types within Classes

We know that classes may define member variables and member functions. Classes may also define their own types as well. In particular, we can nest a class definition within another class. Such a ***nested class*** is often convenient in the design of data structures. For example, suppose that we want to design a data structure, called Book, and we want to provide a mechanism for placing bookmarks to identify particular locations within our book. We could define a nested class, called Bookmark, which is defined within class Book.

```
class Book {
public:
  class Bookmark {
    // ... (Bookmark definition here)
  };
  // ... (Remainder of Book definition)
}
```

We might define a member function that returns a bookmark within the book, say, to the start of some chapter. Outside the class Book, we use the scope-resolution operator, Book::Bookmark, in order to refer to this nested class. We shall see many other examples of nested classes later in the book.

1.5.5 The Standard Template Library

The ***Standard Template Library*** (***STL***) is a collection of useful classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following ***standard containers***. We discuss many of these data structures later in this book, so don't worry if their names seem unfamiliar.

stack	Container with last-in, first-out access
queue	Container with first-in, first-out access
deque	Double-ended queue
vector	Resizeable array
list	Doubly linked list
priority_queue	Queue ordered by value
set	Set
map	Associative array (dictionary)

Templates and the STL Vector Class

One of the important features of the STL is that each such object can store objects of any one type. Contrast this with the Vect class of Section 1.5.2, which can only hold integers. Such a class whose definition depends on a user-specified type is called a ***template***. We discuss templates in greater detail in Chapter 2, but we briefly mention how they are used with container objects here.

We specify the type of the object being stored in the container in angle brackets (<...>). For example, we could define vectors to hold 100 integers, 500 characters, and 20 passengers as follows:

```cpp
#include <vector>
using namespace std;                    // make std accessible

vector<int> scores(100);                // 100 integer scores
vector<char> buffer(500);               // buffer of 500 characters
vector<Passenger> passenList(20);       // list of 20 Passengers
```

As usual, the include statement provides the necessary declarations for using the vector class. Each instance of an STL vector can only hold objects of one type.

STL vectors are superior to standard C++ arrays in many respects. First, as with arrays, individual elements can be indexed using the usual index operator ([ ]). They can also be accessed by the at member function. The advantage of the latter is that it performs range checking and generates an error exception if the index is out of bounds. (We discuss exceptions in Section 2.4.) Recall that standard arrays in C++ do not even know their size, and hence range checking is not even possible. In contrast, a vector object's size is given by its size member function. Unlike

standard arrays, one vector object can be assigned to another, which results in the contents of one vector object being copied to the other. A vector can be resized dynamically by calling the resize member function. We show several examples of uses of the STL vector class below.

```
int i = // ...
cout << scores[i];                  // index (range unchecked)
buffer.at(i) = buffer.at(2 * i);    // index (range checked)
vector<int> newScores = scores;     // copy scores to newScores
scores.resize(scores.size() + 10);  // add room for 10 more elements
```

We discuss the STL further in Chapter 3.

More on STL Strings

In Section 1.1.3, we introduced the STL string class. This class provides a number of useful utilities for manipulating character strings. Earlier, we discussed the use of the addition operator (" +") for concatenating strings, the operator " +=" for appending a string to the end of an existing string, the function size for determining the length of a string, and the indexing operator ("[]") for accessing individual characters of a string.

Let us present a few more string functions. In the table below, let s be an STL string, and let p be either an STL string or a standard C++ string. Let i and m be nonnegative integers. Throughout, we use i to denote the index of a position in a string and we use m to denote the number of characters involved in the operation. (A string's first character is at index $i = 0$.)

s.find(p)	Return the index of first occurrence of string p in s
s.find(p, i)	Return the index of first occurrence of string p in s on or after position i
s.substr(i,m)	Return the substring starting at position i of s and consisting of m characters
s.insert(i, p)	Insert string p just prior to index i in s
s.erase(i, m)	Remove the substring of length m starting at index i
s.replace(i, m, p)	Replace the substring of length m starting at index i with p
getline(is, s)	Read a single line from the input stream is and store the result in s

In order to indicate that a pattern string p is not found, the find function returns the special value string::npos. Strings can also be compared lexicographically, using the C++ comparison operators: $<$, $<=$, $>$, $>=$, $==$, and $!=$.

Here are some examples of the use of these functions.

```cpp
string s = "a dog";                       // "a dog"
s += " is a dog";                         // "a dog is a dog"
cout << s.find("dog");                    // 2
cout << s.find("dog", 3);                 // 11
if (s.find("doug") == string::npos) { }   // true
cout << s.substr(7, 5);                   // "s a d"
s.replace(2, 3, "frog");                  // "a frog is a dog"
s.erase(6, 3);                            // "a frog a dog"
s.insert(0, "is ");                       // "is a frog a dog"
if (s == "is a frog a dog") { }           // true
if (s < "is a frog a toad") { }           // true
if (s < "is a frog a cat") { }            // false
```

1.6 C++ **Program and File Organization**

Let us now consider the broader issue of how to organize an entire C++ program. A typical large C++ program consists of many files, with related pieces of code residing within each file. For example, C++ programmers commonly place each major class in its own file.

Source Files

There are two common file types, source files and header files. *Source files* typically contain most of the executable statements and data definitions. This includes the bodies of functions and definitions of any global variables.

Different compilers use different file naming conventions. Source file names typically have distinctive suffixes, such as ".cc", ".cpp", and ".C". Source files may be compiled separately by the compiler, and then these files are combined into one program by a system program called a *linker*.

Each nonconstant global variable and function may be defined only once. Other source files may share such a global variable or function provided they have a matching declaration. To indicate that a global variable is defined in another file, the type specifier "**extern**" is added. This keyword is not needed for functions. For example, consider the declarations extracted from two files below. The file Source1.cpp defines a global variable cat and function foo. The file Source2.cpp can access these objects by including the appropriate matching declarations and adding "**extern**" for variables.

File: Source1.cpp

```cpp
int cat = 1;                   // definition of cat
int foo(int x) { return x+1; } // definition of foo
```

File: Source2.cpp

```
extern int cat;                    // cat is defined elsewhere
int foo(int x);                    // foo is defined elsewhere
```

Header Files

Since source files using shared objects must provide identical declarations, we commonly store these shared declarations in a ***header file***, which is then read into each such source file using an #include statement. Statements beginning with # are handled by a special program, called the ***preprocessor***, which is invoked automatically by the compiler. A header file typically contains many declarations, including classes, structures, constants, enumerations, and typedefs. Header files generally do not contain the definition (body) of a function. In-line functions are an exception, however, as their bodies are given in a header file.

Except for some standard library headers, the convention is that header file names end with a ".h" suffix. Standard library header files are indicated with angle brackets, as in <iostream>, while other local header files are indicated using quotes, as in "myIncludes.h".

```
#include <iostream>               // system include file
#include "myIncludes.h"           // user-defined include file
```

As a general rule, we should avoid including namespace **using** directives in header files, because any source file that includes such a header file has its namespace expanded as a result. We make one exception to this in our examples, however. Some of our header files include a **using** directive for the STL string class because it is so useful.

1.6.1   An Example Program

To make this description more concrete, let us consider an example of a simple yet complete C++ program. Our example consists of one class, called CreditCard, which defines a credit card object and a procedure that uses this class.

The CreditCard Class

The credit card object defined by CreditCard is a simplified version of a traditional credit card. It has an identifying number, identifying information about the owner, and information about the credit limit and the current balance. It does not charge interest or late payments, but it does restrict charges that would cause a card's balance to go over its spending limit.

The main class structure is presented in the header file CreditCard.h and is shown in Code Fragment 1.2.

```cpp
#ifndef CREDIT_CARD_H                    // avoid repeated expansion
#define CREDIT_CARD_H

#include <string>                        // provides string
#include <iostream>                      // provides ostream

class CreditCard {
public:
  CreditCard(const std::string& no,      // constructor
      const std::string& nm, int lim, double bal=0);
                                         // accessor functions
  std::string    getNumber() const    { return number; }
  std::string    getName() const      { return name; }
  double         getBalance() const   { return balance; }
  int            getLimit() const     { return limit; }

  bool chargeIt(double price);           // make a charge
  void makePayment(double payment);      // make a payment
private:                                 // private member data
  std::string    number;                 // credit card number
  std::string    name;                   // card owner's name
  int            limit;                  // credit limit
  double         balance;                // credit card balance
};
                                         // print card information
std::ostream& operator<<(std::ostream& out, const CreditCard& c);
#endif
```

Code Fragment 1.2: The header file CreditCard.h, which contains the definition of class CreditCard.

Before discussing the class, let us say a bit about the general file structure. The first two lines (containing #ifndef and #define) and the last line (containing #endif) are used to keep the same header file from being expanded twice. We discuss this later. The next lines include the header files for strings and standard input and output.

This class has four private data members. We provide a simple constructor to initialize these members. There are four *accessor functions*, which provide access to read the current values of these member variables. Of course, we could have alternately defined the member variables as being public and saved the work of providing these accessor functions. However, this would allow users to modify any of these member variables directly. We usually prefer to restrict the modification of member variables to special *update functions*. We include two such update functions, chargeIt and makePayment. We have also defined a stream output operator for the class.

The accessor functions and makePayment are short, so we define them within the class body. The other member functions and the output operator are defined outside the class in the file CreditCard.cpp, shown in Code Fragment 1.3. This approach of defining a header file with the class definition and an associated source file with the longer member function definitions is common in C++.

The Main Test Program

Our main program is in the file TestCard.cpp. It consists of a main function, but this function does little more than call the function testCard, which does all the work. We include CreditCard.h to provide the CreditCard declaration. We do not need to include iostream and string, since CreditCard.h does this for us, but it would not have hurt to do so.

The testCard function declares an array of pointers to CreditCard. We allocate three such objects and initialize them. We then perform a number of payments and print the associated information. We show the complete code for the Test class in Code Fragment 1.4.

The output of the Test class is sent to the standard output stream. We show this output in Code Fragment 1.5.

Avoiding Multiple Header Expansions

A typical C++ program includes many different header files, which often include other header files. As a result, the same header file may be expanded many times. Such repeated header expansion is wasteful and can result in compilation errors because of repeated definitions. To avoid this repeated expansion, most header files use a combination of preprocessor commands. Let us explain the process, illustrated in Code Fragment 1.2.

```cpp
#include "CreditCard.h"                         // provides CreditCard

using namespace std;                            // make std:: accessible
                                                // standard constructor
CreditCard::CreditCard(const string& no, const string& nm, int lim, double bal) {
  number = no;
  name = nm;
  balance = bal;
  limit = lim;
}
                                                // make a charge
bool CreditCard::chargeIt(double price) {
  if (price + balance > double(limit))
    return false;                               // over limit
  balance += price;
  return true;                                  // the charge goes through
}

void CreditCard::makePayment(double payment) { // make a payment
  balance -= payment;
}
                                                // print card information
ostream& operator<<(ostream& out, const CreditCard& c) {
  out << "Number = "    << c.getNumber()  << "\n"
      << "Name = "      << c.getName()    << "\n"
      << "Balance = "   << c.getBalance() << "\n"
      << "Limit = "     << c.getLimit()   << "\n";
  return out;
}
```

Code Fragment 1.3: The file CreditCard.cpp, which contains the definition of the out-of-class member functions for class CreditCard.

```cpp
#include <vector>                          // provides STL vector
#include "CreditCard.h"                    // provides CreditCard, cout, string

using namespace std;                       // make std accessible

void testCard() {                          // CreditCard test function
  vector<CreditCard*> wallet(10);          // vector of 10 CreditCard pointers
                                           // allocate 3 new cards
  wallet[0] = new CreditCard("5391 0375 9387 5309", "John Bowman", 2500);
  wallet[1] = new CreditCard("3485 0399 3395 1954", "John Bowman", 3500);
  wallet[2] = new CreditCard("6011 4902 3294 2994", "John Bowman", 5000);

  for (int j=1; j <= 16; j++) {            // make some charges
    wallet[0]->chargeIt(double(j));        // explicitly cast to double
    wallet[1]->chargeIt(2 * j);            // implicitly cast to double
    wallet[2]->chargeIt(double(3 * j));
  }

  cout << "Card payments:\n";
  for (int i=0; i < 3; i++) {              // make more charges
    cout << *wallet[i];
    while (wallet[i]->getBalance() > 100.0) {
      wallet[i]->makePayment(100.0);
      cout << "New balance = " << wallet[i]->getBalance() << "\n";
    }
    cout << "\n";
    delete wallet[i];                      // deallocate storage
  }
}

int main() {                               // main function
  testCard();
  return EXIT_SUCCESS;                      // successful execution
}
```

Code Fragment 1.4: The file TestCard.cpp.

Let us start with the second line. The #define statement defines a preprocessor variable CREDIT_CARD_H. This variable's name is typically based on the header file name, and by convention, it is written in all capitals. The name itself is not important as long as different header files use different names. The entire file is enclosed in a preprocessor "if" block starting with #ifndef on top and ending with #endif at the bottom. The "ifndef" is read "if not defined," meaning that the header file contents will be expanded only if the preprocessor variable CREDIT_CARD_H is *not* defined.

Here is how it works. The first time the header file is encountered, the variable

```
Card payments:
Number = 5391 0375 9387 5309
Name = John Bowman
Balance = 136
Limit = 2500
New balance = 36

Number = 3485 0399 3395 1954
Name = John Bowman
Balance = 272
Limit = 3500
New balance = 172
New balance = 72

Number = 6011 4902 3294 2994
Name = John Bowman
Balance = 408
Limit = 5000
New balance = 308
New balance = 208
New balance = 108
New balance = 8
```

Code Fragment 1.5: Sample program output.

CREDIT_CARD_H has not yet been seen, so the header file is expanded by the preprocessor. In the process of doing this expansion, the second line defines the variable CREDIT_CARD_H. Hence, any attempt to include the header file will find that CREDIT_CARD_H is defined, so the file will not be expanded.

Throughout this book we omit these preprocessor commands from our examples, but they should be included in each header file we write.

1.7 Writing a C++ Program

As with any programming language, writing a program in C++ involves three fundamental steps:

1. Design
2. Coding
3. Testing and Debugging.

We briefly discuss each of these steps in this section.

1.7.1   Design

The design step is perhaps the most important in the process of writing a program. In this step, we decide how to divide the workings of our program into classes, we decide how these classes will interact, what data each will store, and what actions each will perform. Indeed, one of the main challenges that beginning C++ programmers face is deciding what classes to define to do the work of their program. While general prescriptions are hard to come by, there are some general rules of thumb that we can apply when determining how to define our classes.

- *Responsibilities*: Divide the work into different *actors*, each with a different responsibility. Try to describe responsibilities using action verbs. These actors form the classes for the program.
- *Independence*: Define the work for each class to be as independent from other classes as possible. Subdivide responsibilities between classes so that each class has autonomy over some aspect of the program. Give data (as member variables) to the class that has jurisdiction over the actions that require access to this data.
- *Behaviors*: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class are well understood by other classes with which it interacts. These behaviors define the member functions that this class performs. The set of behaviors for a class is sometimes referred to as a *protocol*, since we expect the behaviors for a class to hold together as a cohesive unit.

Defining the classes, together with their member variables and member functions, determines the design of a C++ program. A good programmer will naturally develop greater skill in performing these tasks over time, as experience teaches him or her to notice patterns in the requirements of a program that match patterns that he or she has seen before.

1.7.2   Pseudo-Code

Programmers are often asked to describe algorithms in a way that is intended for human eyes only, prior to writing actual code. Such descriptions are called *pseudo-code*. Pseudo-code is not a computer program, but is more structured than usual prose. Pseudo-code is a mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm. There really is no precise definition of the *pseudo-code* language, however, because of its reliance on natural language. At the same time, to help achieve clarity, pseudo-code mixes natural language with standard programming language constructs. The programming language constructs we choose are those consistent with modern high-level languages such as C, C++, and Java.

These constructs include the following:

- *Expressions:* We use standard mathematical symbols to express numeric and Boolean expressions. We use the left arrow sign ($\leftarrow$) as the assignment operator in assignment statements (equivalent to the $=$ operator in C++) and we use the equal sign ($=$) as the equality relation in Boolean expressions (equivalent to the "$==$" relation in C++).
- *Function declarations:* **Algorithm** *name(arg1,arg2, ...)* declares a new function "name" and its arguments.
- *Decision structures:* **if** *condition* **then** *true-actions* [**else** *false-actions*]. We use indentation to indicate what actions should be included in the true-actions and false-actions.
- *While-loops:* **while** *condition* **do** *actions*. We use indentation to indicate what actions should be included in the loop actions.
- *Repeat-loops:* **repeat** *actions* **until** *condition*. We use indentation to indicate what actions should be included in the loop actions.
- *For-loops:* **for** *variable-increment-definition* **do** *actions*. We use indentation to indicate what actions should be included among the loop actions.
- *Array indexing:* $A[i]$ represents the ith cell in the array A. The cells of an n-celled array A are indexed from $A[0]$ to $A[n-1]$ (consistent with C++).
- *Member function calls:* *object.method(args)* (object is optional if it is understood).
- *Function returns:* **return** *value*. This operation returns the value specified to the method that called this one.
- *Comments:* { *Comment goes here.* }. We enclose comments in braces.

When we write pseudo-code, we must keep in mind that we are writing for a human reader, not a computer. Thus, we should strive to communicate high-level ideas, not low-level implementation details. At the same time, we should not gloss over important steps. Like many forms of human communication, finding the right balance is an important skill that is refined through practice.

1.7.3   Coding

As mentioned above, one of the key steps in coding up an object-oriented program is coding up the descriptions of classes and their respective data and member functions. In order to accelerate the development of this skill, we discuss various *design patterns* for designing object-oriented programs (see Section 2.1.3) at various points throughout this text. These patterns provide templates for defining classes and the interactions between these classes.

Many programmers do their initial coding not on a computer, but by using **CRC cards**. Class-Responsibility-Collaborator (CRC) cards are simple index cards that subdivide the work required of a program. The main idea behind this tool is to

have each card represent a component, which will ultimately become a class in our program. We write the name of each component on the top of an index card. On the left-hand side of the card, we begin writing the responsibilities for this component. On the right-hand side, we list the collaborators for this component, that is, the other components that this component will have to interact with to perform its duties. The design process iterates through an action/actor cycle, where we first identify an action (that is, a responsibility), and we then determine an actor (that is, a component) that is best suited to perform that action. The design is complete when we have assigned all actions to actors.

By the way, in using index cards to begin our coding, we are assuming that each component will have a small set of responsibilities and collaborators. This assumption is no accident, since it helps keep our programs manageable.

An alternative to CRC cards is the use of UML (Unified Modeling Language) diagrams to express the organization of a program, and the use of pseudo-code to describe the algorithms. UML diagrams are a standard visual notation to express object-oriented software designs. Several computer-aided tools are available to build UML diagrams. Describing algorithms in pseudo-code, on the other hand, is a technique that we utilize throughout this book.

Once we have decided on the classes and their respective responsibilities for our programs, we are ready to begin coding. We create the actual code for the classes in our program by using either an independent text editor (such as emacs, notepad, or vi), or the editor embedded in an *integrated development environment* (IDE), such as Microsoft's Visual Studio and Eclipse.

Once we have completed coding for a program (or file), we then compile this file into working code by invoking a compiler. If our program contains syntax errors, they will be identified, and we will have to go back into our editor to fix the offending lines of code. Once we have eliminated all syntax errors and created the appropriate compiled code, we then run our program.

Readability and Style

Programs should be made easy to read and understand. Good programmers should therefore be mindful of their coding style and develop a style that communicates the important aspects of a program's design for both humans and computers. Much has been written about good coding style. Here are some of the main principles.

- Use meaningful names for identifiers. Try to choose names that can be read aloud and reflect the action, responsibility, or data each identifier is naming. The tradition in most C++ circles is to capitalize the first letter of each word in an identifier, except for the first word in an identifier for a variable or method. So, in this tradition, "Date," "Vector," and "DeviceManager"

would identify classes, and "isFull," "insertItem," "studentName," and "studentHeight" would respectively identify member functions and variables.

- Use named constants and enumerations instead of embedded values. Readability, robustness, and modifiability are enhanced if we include a series of definitions of named constant values in a class definition. These can then be used within this class and others to refer to special values for this class. Our convention is to fully capitalize such constants as shown below.

```cpp
const int MIN_CREDITS = 12;      // min. credits in a term
const int MAX_CREDITS = 24;      // max. credits in a term
                                 // enumeration for year
enum Year { FRESHMAN, SOPHOMORE, JUNIOR, SENIOR };
```

- Indent statement blocks. Typically programmers indent each statement block by four spaces. (In this book, we typically use two spaces to avoid having our code overrun the book's margins.)
- Organize each class in a consistent order. In the examples in this book, we usually use the following order:
  1. Public types and nested classes
  2. Public member functions
  3. Protected member functions (internal utilities)
  4. Private member data

  Our class organizations do not always follow this convention. In particular, when we wish to emphasize the implementation details of a class, we present the private members first and the public functions afterwards.

- Use comments that add meaning to a program and explain ambiguous or confusing constructs. In-line comments are good for quick explanations and do not need to be sentences. Block comments are good for explaining the purpose of a method and complex code sections.

1.7.4 Testing and Debugging

Testing is the process of verifying the correctness of a program, while debugging is the process of tracking the execution of a program and discovering the errors in it. Testing and debugging are often the most time-consuming activity in the development of a program.

Testing

A careful testing plan is an essential part of writing a program. While verifying the correctness of a program over all possible inputs is usually not feasible, we should aim at executing the program on a representative subset of inputs. At the very minimum, we should make sure that every method in the program is tested

at least once (method coverage). Even better, each code statement in the program should be executed at least once (statement coverage).

Programs often tend to fail on **special cases** of the input. Such cases need to be carefully identified and tested. For example, when testing a method that sorts an array of integers (that is, arranges them in ascending order), we should consider the following inputs:

- The array has zero length (no elements)
- The array has one element
- All the elements of the array are the same
- The array is already sorted
- The array is reverse sorted

In addition to special inputs to the program, we should also consider special conditions for the structures used by the program. For example, if we use an array to store data, we should make sure that boundary cases, such as inserting/removing at the beginning or end of the subarray holding data, are properly handled. While it is essential to use hand-crafted test suites, it is also advantageous to run the program on a large collection of randomly generated inputs.

There is a hierarchy among the classes and functions of a program induced by the "caller-callee" relationship. Namely, a function A is above a function B in the hierarchy if A calls B. There are two main testing strategies, **top-down** and **bottom-up**, which differ in the order in which functions are tested.

Bottom-up testing proceeds from lower-level functions to higher-level functions. Namely, bottom-level functions, which do not invoke other functions, are tested first, followed by functions that call only bottom-level functions, and so on. This strategy ensures that errors found in a method are not likely to be caused by lower-level functions nested within it.

Top-down testing proceeds from the top to the bottom of the method hierarchy. It is typically used in conjunction with **stubbing**, a boot-strapping technique that replaces a lower-level method with a **stub**, a replacement for the method that simulates the output of the original method. For example, if function A calls function B to get the first line of a file, we can replace B with a stub that returns a fixed string when testing A.

Debugging

The simplest debugging technique consists of using **print statements** (typically using the stream output operator, "<<") to track the values of variables during the execution of the program. The problem with this approach is that the print statements need to be removed or commented out before the program can be executed as part of a "production" software system.

A better approach is to run the program within a ***debugger***, which is a specialized environment for controlling and monitoring the execution of a program. The basic functionality provided by a debugger is the insertion of ***breakpoints*** within the code. When the program is executed within the debugger, it stops at each breakpoint. While the program is stopped, the current value of variables can be inspected. In addition to fixed breakpoints, advanced debuggers allow for specification of ***conditional breakpoints***, which are triggered only if a given expression is satisfied.

Many IDEs, such as Microsoft Visual Studio and Eclipse provide built-in debuggers.

1.8   Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-1.1 Which of the following is **not** a valid C++ variable name? (There may be
more than one.)

    a. i_think_i_am_valid
    b. i_may_have_2_many_digits_2_be_valid
    c. _I_start_and_end_with_underscores_
    d. I_Have_A_Dollar_$ign
    e. I_AM_LONG_AND_HAVE_NO_LOWER_CASE_LETTERS

R-1.2 Write a pseudo-code description of a method for finding the smallest and
largest numbers in an array of integers and compare that to a C++ function
that would do the same thing.

R-1.3 Give a C++ definition of a **struct** called Pair that consists of two mem-
bers. The first is an integer called first, and the second is a double called
second.

R-1.4 What are the contents of string s after executing the following statements.

```
string s = "abc";
string t = "cde";
s += s + t[1] + s;
```

R-1.5 Consider the expression y + 2 * z ++ < 3 - w / 5. Add parentheses
to show the precise order of evaluation given the C++ rules for operator
precedence.

R-1.6 Consider the following attempt to allocate a 10-element array of pointers
to doubles and initialize the associated double values to 0.0. Rewrite the
following (**incorrect**) code to do this correctly. (Hint: Storage for the
doubles needs to be allocated.)

```
double* dp[10]
for (int i = 0; i < 10; i++) dp[i] = 0.0;
```

R-1.7 Write a short C++ function that takes an integer n and returns the sum of
all the integers smaller than n.

R-1.8 Write a short C++ function, isMultiple, that takes two positive **long** values,
n and m, and returns true if and only if n is a multiple of m, that is, $n = mi$
for some integer i.

R-1.9 Write a C++ function printArray(A, m, n) that prints an $m \times n$ two-dimensional array A of integers, declared to be "int** A," to the standard output. Each of the m rows should appear on a separate line.

R-1.10 What (if anything) is different about the behavior of the following two functions f and g that increment a variable and print its value?

```
void f(int x)
  { std::cout << ++x; }
void g(int& x)
  { std::cout << ++x; }
```

R-1.11 Write a C++ class, Flower, that has three member variables of type **string**, **int**, and **float**, which respectively represent the name of the flower, its number of pedals, and price. Your class must include a constructor method that initializes each variable to an appropriate value, and your class should include functions for setting the value of each type, and getting the value of each type.

R-1.12 Modify the CreditCard class from Code Fragment 1.3 to check that the price argument passed to function chargeIt and the payment argument passed to function makePayment are positive.

R-1.13 Modify the CreditCard class from Code Fragment 1.2 to charge interest on each payment.

R-1.14 Modify the CreditCard class from Code Fragment 1.2 to charge a late fee for any payment that is past its due date.

R-1.15 Modify the CreditCard class from Code Fragment 1.2 to include *modifier functions* that allow a user to modify internal variables in a CreditCard class in a controlled manner.

R-1.16 Modify the declaration of the first **for** loop in the Test class in Code Fragment 1.4 so that its charges will eventually cause exactly one of the three credit cards to go over its credit limit. Which credit card is it?

R-1.17 Write a C++ class, AllKinds, that has three member variables of type **int**, **long**, and **float**, respectively. Each class must include a constructor function that initializes each variable to a nonzero value, and each class should include functions for setting the value of each type, getting the value of each type, and computing and returning the sum of each possible combination of types.

R-1.18 Write a short C++ function, isMultiple, that takes two **long** values, n and m, and returns **true** if and only if n is a multiple of m, that is, $n = m \cdot i$ for some integer i.

R-1.19 Write a short C++ function, isTwoPower, that takes an **int** i and returns **true** if and only if i is a power of 2. Do not use multiplication or division, however.

R-1.20 Write a short C++ function that takes an integer n and returns the sum of all the integers smaller than n.

R-1.21 Write a short C++ function that takes an integer n and returns the sum of all the odd integers smaller than n.

R-1.22 Write a short C++ function that takes a positive **double** value x and returns the number of times we can divide x by 2 before we get a number less than 2.

Creativity

C-1.1 Write a pseudo-code description of a method that reverses an array of n integers, so that the numbers are listed in the opposite order than they were before, and compare this method to an equivalent C++ method for doing the same thing.

C-1.2 Write a short C++ function that takes an array of **int** values and determines if there is a pair of numbers in the array whose product is even.

C-1.3 Write a C++ function that takes an STL vector of **int** values and determines if all the numbers are different from each other (that is, they are distinct).

C-1.4 Write a C++ function that takes an STL vector of **int** values and prints all the odd values in the vector.

C-1.5 Write a C++ function that takes an array containing the set of all integers in the range 1 to 52 and shuffles it into random order. Use the built-in function rand, which returns a pseudo-random integer each time it is called. Your function should output each possible order with equal probability.

C-1.6 Write a short C++ program that outputs all possible strings formed by using each of the characters 'a', 'b', 'c', 'd', 'e', and 'f' exactly once.

C-1.7 Write a short C++ program that takes all the lines input to standard input and writes them to standard output in reverse order. That is, each line is output in the correct order, but the ordering of the lines is reversed.

C-1.8 Write a short C++ program that takes two arguments of type STL vector<double>, a and b, and returns the element-by-element product of a and b. That is, it returns a vector c of the same length such that $c[i] = a[i] \cdot b[i]$.

C-1.9 Write a C++ class Vector2, that stores the (x, y) coordinates of a two-dimensional vector, where x and y are of type **double**. Show how to override various C++ operators in order to implement the addition of two vectors (producing a vector result), the multiplication of a scalar times a vector (producing a vector result), and the dot product of two vectors (producing a double result).

C-1.10 Write an efficient C++ function that takes any integer value i and returns 2^i, as a **long** value. Your function should **not** multiply 2 by itself i times; there are much faster ways of computing 2^i.

C-1.11 The **greatest common divisor**, or GCD, of two positive integers n and m is the largest number j, such that n and m are both multiples of j. Euclid proposed a simple algorithm for computing $\text{GCD}(n, m)$, where $n > m$, which is based on a concept known as the Chinese Remainder Theorem. The main idea of the algorithm is to repeatedly perform modulo computations of consecutive pairs of the sequence that starts $(n, m, \ldots)$, until reaching zero. The last nonzero number in this sequence is the GCD of n and m. For example, for $n = 80,844$ and $m = 25,320$, the sequence is as follows:

$$
\begin{aligned}
80,844 \bmod 25,320 &= 4,884 \\
25,320 \bmod 4,884 &= 900 \\
4,884 \bmod 900 &= 384 \\
900 \bmod 384 &= 132 \\
384 \bmod 132 &= 120 \\
132 \bmod 120 &= 12 \\
120 \bmod 12 &= 0
\end{aligned}
$$

So, GCD of $80,844$ and $25,320$ is 12. Write a short C++ function to compute $\text{GCD}(n, m)$ for two integers n and m.

Projects

P-1.1 A common punishment for school children is to write out the same sentence multiple times. Write a C++ stand-alone program that will write out the following sentence one hundred times: "I will always use object-oriented design." Your program should number each of the sentences and it should "accidentally" make eight different random-looking typos at various points in the listing, so that it looks like a human typed it all by hand.

P-1.2 Write a C++ program that, when given a starting day (Sunday through Saturday) as a string, and a four-digit year, prints a calendar for that year. Each month should contain the name of the month, centered over the dates for that month and a line containing the names of the days of the week, running from Sunday to Saturday. Each week should be printed on a separate line. Be careful to check for a leap year.

P-1.3 The **birthday paradox** says that the probability that two people in a room will have the same birthday is more than half as long as the number of

people in the room (n), is more than 23. This property is not really a para-
dox, but many people find it surprising. Design a C++ program that can
test this paradox by a series of experiments on randomly generated birth-
days, which test this paradox for $n = 5, 10, 15, 20, \ldots, 100$. You should run
at least 10 experiments for each value of n and it should output, for each
n, the number of experiments for that n, such that two people in that test
have the same birthday.

Chapter Notes

For more detailed information about the C++ programming language and the Standard
Template Library, we refer the reader to books by Stroustrup [91], Lippmann and La-
joie [67], Musser and Saini [81], and Horstmann [47]. Lippmann also wrote a short in-
troduction to C++ [66]. For more advanced information of how to use C++'s features in
the most effective manner, consult the books by Meyers [77, 76]. For an introduction to
C++ assuming a background of C see the book by Pohl [84]. For an explanation of the
differences between C++ and Java see the book by Budd [17].

Chapter
2

Object-Oriented Design
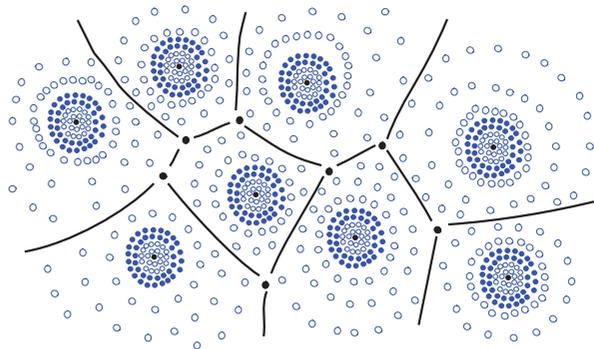
Contents

2.1   Goals, Principles, and Patterns

As the name implies, the main "actors" in the object-oriented design paradigm are called *objects*. An object comes from a *class*, which is a specification of the data *members* that the object contains, as well as the *member functions* (also called *methods* or operations) that the object can execute. Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects. This view of computing is intended to fulfill several goals and incorporate several design principles, which we discuss in this chapter.

2.1.1   Object-Oriented Design Goals

Software implementations should achieve *robustness*, *adaptability*, and *reusability*. (See Figure 2.1.)

| Robustness | Adaptability | Reusability |
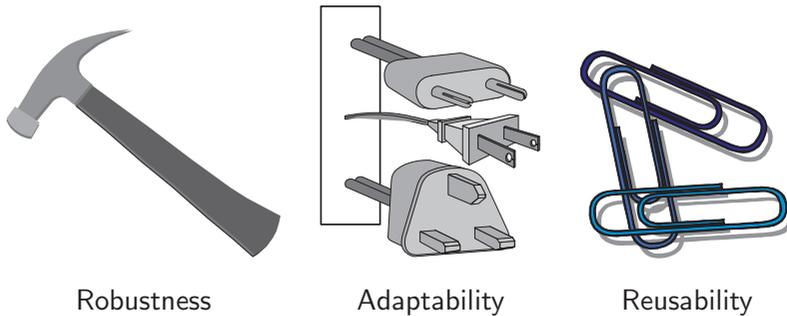
Figure 2.1: Goals of object-oriented design.

Robustness

Every good programmer wants to develop software that is correct, which means that a program produces the right output for all the anticipated inputs in the program's application. In addition, we want software to be *robust*, that is, capable of handling unexpected inputs that are not explicitly defined for its application. For example, if a program is expecting a positive integer (for example, representing the price of an item) and instead is given a negative integer, then the program should be able to recover gracefully from this error. More importantly, in *life-critical applications*, where a software error can lead to injury or loss of life, software that is not robust could be deadly. This point was driven home in the late 1980s in accidents involving Therac-25, a radiation-therapy machine, which severely overdosed six patients between 1985 and 1987, some of whom died from complications resulting from their radiation overdose. All six accidents were traced to software errors.

Adaptability

Modern software applications, such as Web browsers and Internet search engines, typically involve large programs that are used for many years. Software therefore needs to be able to evolve over time in response to changing conditions in its environment. Thus, another important goal of quality software is that it achieves ***adaptability*** (also called ***evolvability***). Related to this concept is ***portability***, which is the ability of software to run with minimal change on different hardware and operating system platforms. An advantage of writing software in C++ is the portability provided by the language itself.

Reusability

Going hand in hand with adaptability is the desire that software be reusable, that is, the same code should be usable as a component of different systems in various applications. Developing quality software can be an expensive enterprise, and its cost can be offset somewhat if the software is designed in a way that makes it easily reusable in future applications. Such reuse should be done with care, however, for one of the major sources of software errors in the Therac-25 came from inappropriate reuse of Therac-20 software (which was not object-oriented and not designed for the hardware platform used with the Therac-25).

2.1.2   Object-Oriented Design Principles

Chief among the principles of the object-oriented approach, which are intended to facilitate the goals outlined above, are the following (see Figure 2.2):

- Abstraction
- Encapsulation
- Modularity.
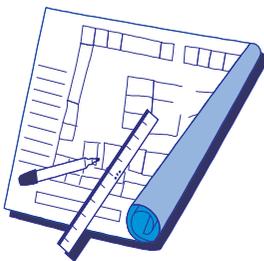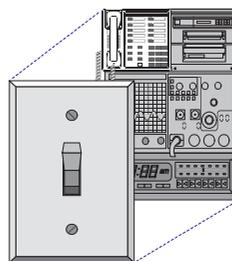
| Abstraction | Encapsulation | Modularity |

Figure 2.2: Principles of object-oriented design.

Abstraction

The notion of *abstraction* is to distill a complicated system down to its most fundamental parts and describe these parts in a simple, precise language. Typically, describing the parts of a system involves naming them and explaining their functionality. Applying the abstraction paradigm to the design of data structures gives rise to *abstract data types* (ADTs). An ADT is a mathematical model of a data structure that specifies the type of the data stored, the operations supported on them, and the types of the parameters of the operations. An ADT specifies *what* each operation does, but not *how* it does it. In C++, the functionality of a data structure is expressed through the public interface of the associated class or classes that define the data structure. By *public interface*, we mean the signatures (names, return types, and argument types) of a class's public member functions. This is the only part of the class that can be accessed by a user of the class.

An ADT is realized by a concrete data structure, which is modeled in C++ by a *class*. A class defines the data being stored and the operations supported by the objects that are instances of the class. Also, unlike interfaces, classes specify *how* the operations are performed in the body of each function. A C++ class is said to *implement an interface* if its functions include all the functions declared in the interface, thus providing a body for them. However, a class can have more functions than those of the interface.

Encapsulation

Another important principle of object-oriented design is the concept of *encapsulation*, which states that different components of a software system should not reveal the internal details of their respective implementations. One of the main advantages of encapsulation is that it gives the programmer freedom in implementing the details of a system. The only constraint on the programmer is to maintain the abstract interface that outsiders see.

Modularity

In addition to abstraction and encapsulation, a fundamental principle of object-oriented design is *modularity*. Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized. In object-oriented design, this code structuring approach centers around the concept of *modularity*. Modularity refers to an organizing principle for code in which different components of a software system are divided into separate functional units.

Hierarchical Organization

The structure imposed by modularity helps to enable software reusability. If software modules are written in an abstract way to solve general problems, then modules can be reused when instances of these same general problems arise in other contexts.

For example, the structural definition of a wall is the same from house to house, typically being defined in terms of vertical studs, spaced at fixed-distance intervals, etc. Thus, an organized architect can reuse his or her wall definitions from one house to another. In reusing such a definition, some parts may require redefinition, for example, a wall in a commercial building may be similar to that of a house, but the electrical system and stud material might be different.

A natural way to organize various structural components of a software package is in a **hierarchical** fashion, which groups similar abstract definitions together in a level-by-level manner that goes from specific to more general as one traverses up the hierarchy. A common use of such hierarchies is in an organizational chart where each link going up can be read as "is a," as in "a ranch is a house is a building." This kind of hierarchy is useful in software design, for it groups together common functionality at the most general level, and views specialized behavior as an extension of the general one.
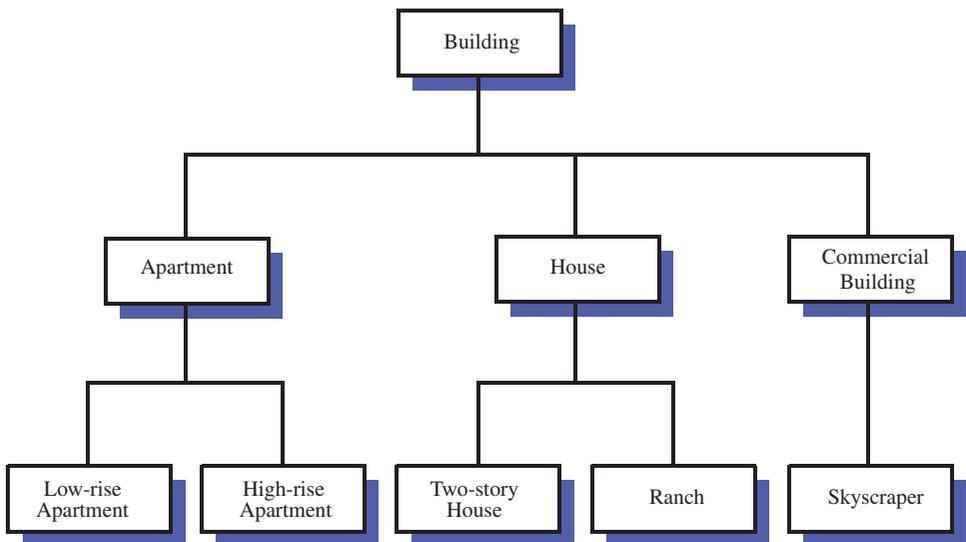
Figure 2.3: An example of an "is a" hierarchy involving architectural buildings.

2.1.3   Design Patterns

One of the advantages of object-oriented design is that it facilitates reusable, ro-
bust, and adaptable software. Designing good code takes more than simply under-
standing object-oriented methodologies, however. It requires the effective use of
object-oriented design techniques.

Computing researchers and practitioners have developed a variety of organiza-
tional concepts and methodologies for designing quality object-oriented software
that is concise, correct, and reusable. Of special relevance to this book is the con-
cept of a ***design pattern***, which describes a solution to a "typical" software design
problem. A pattern provides a general template for a solution that can be applied in
many different situations. It describes the main elements of a solution in an abstract
way that can be specialized for a specific problem at hand. It consists of a name,
which identifies the pattern, a context, which describes the scenarios for which this
pattern can be applied, a template, which describes how the pattern is applied, and
a result, which describes and analyzes what the pattern produces.

We present several design patterns in this book, and we show how they can be
consistently applied to implementations of data structures and algorithms. These
design patterns fall into two groups—patterns for solving algorithm design prob-
lems and patterns for solving software engineering problems. Some of the algo-
rithm design patterns we discuss include the following:

- Recursion (Section 3.5)
- Amortization (Section 6.1.3)
- Divide-and-conquer (Section 11.1.1)
- Prune-and-search, also known as decrease-and-conquer (Section 11.5.1)
- Brute force (Section 12.3.1)
- The greedy method (Section 12.4.2)
- Dynamic programming (Section 12.2)

Likewise, some of the software engineering design patterns we discuss include:

- Position (Section 6.2.1)
- Adapter (Section 5.3.4)
- Iterator (Section 6.2.1)
- Template method (Sections 7.3.7, 11.4, and 13.3.3)
- Composition (Section 8.1.2)
- Comparator (Section 8.1.2)
- Decorator (Section 13.3.1)

Rather than explain each of these concepts here, however, we introduce them
throughout the text as noted above. For each pattern, be it for algorithm engineering
or software engineering, we explain its general use and we illustrate it with at least
one concrete example.

2.2   Inheritance and Polymorphism

To take advantage of hierarchical relationships, which are common in software projects, the object-oriented design approach provides ways of reusing code.

2.2.1   Inheritance in C++

The object-oriented paradigm provides a modular and hierarchical organizing structure for reusing code through a technique called ***inheritance***. This technique allows the design of generic classes that can be specialized to more particular classes, with the specialized classes reusing the code from the generic class. For example, suppose that we are designing a set of classes to represent people at a university. We might have a generic class Person, which defines elements common to all people. We could then define specialized classes such as Student, Administrator, and Instructor, each of which provides specific information about a particular type of person.

A generic class is also known as a ***base class***, ***parent class***, or ***superclass***. It defines "generic" members that apply in a multitude of situations. Any class that ***specializes*** or ***extends*** a base class need not give new implementations for the general functions, for it ***inherits*** them. It should only define those functions that are specialized for this particular class. Such a class is called a ***derived class***, ***child class***, or ***subclass***.

Let us consider an example to illustrate these concepts. Suppose that we are writing a program to deal with people at a university. Below we show a partial implementation of a generic class for a person. We use "// ..." to indicate code that is irrelevant to the example and so has been omitted.

```cpp
class Person {                          // Person (base class)
private:
  string        name;                   // name
  string        idNum;                  // university ID number
public:
  // ...
  void print();                         // print information
  string getName();                     // retrieve name
};
```

Suppose we next wish to define a student object. We can derive our class Stu-

dent from class Person as shown below.

```
class Student : public Person {        // Student (derived from Person)
private:
  string        major;                 // major subject
  int           gradYear;              // graduation year
public:
  // ...
  void print();                        // print information
  void changeMajor(const string& newMajor); // change major
};
```

The "public Person" phrase indicates that the Student is derived from the Person class. (The keyword "**public**" specifies *public inheritance*. We discuss other types of inheritance later.) When we derive classes in this way, there is an implied "is a" relationship between them. In this case, a Student "is a" Person. In particular, a Student object inherits all the member data and member functions of class Person in addition to providing its own members. The relationship between these two classes is shown graphically in a *class inheritance diagram* in Figure 2.4.
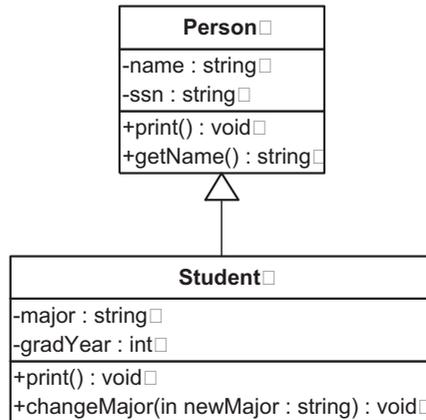
Figure 2.4: A class inheritance diagram, showing a base class Person and derived class Student. Entries tagged with "−" are private and entries tagged with "+" are public. Each block of the diagram consists of three parts: the class name, the class member variables, and the class member functions. The type (or return type) of each member is indicated after the colon (":"). The arrow indicates that Student is derived from Person.

Member Functions

An object of type Person can access the public members of Person. An object of type Student can access the public members of both classes. If a Student object invokes the shared print function, it will use its own version by default. We use the ***class scope operator*** (::) to specify which class's function is used, as in Person::print and Student::print. Note that an object of type Person cannot access members of the base type, and thus it is not possible for a Person object to invoke the changeMajor function of class Student.

```
Person person("Mary", "12-345");   // declare a Person
Student student("Bob", "98-764", "Math", 2012); // declare a Student

cout << student.getName() << endl; // invokes Person::getName()
person.print();                    // invokes Person::print()
student.print();                   // invokes Student::print()
person.changeMajor("Physics");     // ERROR!
student.changeMajor("English");    // okay
```

C++ programmers often find it useful for a derived class to explicitly invoke a member function of a base class. For example, in the process of printing information for a student, it is natural to first print the information of the Person base class, and then print information particular to the student. Performing this task is done using the class scope operator.

```
void Person::print() {                  // definition of Person print
  cout << "Name " << name << endl;
  cout << "IDnum " << idNum << endl;
}

void Student::print() {                 // definition of Student print
  Person::print();                      // first print Person information
  cout << "Major " << major << endl;
  cout << "Year " << gradYear << endl;
}
```

Without the "Person::" specifier used above, the Student::print function would call itself recursively, which is not what we want.

Protected Members

Even though class Student is inherited from class Person, member functions of Student do not have access to private members of Person. For example, the following is illegal.

```
void Student::printName() {
  cout << name << '\n';        // ERROR! name is private to Person
}
```

Special access privileges for derived classes can be provided by declaring members to be "**protected**." A protected member is "public" to all classes derived from this one, but "private" to all other functions. From a syntactic perspective, the keyword **protected** behaves in the same way as the keyword **private** and **public**. In the class example above, had we declared *name* to be protected rather than private, the above function printName would work fine.

Although C++ makes no requirements on the order in which the various sections of a class appear, there are two common ways of doing it. The first is to declare public members first and private members last. This emphasizes the elements that are important to a user of the class. The other way is to present private members first and public members last. This tends to be easier to read for an implementor. Of course, clarity is a more important consideration than adherence to any standard.

Illustrating Class Protection

Consider for example, three classes: a base class Base, a derived class Derived, and an unrelated class Unrelated. The base class defines three integer members, one of each access type.

```
class Base {
  private:      int priv;
  protected:    int prot;
  public:       int publ;
};

class Derived: public Base {
  void someMemberFunction() {
    cout << priv;                // ERROR: private member
    cout << prot;                // okay
    cout << publ;                // okay
  }
};

class Unrelated {
  Base X;

  void anotherMemberFunction() {
    cout << X.priv;              // ERROR: private member
    cout << X.prot;              // ERROR: protected member
    cout << X.publ;              // okay
  }
};
```

When designing a class, we should give careful thought to the access privileges we give each member variable or function. Member variables are almost

always declared to be private or at least protected, since they determine the details of the class's implementation. A user of the class can access only the public class members, which consist of the principal member functions for accessing and manipulating class objects. Finally, protected members are commonly used for utility functions, which may be useful to derived classes. We will see many examples of these three access types in the examples appearing in later chapters.

Constructors and Destructors

We saw in Section 1.5.2, that when a class object is created, the class's constructor is called. When a derived class is constructed, it is the responsibility of this class's constructor to take care that the appropriate constructor is called for its base class. Class hierarchies in C++ are constructed bottom-up: base class first, then its members, then the derived class itself. For this reason, the constructor for a base class needs to be called in the initializer list (see Section 1.5.2) of the derived class. The example below shows how constructors might be implemented for the Person and Student classes.

```
Person::Person(const string& nm, const string& id)
  : name(nm),                        // initialize name
    idNum(id) { }                    // initialize ID number

Student::Student(const string& nm, const string& id,
             const string& maj, int year)
  : Person(nm, id),                  // initialize Person members
    major(maj),                      // initialize major
    gradYear(year) { }               // initialize graduation year
```

Only the Person(nm, id) call has to be in the initializer list. The other initializations could be placed in the constructor function body ({...}), but putting class initializations in the initialization list is generally more efficient. Suppose that we create a new student object.

```
Student* s = new Student("Carol", "34-927", "Physics", 2014);
```

Note that the constructor for the Student class first makes a function call to Person("Carol", "34-927") to initialize the Person base class, and then it initializes the major to "Physics" and the year to 2014.

Classes are destroyed in the reverse order from their construction, with derived classes destroyed before base classes. For example, suppose that we declared destructors for these two classes. (Note that destructors are not really needed in this case, because neither class allocates storage or other resources.)

```
Person::~Person() { ... }           // Person destructor
Student::~Student() { ... }         // Student destructor
```

If we were to destroy our student object, the Student destructor would be called
first, followed by the Person destructor. Unlike constructors, the Student destructor
does not need to (and is not allowed to) call the Person destructor. This happens
automatically.

```
delete s;                              // calls ~Student() then ~Person()
```

Static Binding

When a class is derived from a base class, as with Student and Person, the derived
class becomes a *subtype* of the base class, which means that we can use the derived
class wherever the base class is acceptable. For example, suppose that we create an
array of pointers to university people.

```
Person* pp[100];                       // array of 100 Person pointers
pp[0] = new Person(...);               // add a Person (details omitted)
pp[1] = new Student(...);              // add a Student (details omitted)
```

Since getName is common to both classes, it can be invoked on either elements
of the array. A more interesting issue arises if we attempt to invoke print. Since
$pp[1]$ holds the address of a Student object, we might think that the function Stu-
dent::print would be called. Surprisingly, the function Person::print is called in
both cases, in spite of the apparent difference in the two objects. Furthermore, $pp[i]$
is not even allowed to access Student member functions.

```
cout << pp[1]->getName() << '\n'; // okay
pp[0]->print();                        // calls Person::print()
pp[1]->print();                        // also calls Person::print() (!)
pp[1]->changeMajor("English");      // ERROR!
```

The reason for this apparently anomalous behavior is called *static binding*—when
determining which member function to call, C++'s default action is to consider an
object's *declared type*, not its actual type. Since $pp[1]$ is declared to be a pointer to
a Person, the members for that class are used. Nonetheless, C++ provides a way to
achieve the desired dynamic effect using the technique we describe next.

Dynamic Binding and Virtual Functions

As we saw above, C++ uses *static binding* by default to determine which member
function to call for a derived class. Alternatively, in *dynamic binding*, an object's
contents determine which member function is called. To specify that a member
function should use dynamic binding, the keyword "**virtual**" is added to the func-
tion's declaration. Let us redefine our Person and Student, but this time we will

declare the print function to be virtual.

```
class Person {                          // Person (base class)
  virtual void print() { ... }          // print (details omitted)
  // ...
};
class Student : public Person {         // Student (derived from Person)
  virtual void print() { ... }          // print (details omitted)
  // ...
};
```

Let us consider the effect of this change on our array example, thereby illustrating the usefulness of dynamic binding.

```
Person* pp[100];                        // array of 100 Person pointers
pp[0] = new Person(...);                // add a Person (details omitted)
pp[1] = new Student(...);               // add a Student (details omitted)
pp[0]->print();                         // calls Person::print()
pp[1]->print();                         // calls Student::print()
```

In this case, *pp*[1] contains a pointer to an object of type Student, and by the power of dynamic binding with virtual functions, the function Student::print will be called. The decision as to which function to call is made at run-time, hence the name ***dynamic binding***.

Virtual Destructors

There is no such thing as a virtual constructor. Such a concept does not make any sense. Virtual destructors, however, are very important. In our array example, since we store objects of both types Person and Student in the array, it is important that the appropriate destructor be called for each object. However, if the destructor is nonvirtual, then only the Person destructor will be called in each case. In our example, this choice is not a problem. But if the Student class had allocated memory dynamically, the fact that the wrong destructor is called would result in a memory leak (see Section 1.5.3).

When writing a base class, we cannot know, in general, whether a derived class may need to implement a destructor. So, to be safe, when defining any virtual functions, it is recommended that a virtual destructor be defined as well. This destructor may do nothing at all, and that is fine. It is provided just in case a derived class needs to define its own destructor. This principle is encapsulated in the following rule of thumb.

Remember | If a base class defines any virtual functions, it should define a ***virtual destructor***, even if it is empty.

Dynamic binding is a powerful technique, since it allows us to create an object, such as the array *pp* above, whose behavior varies depending on its contents. This technique is fundamental to the concept of polymorphism, which we discuss in the next section.

2.2.2   Polymorphism

Literally, "polymorphism" means "many forms." In the context of object-oriented design, it refers to the ability of a variable to take different types. Polymorphism is typically applied in C++ using pointer variables. In particular, a variable *p* declared to be a pointer to some class S implies that *p* can point to any object belonging to any derived class T of S.

Now consider what happens if both of these classes define a virtual member function a, and let us consider which of these functions is called when we invoke *p*->a(). Since dynamic binding is used, if *p* points to an object of type T, then it invokes the function T::a. In this case, T is said to **override** function a from S. Alternatively, if *p* points to an object of type S, it will invoke S::a.

Polymorphism such as this is useful because the caller of *p*->a() does not have to know whether the pointer *p* refers to an instance of T or S in order to get the a function to execute correctly. A pointer variable *p* that points to a class object that has at least one virtual function is said to be **polymorphic**. That is, *p* can take many forms, depending on the specific class of the object it is referring to. This kind of functionality allows a specialized class T to extend a class S, inherit the "generic" functions from class S, and redefine other functions from class S to account for specific properties of objects of class T.

Inheritance, polymorphism, and function overloading support reusable software. We can define classes that inherit generic member variables and functions and can then define new, more specific variables and functions that deal with special aspects of objects of the new class. For example, suppose that we defined a generic class Person and then derived three classes Student, Administrator, and Instructor. We could store pointers to all these objects in a list of type Person*. When we invoke a virtual member function, such as print, to any element of the list, it will call the function appropriate to the individual element's type.

Specialization

There are two primary ways of using inheritance, one of which is **specialization**. In using specialization, we are specializing a general class to a particular derived class. Such derived classes typically possess an "is a" relationship to their base class. The derived classes inherit all the members of the base class. For each inherited function, if that function operates correctly, independent of whether it is operating for a specialization, no additional work is needed. If, on the other

hand, a general function of the base class would not work correctly on the derived class, then we should override the function to have the correct functionality for the derived class.

For example, we could have a general class, Dog, which has a function drink and a function sniff. Specializing this class to a Bloodhound class would probably not require that we override the drink function, as all dogs drink pretty much the same way. But it could require that we override the sniff function, as a Bloodhound has a much more sensitive sense of smell than a "generic" dog. In this way, the Bloodhound class specializes the functions of its base class, Dog.

Extension

Another way of using inheritance is **extension**. In using extension, we reuse the code written for functions of the base class, but we then add new functions that are not present in the base class, so as to extend its functionality. For example, returning to our Dog class, we might wish to create a derived class, BorderCollie, which inherits all the generic functions of the Dog class, but then adds a new function, herd, since Border Collies have a herding instinct that is not present in generic dogs, thereby extending the functionality of a generic dog.

2.2.3   Examples of Inheritance in C++

To make the concepts of inheritance and polymorphism more concrete, let us consider a simple example in C++. We consider an example of several classes that print numeric progressions. A **numeric progression** is a sequence of numbers, where the value of each number depends on one or more of the previous values. For example, an **arithmetic progression** determines a next number by addition of a fixed increment. A **geometric progression** determines a next number by multiplication by a fixed base value. In any case, a progression requires a way of defining its first value and it needs a way of identifying the current value as well.

Arithmetic progression (increment 1)	$0, 1, 2, 3, 4, 5, \ldots$
Arithmetic progression (increment 3)	$0, 3, 6, 9, 12, \ldots$
Geometric progression (base 2)	$1, 2, 4, 8, 16, 32, \ldots$
Geometric progression (base 3)	$1, 3, 9, 27, 81, \ldots$

We begin by defining a class, Progression, which is declared in the code fragment below. It defines the "generic" members and functions of a numeric progression. Specifically, it defines the following two long-integer variable members:

- *first*: first value of the progression
- *cur*: current value of the progression

Because we want these variables to be accessible from derived classes, we declare them to be protected.

We define a constructor, Progression, a destructor, ˜Progression, and the following three member functions.

> firstValue(): Reset the progression to the first value and return it.

> nextValue(): Step the progression to the next value and return it.

printProgression(*n*): Reset the progression and print its first *n* values.

```
class Progression {                          // a generic progression
public:
  Progression(long f = 0)                    // constructor
    : first(f), cur(f) { }
  virtual ˜Progression() { };                // destructor
  void printProgression(int n);              // print the first n values
protected:
  virtual long firstValue();                 // reset
  virtual long nextValue();                  // advance
protected:
  long first;                                // first value
  long cur;                                  // current value
};
```

The member function printProgression is public and is defined below.

```
void Progression::printProgression(int n) {  // print n  values
  cout << firstValue();                      // print the first
  for (int i = 2; i <= n; i++)               // print 2 through n
    cout << ' ' << nextValue();
  cout << endl;
}
```

In contrast, the member functions firstValue and nextValue are intended as utilities that will only be invoked from within this class or its derived classes. For this reason, we declare them to be protected. They are defined below.

```
long Progression::firstValue() {             // reset
  cur = first;
  return cur;
}
long Progression::nextValue() {              // advance
  return ++cur;
}
```

It is our intention that, in order to generate different progressions, derived classes will override one or both of these functions. For this reason, we have declared both to be virtual. Because there are virtual functions in our class, we have also provided a virtual destructor in order to be safe. (Recall the discussion of virtual destructors from Section 2.2.1.) At this point the destructor does nothing, but this might be overridden by derived classes.

Arithmetic Progression Class

Let us consider a class ArithProgression, shown below. We add a new member variable *inc*, which provides the value to be added to each new element of the progression. We also override the member function nextValue to produce the desired new behavior.

```
class ArithProgression : public Progression {    // arithmetic progression
public:
  ArithProgression(long i = 1);                   // constructor
protected:
  virtual long nextValue();                       // advance
protected:
  long inc;                                       // increment
};
```

The constructor and the new member function nextValue are defined below. Observe that the constructor invokes the base class constructor Progression to initialize the base object in addition to initializing the value of *inc*.

```
ArithProgression::ArithProgression(long i)    // constructor
  : Progression(), inc(i) { }

long ArithProgression::nextValue() {          // advance by adding
  cur += inc;
  return cur;
}
```

Polymorphism is at work here. When a Progression pointer is pointing to an ArithProgression object, it will use the ArithProgression functions firstValue and nextValue. Even though the function printProgression is not virtual, it makes use of this polymorphism. Its calls to the firstValue and nextValue functions are implicitly for the "current" object, which will be of the ArithProgression class.

A Geometric Progression Class

Let us next define GeomProgression that implements a geometric progression. As with the ArithProgression class, this new class inherits the member variables *first* and *cur*, and the member functions firstValue and printProgression from Progression. We add a new member variable *base*, which holds the base value to be multiplied to form each new element of the progression. The constructor initializes the base class with a starting value of 1 rather than 0. The function nextValue applies

multiplication to obtain the next value.

```
class GeomProgression : public Progression {  // geometric progression
public:
  GeomProgression(long b = 2);                 // constructor
protected:
  virtual long nextValue();                    // advance
protected:
  long base;                                   // base value
};

GeomProgression::GeomProgression(long b)       // constructor
  : Progression(1), base(b) { }

long GeomProgression::nextValue() {            // advance by multiplying
  cur *= base;
  return cur;
}
```

A Fibonacci Progression Class

As a further example, we define a FibonacciProgression class that represents an-
other kind of progression, the ***Fibonacci progression***, where the next value is de-
fined as the sum of the current and previous values. We show the FibonacciPro-
gression class below. Recall that each element of a Fibonacci series is the sum of
the previous two elements.

$$\text{Fibonacci progression (first} = 0, \text{second} = 1): \quad 0, 1, 1, 2, 3, 5, 8, \dots$$

In addition to the current value *cur* in the Progression base class, we also store
here the value of the previous element, denoted *prev*. The constructor is given the
first two elements of the sequence. The member variable *first* is inherited from the
base class. We add a new member variable *second*, to store this second element.
The default values for the first and second elements are 0 and 1, respectively.

```
class FibonacciProgression : public Progression { // Fibonacci progression
public:
  FibonacciProgression(long f = 0, long s = 1); // constructor
protected:
  virtual long firstValue();                   // reset
  virtual long nextValue();                    // advance
protected:
  long second;                                 // second value
  long prev;                                   // previous value
};
```

The initialization process is a bit tricky because we need to create a "fictitious"
element that precedes the first element. Note that setting this element to the value

second − *first* achieves the desired result. This change is reflected both in the constructor and the overridden member function firstValue. The overridden member function nextValue copies the current value to the previous value. We need to store the old previous value in a temporary variable.

```
FibonacciProgression::FibonacciProgression(long f, long s)
  : Progression(f), second(s), prev(second − first)  { }

long FibonacciProgression::firstValue() {          // reset
  cur = first;
  prev = second − first;                           // create fictitious prev
  return cur;
}

long FibonacciProgression::nextValue() {           // advance
  long temp = prev;
  prev = cur;
  cur += temp;
  return cur;
}
```

Combining the Progression Classes

In order to visualize how the three different progression classes are derived from the generic Progression class, we give their inheritance diagram in Figure 2.5.
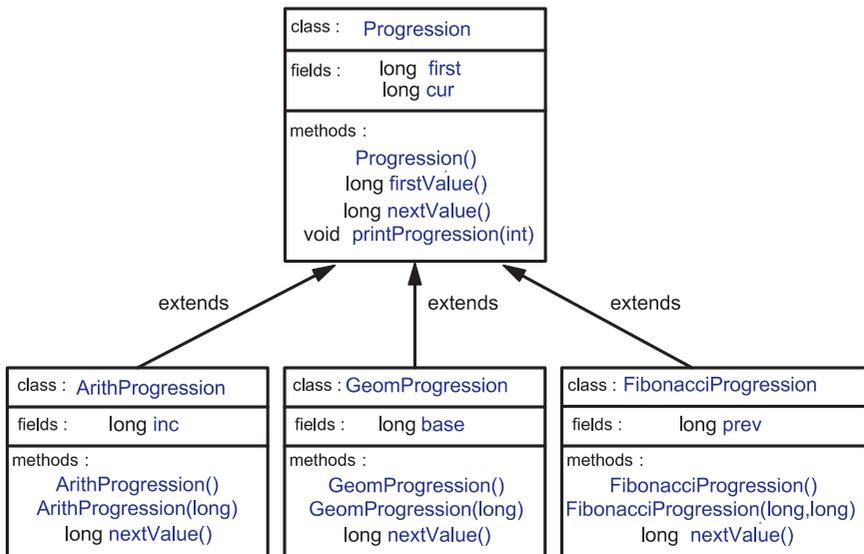
Figure 2.5: Inheritance diagram for class Progression and its subclasses.

To complete our example, we define the main function shown in Code Fragment 2.1, which performs a simple test of each of the three classes. In this class, variable *prog* is a polymorphic array of pointers to class Progression. Since each of its members points to an object of class ArithProgression, GeomProgression, or FibonacciProgression, the functions appropriate to the given progression are invoked in each case. The output is shown in Code Fragment 2.2. Notice that this program has a (unimportant) memory leak because we never deleted the allocated object.

The example presented in this section provides a simple illustration of inheritance and polymorphism in C++. The Progression class, its derived classes, and the tester program have a number of shortcomings, however, which might not be immediately apparent. One problem is that the geometric and Fibonacci progressions grow quickly, and there is no provision for handling the inevitable overflow of the long integers involved. For example, since $3^{40} > 2^{63}$, a geometric progression with base $b = 3$ will overflow a 64-bit long integer after 40 iterations. Likewise, the 94th Fibonacci number is greater than 2^{63}; hence, the Fibonacci progression will overflow a 64-bit long integer after 94 iterations. Another problem is that we may not allow arbitrary starting values for a Fibonacci progression. For example, do we allow a Fibonacci progression starting with 0 and -1? Dealing with input errors or error conditions that occur during the running of a C++ program requires that we have some mechanism for handling them. We discuss this topic later in Section 2.4.

2.2.4   Multiple Inheritance and Class Casting

In the examples we have shown so far, a subclass has been derived from a single base class and we didn't have to deal with the problem of viewing an object of a specific declared class as also being of an inherited type. We discuss some related, more-advanced C++ programming issues in this section.

Multiple and Restricted Inheritance

In C++, we are allowed to derive a class from a number of base classes, that is, C++ allows *multiple inheritance*. Although multiple inheritance can be useful, especially in defining interfaces, it introduces a number of complexities. For example, if both base classes provide a member variable with the same name or a member function with the same declaration, the derived class must specify from which base class the member should be used (which is complicated). For this reason, we use single inheritance almost exclusively.

We have been using public inheritance in our previous examples, indicated by the keyword **public** in specifying the base class. Remember that private base class members are not accessible in a derived class. Protected and public members of the base class become protected and public members of the derived class, respectively.

```
/** Test program for the progression classes */
int main() {
  Progression* prog;
                                    // test ArithProgression
  cout << "Arithmetic progression with default increment:\n";
  prog = new ArithProgression();
  prog->printProgression(10);
  cout << "Arithmetic progression with increment 5:\n";
  prog = new ArithProgression(5);
  prog->printProgression(10);
                                    // test GeomProgression
  cout << "Geometric progression with default base:\n";
  prog = new GeomProgression();
  prog->printProgression(10);
  cout << "Geometric progression with base 3:\n";
  prog = new GeomProgression(3);
  prog->printProgression(10);
                                    // test FibonacciProgression
  cout << "Fibonacci progression with default start values:\n";
  prog = new FibonacciProgression();
  prog->printProgression(10);
  cout << "Fibonacci progression with start values 4 and 6:\n";
  prog = new FibonacciProgression(4, 6);
  prog->printProgression(10);
  return EXIT_SUCCESS;              // successful execution
}
```

Code Fragment 2.1: Program for testing the progression classes.

```
Arithmetic progression with default increment:
0 1 2 3 4 5 6 7 8 9
Arithmetic progression with increment 5:
0 5 10 15 20 25 30 35 40 45
Geometric progression with default base:
1 2 4 8 16 32 64 128 256 512
Geometric progression with base 3:
1 3 9 27 81 243 729 2187 6561 19683
Fibonacci progression with default start values:
0 1 1 2 3 5 8 13 21 34
Fibonacci progression with start values 4 and 6:
4 6 10 16 26 42 68 110 178 288
```

Code Fragment 2.2: Output of TestProgression program from Code Fragment 2.1.

C++ supports two other types of inheritance. These different types of inheritance diminish the access rights for base class members. In ***protected inheritance***, fields declared to be public in the base class become protected in the child class. In ***private inheritance***, fields declared to be public and protected in the base class become private in the derived class. An example is shown below.

```
class Base {                                // base class
  protected:    int foo;
  public:       int bar;
};

class Derive1 : public Base {               // public inheritance
  // foo is protected and bar is public
};

class Derive2 : protected Base {            // protected inheritance
  // both foo and bar are protected
};

class Derive3 : private Base {              // public inheritance
  // both foo and bar are private
};
```

Protected and private inheritance are not used as often as public inheritance. We only use public inheritance in this book.

Casting in an Inheritance Hierarchy

An object variable can be viewed as being of various types, but it can be declared as only one type. Thus, a variable's declared type determines how it is used, and even determines how certain functions will act on it. Enforcing that all variables be typed and that operations declare the types they expect is called ***strong typing***, which helps prevent bugs. Nonetheless, we sometimes need to explicitly change, or ***cast***, a variable from one type to another. We have already introduced type casting in Section 1.2.1. We now discuss how it works for classes.

To illustrate an example where we may want to perform a cast, recall our class hierarchy consisting of a base class Person and derived class Student. Suppose that we are storing pointers to objects of both types in an array *pp*. The following attempt to change a student's major would be flagged as an error by the compiler.

```
Person* pp[100];                      // array of 100 Person pointers
pp[0] = new Person(...);              // add a Person (details omitted)
pp[1] = new Student(...);             // add a Student (details omitted)
// ...
pp[1]−>changeMajor("English");     // ERROR!
```

The problem is that the base class Person does not have a function changeMajor. Notice that this is different from the case of the function print because the print function was provided in both classes. Nonetheless, we "know" that $pp[1]$ points to an object of class Student, so this operation should be legal.

To access the changeMajor function, we need to cast the $pp[1]$ pointer from type Person* to type Student*. Because the contents of a variable are dynamic, we need to use the C++ run-time system to determine whether this cast is legal, which is what a ***dynamic cast*** does. The syntax of a dynamic cast is shown below.

   dynamic_cast < *desired_type* > (*expression*)

Dynamic casting can only be applied to polymorphic objects, that is, objects that come from a class with at least one virtual function. Below we show how to use dynamic casting to change the major of $pp[1]$.

```
Student* sp = dynamic_cast<Student*>(pp[1]); // cast pp[1] to Student*
sp->changeMajor("Chemistry");                // now changeMajor is legal
```

Dynamic casting is most often applied for casting pointers within the class hierarchy. If an illegal pointer cast is attempted, then the result is a null pointer. For example, we would get a NULL pointer from an attempt to cast $pp[0]$ as above, since it points to a Person object.

To illustrate the use of dynamic cast, we access all the elements of the pp array and, for objects of (actual) type Student, change the major to "Undecided"

```
for (int i = 0; i < 100; i++) {
  Student *sp = dynamic_cast<Student*>(pp[i]);
  if (sp != NULL)                          // cast succeeded?
    sp->changeMajor("Undecided");          // change major
}
```

The casting we have discussed here could also have been done using the traditional C-style cast or through a static cast (recall Section 1.2.1). Unfortunately, no error checking would be performed in that case. An attempt to cast a Person object pointer to a Student pointer would succeed "silently," but any attempt to use such a pointer would have disastrous consequences.

2.2.5   Interfaces and Abstract Classes

For two objects to interact, they must "know" about each other's member functions. To enforce this "knowledge," the object-oriented design paradigm asks that classes specify the ***application programming interface*** (API), or simply ***interface***, that their objects present to other objects. In the ***ADT-based*** approach (see Section 2.1.2) to data structures followed in this book, an interface defining an ADT

is specified as a type definition and a collection of member functions for this type, with the arguments for each function being of specified types.

Some programming languages provide a mechanism for defining ADTs. One example is Java's *interface*. An interface is a collection of function declarations with no data and no bodies. That is, the member functions of an interface are always empty. When a class implements an interface, it must implement all of the member functions declared in the interface.

C++ does not provide a direct mechanism for specifying interfaces. Nonetheless, throughout this book we often provide *informal interfaces*, even though they are not legal C++ structures. For example, a *stack* data structure (see Chapter 5) is a container that supports various operations such as inserting (or *pushing*) an element onto the top of the stack, removing (or *popping*) an element from the top of the stack, and testing whether the stack is empty. Below we provide an example of a minimal interface for a stack of integers.

```cpp
class Stack {                        // informal interface – not a class
public:
  bool isEmpty() const;              // is the stack empty?
  void push(int x);                  // push x onto the stack
  int pop();                         // pop the stack and return result
};
```

Abstract Classes

The above informal interface is *not* a valid construct in C++; it is just a documentation aid. In particular, it does not contain any data members or definitions of member functions. Nonetheless, it is useful, since it provides important information about a stack's public member functions and how they are called.

An *abstract class* in C++ is a class that is used only as a base class for inheritance; it cannot be used to create instances directly. At first the idea of creating a class that cannot be instantiated seems to be nonsense, but it is often very important. For example, suppose that we want to define a set of geometric shape classes, say, Circle, Rectangle, and Triangle. It is natural to derive these related classes from a single generic base class, say, Shape. Each of the derived classes will have a virtual member function draw, which draws the associated object. The rules of inheritance require that we define such a function for the base class, but it is unclear what such a function means for a generic shape.

One way to handle this would be to define Shape::draw with an empty function body ({ }), which would be a rather unnatural solution. What is really desired here is some way to inform the compiler that the class Shape is *abstract*; it is not possible to create objects of type Shape, only its subclasses. In C++, we define a class as being abstract by specifying that one or more members of its functions are *abstract*, or *pure virtual*. A function is declared pure virtual by giving "=0" in

place of its body. C++ does not allow the creation of an object that has one or more pure virtual functions. Thus, any derived class must provide concrete definitions for all pure virtual functions of the base class.

As an example, recall our Progression class and consider the member function nextValue, which computes the next value in the progression. The meaning of this function is clear for each of the derived classes: ArithProgression, GeomProgression, and FibonacciProgression. However, in the base class Progression we invented a rather arbitrary default for the nextValue function. (Go back and check it. What progression does it compute?) It would be more natural to leave this function undefined. We show below how to make it a ***pure virtual*** member function.

```cpp
class Progression {                 // abstract base class
  // ...
  virtual long nextValue() = 0;     // pure virtual function
  // ...
};
```

As a result, the compiler will not allow the creation of objects of type Progression, since the function nextValue is "pure virtual." However, its derived classes, ArithProgression for example, can be defined because they provide a definition for this member function.

Interfaces and Abstract Base Classes

We said above that C++ does not provide a direct mechanism for defining interfaces for abstract data types. Nevertheless, we can use abstract base classes to achieve much of the same purpose.

In particular, we may construct a class for an interface in which all the functions are pure virtual as shown below for the example of a simple stack ADT.

```cpp
class Stack {                          // stack interface as an abstract class
public:
  virtual bool isEmpty() const = 0;    // is the stack empty?
  virtual void push(int x) = 0;        // push x onto the stack
  virtual int pop() = 0;               // pop the stack and return result
};
```

A class that implements this stack interface can be derived from this abstract base class, and then provide concrete definitions for all of these virtual functions as

shown below.

```
class ConcreteStack : public Stack {  // implements Stack
public:
  virtual bool isEmpty() { ... }        // implementation of members
  virtual void push(int x) { ... }      // ... (details omitted)
  virtual int pop() { ... }
private:
  // ...                                // member data for the implementation
};
```

There are practical limitations to this method of defining interfaces, so we only use informal interfaces for the purpose of illustrating ADTs.

2.3   Templates

Inheritance is only one mechanism that C++ provides in support of polymorphism. In this section, we consider another way—using *templates*.

2.3.1   Function Templates

Let us consider the following function, which returns the minimum of two integers.

```
int integerMin(int a, int b)          // returns the minimum of a and b
  { return (a < b ? a : b); }
```

Such a function is very handy, so we might like to define a similar function for computing the minimum of two variables of other types, such as long, short, float, and double. Each such function would require a different declaration and definition, however, and making many copies of the same function is an error-prone solution, especially for longer functions.

C++ provides an automatic mechanism, called the *function template*, to produce a generic function for an arbitrary type T. A function template provides a well-defined pattern from which a concrete function may later be formally defined or *instantiated*. The example below defines a genericMin function template.

```
template <typename T>
T genericMin(T a, T b) {               // returns the minimum of a and b
  return (a < b ? a : b);
}
```

The declaration takes the form of the keyword "**template**" followed by the notation <typename T>, which is the parameter list for the template. In this case, there is

just one parameter T. The keyword "**typename**" indicates that T is the name of some type. (Older versions of C++ do not support this keyword and instead the keyword "**class**" must be used.) We can have other types of template parameters, integers for example, but type names are the most common. Observe that the type parameter T takes the place of "**int**" in the original definition of the genericMin function.

We can now invoke our templated function to compute the minimum of objects of many different types. The compiler looks at the argument types and determines which form of the function to *instantiate*.

```
cout << genericMin(3, 4) << ' '     // = genericMin<int>(3,4)
    << genericMin(1.1, 3.1) << ' '   // = genericMin<double>(1.1, 3.1)
    << genericMin('t', 'g') << endl; // = genericMin<char>('t','g')
```

The template type does not need to be a fundamental type. We could use any type in this example, provided that the less than operator (<) is defined for this type.

2.3.2 Class Templates

In addition to function templates, C++ allows classes to be templated, which is a powerful mechanism because it allows us to provide one data structure declaration that can be applied to many different types. In fact, the Standard Template Library uses class templates extensively.

Let us consider an example of a template for a restricted class BasicVector that stores a vector of elements, which is a simplified version of a structure discussed in greater detail in Chapter 6. This class has a constructor that is given the size of the array to allocate. In order to access elements of the array, we overload the indexing operator "[ ]."

We present a partial implementation of a class template for class BasicVector below. We have omitted many of the other member functions, such as the copy constructor, assignment operator, and destructor. The template parameter T takes the place of the actual type that will be stored in the array.

```
template <typename T>
class BasicVector {                 // a simple vector class
public:
  BasicVector(int capac = 10);      // constructor
  T& operator[](int i)              // access element at index i
    { return a[i]; }
  // ... other public members omitted
private:
  T* a;                             // array storing the elements
  int capacity;                     // length of array a
};
```

We have defined one member function (the indexing operator) within the class body, and below we show how the other member function (the constructor) can be defined outside the class body. The constructor initializes the capacity value and allocates the array storage.

```
template <typename T>                // constructor
BasicVector<T>::BasicVector(int capac) {
  capacity = capac;
  a = new T[capacity];               // allocate array storage
}
```

To *instantiate* a concrete instance of the class BasicVector, we provide the class name followed by the actual type parameter enclosed in angled brackets (<...>). The code fragment below shows how we would define three vectors, one of type **int**, one of type **double**, and one of type string.

```
BasicVector<int>     iv(5);          // vector of 5 integers
BasicVector<double>  dv(20);         // vector of 20 doubles
BasicVector<string>  sv(10);         // vector of 10 strings
```

Since we have overloaded the indexing operator, we can access elements of each array in the same manner as we would for any C++ array.

```
iv[3] = 8;
dv[14] = 2.5;
sv[7] = "hello";
```

Templated Arguments

The actual argument in the instantiation of a class template can itself be a templated type. For example, we could create a BasicVector whose individual elements are themselves of type BasicVector<int>.

```
BasicVector<BasicVector<int> > xv(5); // a vector of vectors
// . . .
xv[2][8] = 15;
```

In this case, because no capacity argument could be provided to the constructor, each element of the vector is constructed using the default capacity of 10. Thus the above definition declares a BasicVector consisting of five elements, each of which is a BasicVector consisting of 10 integers. Such a structure therefore behaves much like a two-dimensional array of integers.

Note that in the declaration of *xv* above, we intentionally left a space after "**<int>**." The reason is that without the space, the character combination ">>" would be interpreted as a bitwise right-shift operator by the compiler (see Section 1.2).

2.4 Exceptions

Exceptions are unexpected events that occur during the execution of a program. An exception can be the result of an error condition or simply an unanticipated input. In C++, exceptions can be thought of as being objects themselves.

2.4.1 Exception Objects

In C++, an exception is "***thrown***" by code that encounters some unexpected condition. Exceptions can also be thrown by the C++ run-time environment should it encounter an unexpected condition like running out of memory. A thrown exception is "***caught***" by other code that "handles" the exception somehow, or the program is terminated unexpectedly. (We say more about catching exceptions shortly.)

Exceptions are a relatively recent addition to C++. Prior to having exceptions, errors were typically handled by having the program abort at the source of the error or by having the involved function return some special value. Exceptions provide a much cleaner mechanism for handling errors. Nevertheless, for historical reasons, many of the functions in the C++ standard library do not throw exceptions. Typically they return some sort of special error status, or set an error flag, which can be tested.

Exceptions are thrown when a piece of code finds some sort of problem during execution. Since there are many types of possible errors, when an exception is thrown, it is identified by a type. Typically this type is a class whose members provide information as to the exact nature of the error, for example a string containing a descriptive error message.

Exception types often form hierarchies. For example, let's imagine a hypothetical mathematics library, which may generate many different types of errors. The library might begin by defining one generic exception, MathException, representing all types of mathematical errors, and then derive more specific exceptions for particular error conditions. The *errMsg* member holds a message string with an informative message. Here is a possible definition of this generic class.

```cpp
class MathException {                        // generic math exception
public:
  MathException(const string& err)          // constructor
    : errMsg(err) { }
  string getError() { return errMsg; }       // access error message
private:
  string errMsg;                             // error message
};
```

Using Inheritance to Define New Exception Types

The above MathException class would likely have other member functions, for example, for accessing the error message. We may then add more specific exceptions, such as ZeroDivide, to handle division by zero, and NegativeRoot, to handle attempts to compute the square root of a negative number. We could use class inheritance to represent this hierarchical relationship, as follows.

```cpp
class ZeroDivide : public MathException {
public:
  ZeroDivide(const string& err)                 // divide by zero
   : MathException(err) { }
};

class NegativeRoot : public MathException {
public:
  NegativeRoot(const string& err)               // negative square root
   : MathException(err) { }
};
```

2.4.2 Throwing and Catching Exceptions

Exceptions are typically processed in the context of "try" and "catch" blocks. A *try block* is a block of statements proceeded by the keyword **try**. After a try block, there are one or more *catch blocks*. Each catch block specifies the type of exception that it catches. Execution begins with the statements of the try block. If all goes smoothly, then execution leaves the try block and skips over its associated catch blocks. If an exception is thrown, then the control immediately jumps into the appropriate catch block for this exception.

For example, suppose that we were to use our mathematical library as part of the implementation of a numerical application. We would enclose the computations of the application within a try block. After the try block, we would catch and deal with any exceptions that arose in the computation.

```cpp
try {
  // ... application computations
  if (divisor == 0)                             // attempt to divide by 0?
    throw ZeroDivide("Divide by zero in Module X");
}
catch (ZeroDivide& zde) {
  // handle division by zero
}
catch (MathException& me) {
  // handle any math exception other than division by zero
}
```

Processing the above try block is done as follows. The computations of the try block are executed. When an attempt is discovered to divide by zero, ZeroDivide is thrown, and execution jumps immediately to the associated **catch** statement where corrective recovery and clean up should be performed.

Let us study the entire process in somewhat greater detail. The **throw** statement is typically written as follows:

   throw *exception_name*(*arg1*, *arg2*, ...)

where the arguments are passed to the exception's constructor.

Exceptions may also be thrown by the C++ run-time system itself. For example, if an attempt to allocate space in the free store using the **new** operator fails due to lack of space, then a bad_alloc exception is thrown by the system.

When an exception is thrown, it must be *caught* or the program will abort. In any particular function, an exception in that function can be passed through to the calling function or it can be caught in that function. When an exception is caught, it can be analyzed and dealt with. The general syntax for a ***try-catch block*** in C++ is as follows:

   try
       try_statements
   catch (*exception_type_1 identifier_1*)
       catch_statements_1
   . . .
   catch (*exception_type_n identifier_n*)
       catch_statements_n

Execution begins in the "*try_statements*." If this execution generates no exceptions, then the flow of control continues with the first statement after the last line of the entire try-catch block. If, on the other hand, an exception is generated, execution in the try block terminates at that point and execution jumps to the first catch block matching the exception thrown. Thus, an exception thrown for a derived class will be caught by its base class. For example, if we had thrown NegativeRoot in the example above, it would be caught by catch block for MathException. Note that because the system executes the first matching catch block, exceptions should be listed in order of most specific to least specific. The special form "catch(...)" catches ***all*** exceptions.

The "*identifier*" for the catch statement identifies the exception object itself. As we said before, this object usually contains additional information about the exception, and this information may be accessed from within the catch block. As is common in passing class arguments, the exception is typically passed as a reference or a constant reference. Once execution of the catch block completes, control flow continues with the first statement after the last catch block.

The recovery action taken in a catch block depends very much on the particular application. It may be as simple as printing an error message and terminating the

program. It may require complex clean-up operations, such as deallocating dynamically allocated storage and restoring the program's internal state. There are also some interesting cases in which the best way to handle an exception is to ignore it (which can be specified by having an empty catch block). Ignoring an exception is usually done, for example, when the programmer does not care whether there was an exception or not. Another legitimate way of handling exceptions is to throw another exception, possibly one that specifies the exceptional condition more precisely.

2.4.3   Exception Specification

When we declare a function, we should also specify the exceptions it might throw. This convention has both a functional and courteous purpose. For one, it lets users know what to expect. It also lets the compiler know which exceptions to prepare for. The following is an example of such a function definition.

```
void calculator() throw(ZeroDivide, NegativeRoot) {
    // function body ...
}
```

This definition indicates that the function calculator (and any other functions it calls) can throw these two exceptions or exceptions derived from these types, but no others.

By specifying all the exceptions that might be thrown by a function, we prepare others to be able to handle all of the exceptional cases that might arise from using this function. Another benefit of declaring exceptions is that we do not need to catch those exceptions in our function, which is appropriate, for example, in the case where other code is responsible for causing the circumstances leading up to the exception.

The following illustrates an exception that is "passed through."

```
void getReadyForClass() throw(ShoppingListTooSmallException,
                              OutOfMoneyException) {
    goShopping();  // I don't have to try or catch the exceptions
                   // which goShopping() might throw because
                   // getReadyForClass() will just pass these along.
    makeCookiesForTA();
}
```

A function can declare that it throws as many exceptions as it likes. Such a listing can be simplified somewhat if all exceptions that can be thrown are derived classes of the same exception. In this case, we only have to declare that a function throws the appropriate base class.

Suppose that a function does not contain a **throw** specification. It would be natural to assume that such a function does not throw any exceptions. In fact, it has quite a different meaning. If a function does not provide a **throw** specification, then it may throw *any* exception. Although this is confusing, it is necessary to maintain compatibility with older versions of C++. To indicate that a function throws no exceptions, provide the **throw** specifier with an empty list of exceptions.

```
void func1();                        // can throw any exception
void func2() throw();                // can throw no exceptions
```

Generic Exception Class

We declare many different exceptions in this book. In order to structure these exceptions hierarchically, we need to have one generic exception class that serves as the "mother of all exceptions." C++ does not provide such a generic exception, so we created one of our own. This class, called RuntimeException, is shown below. It has an error message as its only member. It provides a constructor that is given an informative error message as its argument. It also provides a member function getMessage that allows us to access this message.

```
class RuntimeException {             // generic run-time exception
private:
  string errorMsg;
public:
  RuntimeException(const string& err) { errorMsg = err; }
  string getMessage() const { return errorMsg; }
};
```

By deriving all of our exceptions from this base class, for any exception *e*, we can output *e*'s error message by invoking the inherited getMessage function.

2.5   Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-2.1 What are some potential efficiency disadvantages of having very deep inheritance trees, that is, a large set of classes, A, B, C, and so on, such that B extends A, C extends B, D extends C, etc.?

R-2.2 What are some potential efficiency disadvantages of having very shallow inheritance trees, that is, a large set of classes, A, B, C, and so on, such that all of these classes extend a single class, Z?

R-2.3 Give three examples of life-critical software applications.

R-2.4 Give an example of a software application where adaptability can mean the difference between a prolonged sales lifetime and bankruptcy.

R-2.5 Describe a component from a text-editor GUI (other than an "edit" menu) and the member functions that it encapsulates.

R-2.6 Draw a class inheritance diagram for the following set of classes.

- Class Goat extends Object and adds a member variable *tail* and functions milk and jump.
- Class Pig extends Object and adds a member variable *nose* and functions eat and wallow.
- Class Horse extends Object and adds member variables *height* and *color*, and functions run and jump.
- Class Racer extends Horse and adds a function race.
- Class Equestrian extends Horse and adds a member variable *weight* and functions trot and isTrained.

R-2.7 A derived class's constructor explicitly invokes its base class's constructor, but a derived class's destructor cannot invoke its base class's destructor. Why does this apparent asymmetry make sense?

R-2.8 Give a short fragment of C++ code that uses the progression classes from Section 2.2.3 to find the 7th value of a Fibonacci progression that starts with 3 and 4 as its first two values.

R-2.9 If we choose $inc = 128$, how many calls to the nextValue function from the ArithProgression class of Section 2.2.3 can we make before we cause a long-integer overflow, assuming a 64-bit long integer?

R-2.10 Suppose we have a variable *p* that is declared to be a pointer to an object of type Progression using the classes of Section 2.2.3. Suppose further that *p* actually points to an instance of the class GeomProgression that was created with the default constructor. If we cast *p* to a pointer of type Progression and call *p*->firstValue(), what will be returned? Why?

R-2.11 Consider the inheritance of classes from Exercise R-2.6, and let *d* be an object variable of type Horse. If *d* refers to an actual object of type Equestrian, can it be cast to the class Racer? Why or why not?

R-2.12 Generalize the Person-Student class hierarchy to include classes Faculty, UndergraduateStudent, GraduateStudent, Professor, Instructor. Explain the inheritance structure of these classes, and derive some appropriate member variables for each class.

R-2.13 Give an example of a C++ code fragment that performs an array reference that is possibly out of bounds, and if it is out of bounds, the program catches that exception and prints an appropriate error message.

R-2.14 Consider the following code fragment:

```cpp
class Object
  { public: virtual void printMe() = 0; };
class Place : public Object
  { public: virtual void printMe() { cout << "Buy it.\n"; } };
class Region : public Place
  { public: virtual void printMe() { cout << "Box it.\n"; } };
class State : public Region
  { public: virtual void printMe() { cout << "Ship it.\n"; } };
class Maryland : public State
  { public: virtual void printMe() { cout << "Read it.\n"; } };

int main() {
  Region*  mid = new State;
  State*   md  = new Maryland;
  Object*  obj = new Place;
  Place*   usa = new Region;
  md->printMe();
  mid->printMe();
  (dynamic_cast<Place*>(obj))->printMe();
  obj = md;
  (dynamic_cast<Maryland*>(obj))->printMe();
  obj = usa;
  (dynamic_cast<Place*>(obj))->printMe();
  usa = md;
  (dynamic_cast<Place*>(usa))->printMe();
  return EXIT_SUCCESS;
}
```

What is the output from calling the main function of the Maryland class?

R-2.15 Write a short C++ function that counts the number of vowels in a given character string.

R-2.16 Write a short C++ function that removes all the punctuation from a string s storing a sentence. For example, this operation would transform the string `"Let's try, Mike."` to `"Lets try Mike"`.

R-2.17 Write a short program that takes as input three integers, a, b, and c, and determines if they can be used in a correct arithmetic formula (in the given order), like "$a + b = c$," "$a = b - c$," or "$a * b = c$."

R-2.18 Write a short C++ program that creates a Pair class that can store two objects declared as generic types. Demonstrate this program by creating and printing Pair objects that contain five different kinds of pairs, such as <int,string> and <float,long>.

Creativity

C-2.1 Give an example of a C++ program that outputs its source code when it is run. Such a program is called a *quine*.

C-2.2 Suppose you are on the design team for a new e-book reader. What are the primary classes and functions that the C++ software for your reader will need? You should include an inheritance diagram for this code, but you don't need to write any actual code. Your software architecture should at least include ways for customers to buy new books, view their list of purchased book, and read their purchased books.

C-2.3 Most modern C++ compilers have optimizers that can detect simple cases when it is logically impossible for certain statements in a program to ever be executed. In such cases, the compiler warns the programmer about the useless code. Write a short C++ function that contains code for which it is provably impossible for that code to ever be executed, but your favorite C++ compiler does not detect this fact.

C-2.4 Design a class Line that implements a line, which is represented by the formula $y = ax + b$. Your class should store a and b as **double** member variables. Write a member function intersect(ℓ) that returns the x coordinate at which this line intersects line ℓ. If the two lines are parallel, then your function should throw an exception Parallel. Write a C++ program that creates a number of Line objects and tests each pair for intersection. Your program should print an appropriate error message for parallel lines.

C-2.5 Write a C++ class that is derived from the Progression class to produce a progression where each value is the absolute value of the difference between the previous two values. You should include a default constructor that starts with 2 and 200 as the first two values and a parametric constructor that starts with a specified pair of numbers as the first two values.

C-2.6 Write a C++ class that is derived from the Progression class to produce a progression where each value is the square root of the previous value. (Note that you can no longer represent each value with an integer.) You should include a default constructor that starts with 65,536 as the first value and a parametric constructor that starts with a specified (**double**) number as the first value.

C-2.7 Write a program that consists of three classes, A, B, and C, such that B is a subclass of A and C is a subclass of B. Each class should define a member variable named "x" (that is, each has its own variable named x). Describe a way for a member function in C to access and set A's version of x to a given value, without changing B or C's version.

C-2.8 Write a set of C++ classes that can simulate an Internet application, where one party, Alice, is periodically creating a set of packets that she wants to send to Bob. The Internet process is continually checking if Alice has any packets to send, and if so, it delivers them to Bob's computer, and Bob is periodically checking if his computer has a packet from Alice, and, if so, he reads and deletes it.

C-2.9 Write a C++ program that can input any polynomial in standard algebraic notation and outputs the first derivative of that polynomial.

Projects

P-2.1 Write a C++ program that can take a positive integer greater than 2 as input and write out the number of times one must repeatedly divide this number by 2 before getting a value less than 2.

P-2.2 Write a C++ program that "makes change." Your program should input two numbers, one that is a monetary amount charged and the other that is a monetary amount given. It should return the number of each kind of bill and coin to give back as change for the difference between the amounts given and charged. The values assigned to the bills and coins can be based on the monetary system of any government. Try to design your program so that it returns the fewest number of bills and coins as possible.

P-2.3 Implement a templated C++ class Vector that manipulates a numeric vector. Your class should be templated with any numerical scalar type T, which supports the operations $+$ (addition), $-$ (subtraction), and $*$ (multiplication). In addition, type T should have constructors $T(0)$, which produces the additive identity element (typically 0) and $T(1)$, which produces the multiplicative identity (typically 1). Your class should provide a constructor, which is given the size of the vector as an argument. It should provide member functions (or operators) for vector addition, vector subtraction, multiplication of a scalar and a vector, and vector dot product.

Write a class Complex that implements a complex number by overloading the operators for addition, subtraction, and multiplication. Implement three concrete instances of your class Vector with the scalar types **int**, **double**, and Complex, respectively.

P-2.4 Write a simulator as in the previous project, but add a Boolean gender field and a floating-point strength field to each Animal object. Now, if two animals of the same type try to collide, then they only create a new instance of that type of animal if they are of different genders. Otherwise, if two animals of the same type and gender try to collide, then only the one of larger strength survives.

P-2.5 Write a C++ program that has a Polygon interface that has abstract functions, area(), and perimeter(). Implement classes for Triangle, Quadrilateral, Pentagon, Hexagon, and Octagon, which implement this interface, with the obvious meanings for the area() and perimeter() functions. Also implement classes, IsoscelesTriangle, EquilateralTriangle, Rectangle, and Square, which have the appropriate inheritance relationships. Finally, write a simple user interface that allows users to create polygons of the various types, input their geometric dimensions, and then output their area and perimeter. For extra effort, allow users to input polygons by specifying their vertex coordinates and be able to test if two such polygons are similar.

P-2.6 Write a C++ program that inputs a document and then outputs a bar-chart plot of the frequencies of each alphabet character that appears in that document.

P-2.7 Write a C++ program that inputs a list of words separated by whitespace, and outputs how many times each word appears in the list. You need not worry about efficiency at this point, however, as this topic is something that will be addressed later in this book.

Chapter Notes

For a broad overview of developments in computer science and engineering, we refer the reader to *The Computer Science and Engineering Handbook* [96]. For more information about the Therac-25 incident, please see the paper by Leveson and Turner [63].

The reader interested in studying object-oriented programming further, is referred to the books by Booch [13], Budd [16], and Liskov and Guttag [68]. Liskov and Guttag [68] also provide a nice discussion of abstract data types, as does the survey paper by Cardelli and Wegner [19] and the book chapter by Demurjian [27] in the *The Computer Science and Engineering Handbook* [96]. Design patterns are described in the book by Gamma *et al.* [35]. The class inheritance diagram notation we use is derived from the Gamma *et al.*

Chapter

3

Arrays, Linked Lists, and Recursion

Contents

3.1   Using Arrays

In this section, we explore a few applications of arrays—the concrete data structures introduced in Section 1.1.3 that access their entries using integer indices.

3.1.1   Storing Game Entries in an Array

The first application we study is for storing entries in an array; in particular, high score entries for a video game. Storing objects in arrays is a common use for arrays, and we could just as easily have chosen to store records for patients in a hospital or the names of players on a football team. Nevertheless, let us focus on storing high score entries, which is a simple application that is already rich enough to present some important data structuring concepts.

Let us begin by thinking about what we want to include in an object representing a high score entry. Obviously, one component to include is an integer representing the score itself, which we call *score*. Another useful thing to include is the name of the person earning this score, which we simply call *name*. We could go on from here, adding fields representing the date the score was earned or game statistics that led to that score. Let us keep our example simple, however, and just have two fields, *score* and *name*. The class structure is shown in Code Fragment 3.1.

```
class GameEntry {                          // a game score entry
public:
  GameEntry(const string& n="", int s=0); // constructor
  string getName() const;                  // get player name
  int getScore() const;                    // get score
private:
  string name;                             // player's name
  int score;                               // player's score
};
```

Code Fragment 3.1: A C++ class representing a game entry.

In Code Fragment 3.2, we provide the definitions of the class constructor and two accessor member functions.

```
GameEntry::GameEntry(const string& n, int s) // constructor
  : name(n), score(s) { }
                                             // accessors
string GameEntry::getName() const { return name; }
int GameEntry::getScore() const { return score; }
```

Code Fragment 3.2: GameEntry constructor and accessors.

A Class for High Scores

Let's now design a class, called Scores, to store our game-score information. We store the highest scores in an array *entries*. The maximum number of scores may vary from instance to instance, so we create a member variable, *maxEntries*, storing the desired maximum. Its value is specified when a Scores object is first constructed. In order to keep track of the actual number of entries, we define a member variable *numEntries*. It is initialized to zero, and it is updated as entries are added or removed. We provide a constructor, a destructor, a member function for adding a new score, and one for removing a score at a given index. The definition is given in Code Fragment 3.3.

```
class Scores {                          // stores game high scores
public:
  Scores(int maxEnt = 10);              // constructor
  ~Scores();                            // destructor
  void add(const GameEntry& e);         // add a game entry
  GameEntry remove(int i)               // remove the ith entry
      throw(IndexOutOfBounds);
private:
  int maxEntries;                       // maximum number of entries
  int numEntries;                       // actual number of entries
  GameEntry* entries;                   // array of game entries
};
```

Code Fragment 3.3: A C++ class for storing high game scores.

In Code Fragment 3.4, we present the class constructor, which allocates the desired amount of storage for the array using the "new" operator. Recall from Section 1.1.3 that C++ represents a dynamic array as a pointer to its first element, and this command returns such a pointer. The class destructor, ~Scores, deletes this array.

```
Scores::Scores(int maxEnt) {           // constructor
  maxEntries = maxEnt;                  // save the max size
  entries = new GameEntry[maxEntries];  // allocate array storage
  numEntries = 0;                       // initially no elements
}

Scores::~Scores() {                    // destructor
  delete[] entries;
}
```

Code Fragment 3.4: A C++ class GameEntry representing a game entry.

The entries that have been added to the array are stored in indices 0 through *numEntries* − 1. As more users play our video game, additional GameEntry objects

are copied into the array. This is done using the class's add member function, which we describe below. Only the highest *maxEntries* scores are retained. We also provide a member function, remove(i), which removes the entry at index i from the array. We assume that $0 \leq i \leq numEntries - 1$. If not, the remove function, throws an *IndexOutOfBounds* exception. We do not define this exception here, but it is derived from the class RuntimeException from Section 2.4.

In our design, we have chosen to order the GameEntry objects by their *score* values, from highest to lowest. (In Exercise C-3.2, we explore an alternative design in which entries are not ordered.) We illustrate an example of the data structure in Figure 3.1.

Mike	Rob	Paul	Anna	Rose	Jack				
1105	750	720	660	590	510				
0	1	2	3	4	5	6	7	8	9

Figure 3.1: The *entries* array of length eight storing six GameEntry objects in the cells from index 0 to 5. Here *maxEntries* is 10 and *numEntries* is 6.

Insertion

Next, let us consider how to add a new GameEntry e to the array of high scores. In particular, let us consider how we might perform the following update operation on an instance of the Scores class.

> add(e): Insert game entry e into the collection of high scores. If this causes the number of entries to exceed *maxEntries*, the smallest is removed.

The approach is to shift all the entries of the array whose scores are smaller than e's score to the right, in order to make space for the new entry. (See Figure 3.2.)

Jill
740

Mike	Rob		Paul	Anna	Rose	Jack			
1105	750		720	660	590	510			
0	1	2	3	4	5	6	7	8	9
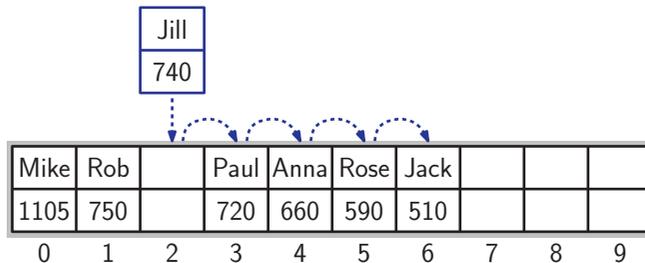
Figure 3.2: Preparing to add a new GameEntry object ("Jill", 740) to the *entries* array. In order to make room for the new entry, we shift all the entries with smaller scores to the right by one position.

Once we have identified the position in the *entries* array where the new game entry, *e*, belongs, we copy *e* into this position. (See Figure 3.3.)

Mike	Rob	Jill	Paul	Anna	Rose	Jack			
1105	750	740	720	660	590	510			

0   1   2   3   4   5   6   7   8   9

Figure 3.3: After adding the new entry at index 2.

The details of our algorithm for adding the new game entry *e* to the *entries* array are similar to this informal description and are given in Code Fragment 3.5. First, we consider whether the array is already full. If so, we check whether the score of the last entry in the array (which is at *entries*[*maxEntries* − 1]) is at least as large as *e*'s score. If so, we can return immediately since *e* is not high enough to replace any of the existing highest scores. If the array is not yet full, we know that one new entry will be added, so we increment the value of *numEntries*. Next, we identify all the entries whose scores are smaller than *e*'s and shift them one entry to the right. To avoid overwriting existing array entries, we start from the right end of the array and work to the left. The loop continues until we encounter an entry whose score is not smaller than *e*'s, or we fall off the front end of the array. In either case, the new entry is added at index *i* + 1.

```
void Scores::add(const GameEntry& e) {    // add a game entry
  int newScore = e.getScore();            // score to add
  if (numEntries == maxEntries) {         // the array is full
    if (newScore <= entries[maxEntries−1].getScore())
      return;                             // not high enough - ignore
  }
  else numEntries++;                      // if not full, one more entry

  int i = numEntries−2;                   // start with the next to last
  while ( i >= 0 && newScore > entries[i].getScore() ) {
    entries[i+1] = entries[i];            // shift right if smaller
    i−−;
  }
  entries[i+1] = e;                       // put e in the empty spot
}
```

Code Fragment 3.5: C++ code for inserting a GameEntry object.

Check the code carefully to see that all the limiting cases have been handled correctly by the add function (for example, largest score, smallest score, empty array, full array). The number of times we perform the loop in this function depends on the number of entries that we need to shift. This is pretty fast if the number of entries is small. But if there are a lot to move, then this method could be fairly slow.

Object Removal

Suppose some hot shot plays our video game and gets his or her name on our high score list. In this case, we might want to have a function that lets us remove a game entry from the list of high scores. Therefore, let us consider how we might remove a GameEntry object from the *entries* array. That is, let us consider how we might implement the following operation:

> remove(i): Remove and return the game entry e at index i in the *entries* array. If index i is outside the bounds of the *entries* array, then this function throws an exception; otherwise, the *entries* array is updated to remove the object at index i and all objects previously stored at indices higher than i are "shifted left" to fill in for the removed object.

Our implementation of remove is similar to that of add, but in reverse. To remove the entry at index i, we start at index i and move all the entries at indices higher than i one position to the left. (See Figure 3.4.)
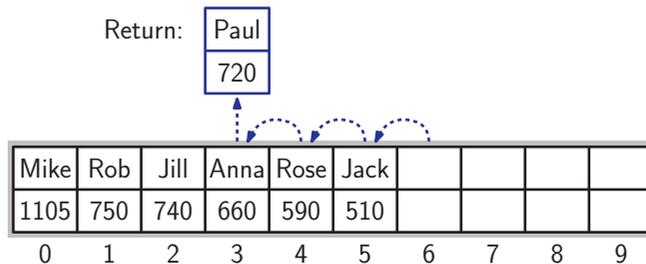
Figure 3.4: Removal of the entry ("Paul", 720) at index 3.

The code for performing the removal is presented in Code Fragment 3.6.

```cpp
GameEntry Scores::remove(int i) throw(IndexOutOfBounds) {
  if ((i < 0) || (i >= numEntries))        // invalid index
    throw IndexOutOfBounds("Invalid index");
  GameEntry e = entries[i];                // save the removed object
  for (int j = i+1; j < numEntries; j++)
    entries[j−1] = entries[j];             // shift entries left
  numEntries−−;                            // one fewer entry
  return e;                                // return the removed object
}
```

Code Fragment 3.6: C++ code for performing the remove operation.

The removal operation involves a few subtle points. In order to return the value of the removed game entry (let's call it e), we must first save e in a temporary variable. When we are done, the function will return this value. The shifting process starts at the position just following the removal, $j = i + 1$. We repeatedly copy the entry at index j to index $j - 1$, and then increment j, until coming to the last element of the set. Similar to the case of insertion, this left-to-right order is essential to avoid overwriting existing entries. To complete the function, we return a copy of the removed entry that was saved in e.

These functions for adding and removing objects in an array of high scores are simple. Nevertheless, they form the basis of techniques that are used repeatedly to build more sophisticated data structures. These other structures may be more general than our simple array-based solution, and they may support many more operations. But studying the concrete array data structure, as we are doing now, is a great starting point for understanding these more sophisticated structures, since every data structure has to be implemented using concrete means.

3.1.2 Sorting an Array

In the previous subsection, we worked hard to show how we can add or remove objects at a certain index i in an array while keeping the previous order of the objects intact. In this section, we consider how to rearrange objects of an array that are ordered arbitrarily in ascending order. This is known as ***sorting***.

We study several sorting algorithms in this book, most of which appear in Chapter 11. As a warmup, we describe a simple sorting algorithm called ***insertion-sort***. In this case, we describe a specific version of the algorithm where the input is an array of comparable elements. We consider more general kinds of sorting algorithms later in this book.

We begin with a high-level outline of the insertion-sort algorithm. We start with the first element in the array. One element by itself is already sorted. Then we consider the next element in the array. If it is smaller than the first, we swap them. Next we consider the third element in the array. We swap it leftward until it is in its proper order with the first two elements. We continue in this manner with each element of the array, swapping it leftward until it is in its proper position.

It is easy to see why this algorithm is called "insertion-sort"—each iteration of the algorithm inserts the next element into the current sorted part of the array, which was previously the subarray in front of that element. We may implement the above outline using two nested loops. The outer loop considers each element in the array in turn, and the inner loop moves that element to its proper location with the (sorted) subarray of elements that are to its left. We illustrate the resulting algorithm in Code Fragment 3.7.

This description is already quite close to actual C++ code. It indicates which

Algorithm InsertionSort(A):
    Input: An array A of n comparable elements
    Output: The array A with elements rearranged in nondecreasing order
    for $i \leftarrow 1$ to $n - 1$ **do**
        {Insert $A[i]$ at its proper location in $A[0], A[1], \ldots, A[i-1]$}
        $cur \leftarrow A[i]$
        $j \leftarrow i - 1$
        while $j \geq 0$ and $A[j] > cur$ **do**
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j - 1$
        $A[j+1] \leftarrow cur$ {cur is now in the right place}

Code Fragment 3.7: Algorithmic description of the insertion-sort algorithm.

temporary variables are needed, how the loops are structured, and what decisions need to be made. We illustrate an example run in Figure 3.5.

We present C++ code for our insertion-sort algorithm in Code Fragment 3.8. We assume that the array to be sorted consists of elements of type **char**, but it is easy to generalize this to other data types. The array A in the algorithm is implemented as a **char** array. Recall that each array in C++ is represented as a pointer to its first element, so the parameter A is declared to be of type **char***. We also pass the size of the array in an integer parameter n. The rest is a straightforward translation of the description given in Code Fragment 3.7 into C++ syntax.

```
void insertionSort(char* A, int n) {       // sort an array of n characters
  for (int i = 1; i < n; i++) {            // insertion loop
    char cur = A[i];                       // current character to insert
    int j = i − 1;                         // start at previous character
    while ((j >= 0) && (A[j] > cur)) {     // while A[j] is out of order
      A[j + 1] = A[j];                     // move A[j] right
      j−−;                                 // decrement j
    }
    A[j + 1] = cur;                        // this is the proper place for cur
  }
}
```

Code Fragment 3.8: C++ code implementing the insertion-sort algorithm.

An interesting thing happens in the insertion-sort algorithm if the array is already sorted. In this case, the inner loop does only one comparison, determines that there is no swap needed, and returns back to the outer loop. Of course, we might have to do a lot more work than this if the input array is extremely out of order. Indeed, the worst case arises if the initial array is given in descending order.
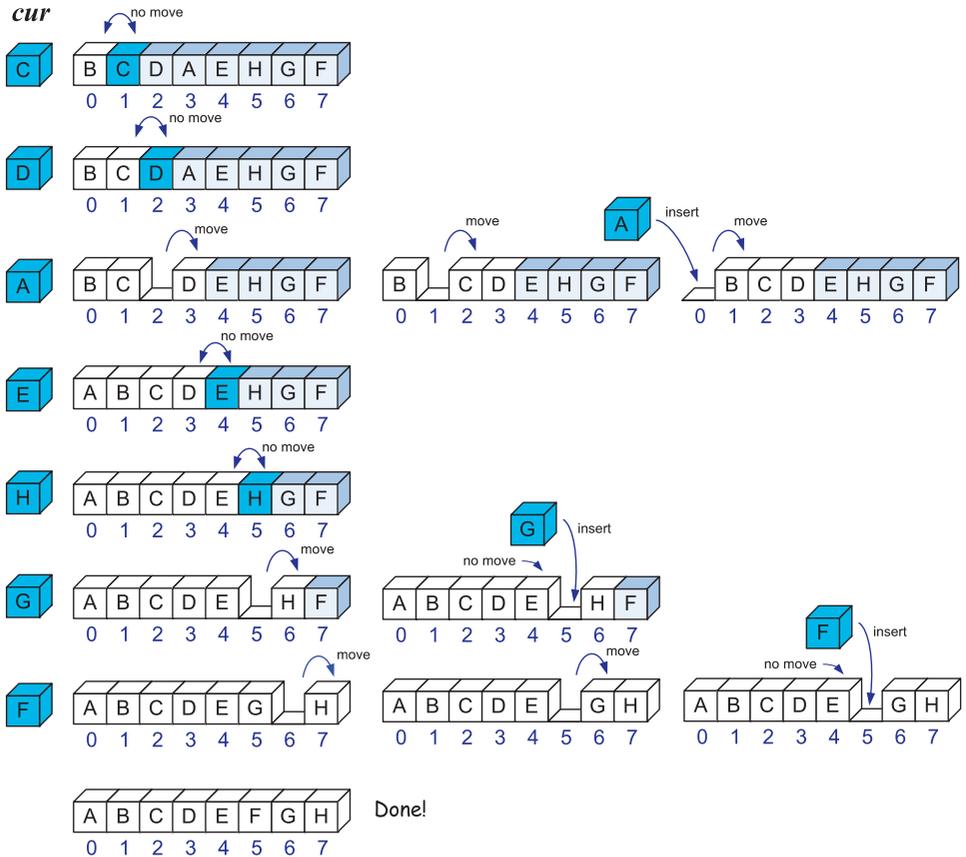
Figure 3.5: Execution of the insertion-sort algorithm on an array of eight characters. We show the completed (sorted) part of the array in white, and we color the next element that is being inserted into the sorted part of the array with light blue. We also highlight the character on the left, since it is stored in the *cur* variable. Each row corresponds to an iteration of the outer loop, and each copy of the array in a row corresponds to an iteration of the inner loop. Each comparison is shown with an arc. In addition, we indicate whether that comparison resulted in a move or not.

3.1.3 Two-Dimensional Arrays and Positional Games

Many computer games, be they strategy games, simulation games, or first-person conflict games, use a two-dimensional "board." Programs that deal with such *positional games* need a way of representing objects in a two-dimensional space. A natural way to do this is with a *two-dimensional array*, where we use two indices, say *i* and *j*, to refer to the cells in the array. The first index usually refers to a row number and the second to a column number. Given such an array we can then maintain two-dimensional game boards, as well as perform other kinds of computations

involving data that is stored in rows and columns.

Arrays in C++ are one-dimensional; we use a single index to access each cell of an array. Nevertheless, there is a way we can define two-dimensional arrays in C++—we can create a two-dimensional array as an array of arrays. That is, we can define a two-dimensional array to be an array with each of its cells being another array. Such a two-dimensional array is sometimes also called a ***matrix***. In C++, we declare a two-dimensional array as follows:

> **int** M[8][10];   // matrix with 8 rows and 10 columns

This statement creates a two-dimensional "array of arrays," M, which is 8×10, having 8 rows and 10 columns. That is, M is an array of length 8 such that each element of M is an array of length 10 of integers. (See Figure 3.6.)

	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

Figure 3.6: A two-dimensional integer array that has 8 rows and 10 columns. The value of $M[3][5]$ is 100 and the value of $M[6][2]$ is 632.

Given integer variables i and j, we could output the element of row i and column j (or equivalently, the jth element of the ith array) as follows:

> cout << M[i][j];   // output element in row i column j

It is often a good idea to use symbolic constants to define the dimensions in order to make your intentions clearer to someone reading your program.

> **const int** N_DAYS = 7;
> **const int** N_HOURS = 24;
> **int** schedule[N_DAYS][N_HOURS];

Dynamic Allocation of Matrices

If the dimensions of a two-dimensional array are not known in advance, it is necessary to allocate the array dynamically. This can be done by applying the method that we discussed earlier for allocating arrays in Section 1.1.3, but instead, we need to apply it to each individual row of the matrix.

For example, suppose that we wish to allocate an integer matrix with *n* rows and *m* columns. Each row of the matrix is an array of integers of length *m*. Recall that a dynamic array is represented as a pointer to its first element, so each row would be declared to be of type **int***. How do we group the individual rows together to form the matrix? The matrix is an array of row pointers. Since each row pointer is of type **int***, the matrix is of type **int****, that is, a pointer to a pointer of integers.

To generate our matrix, we first declare *M* to be of this type and allocate the *n* row pointers with the command "M = **new int***[n]." The *i*th row of the matrix is allocated with the statement "M[i] = **new int**[m]." In Code Fragment 3.9, we show how to do this given two integer variables *n* and *m*.

```
int** M = new int*[n];            // allocate an array of row pointers
for (int i = 0; i < n; i++)
  M[i] = new int[m];              // allocate the i-th row
```

Code Fragment 3.9: Allocating storage for a matrix as an array of arrays.

Once allocated, we can access its elements just as before, for example, as "*M*[*i*][*j*]." As shown in Code Fragment 3.10, deallocating the matrix involves reversing these steps. First, we deallocate each of the rows, one by one. We then deallocate the array of row pointers. Since we are deleting an array, we use the command "**delete[]**."

```
for (int i = 0; i < n; i++)
  delete[] M[i];                  // delete the i-th row
delete[] M;                       // delete the array of row pointers
```

Code Fragment 3.10: Deallocating storage for a matrix as an array of arrays.

Using STL Vectors to Implement Matrices

As we can see from the previous section, dynamic allocation of matrices is rather cumbersome. The STL vector class (recall Section 1.5.5) provides a much more elegant way to process matrices. We adapt the same approach as above by implementing a matrix as a vector of vectors. Each row of our matrix is declared as "vector<int>." Thus, the entire matrix is declared to be a vector of rows, that is, "vector<vector<int>>." Let us declare *M* to be of this type.

Letting *n* denote the desired number of rows in the matrix, the constructor call *M*(*n*) allocates storage for the rows. However, this does not allocate the desired number of columns. The reason is that the default constructor is called for each row, and the default is to construct an empty array.

To fix this, we make use of a nice feature of the vector class constructor. There is an optional second argument, which indicates the value to use when initializing

each element of the vector. In our case, each element of *M* is a vector of *m* integers, that is, "vector<int>(m)." Thus, given integer variables *n* and *m*, the following code fragment generates an $n \times m$ matrix as a vector of vectors.

```
vector< vector<int> > M(n, vector<int>(m));
cout << M[i][j] << endl;
```

The space between vector<int> and the following ">" has been added to prevent ambiguity with the C++ input operator ">>." Because the STL vector class automatically takes care of deleting its members, we do not need to write a loop to explicitly delete the rows, as we needed with dynamic arrays.

Two-dimensional arrays have many applications. Next, we explore a simple application of two-dimensional arrays for implementing a positional game.

Tic-Tac-Toe

As most school children know, ***Tic-Tac-Toe*** is a game played on a three-by-three board. Two players, X and O, alternate in placing their respective marks in the cells of this board, starting with player X. If either player succeeds in getting three of his or her marks in a row, column, or diagonal, then that player wins.

This is admittedly not a sophisticated positional game, and it's not even that much fun to play, since a good player O can always force a tie. Tic-Tac-Toe's saving grace is that it is a nice, simple example showing how two-dimensional arrays can be used for positional games. Software for more sophisticated positional games, such as checkers, chess, or the popular simulation games, are all based on the same approach we illustrate here for using a two-dimensional array for Tic-Tac-Toe. (See Exercise P-7.11.)

The basic idea is to use a two-dimensional array, *board*, to maintain the game board. Cells in this array store values that indicate if that cell is empty or stores an X or O. That is, *board* is a three-by-three matrix. For example, its middle row consists of the cells *board*[1][0], *board*[1][1], and *board*[1][2]. In our case, we choose to make the cells in the *board* array be integers, with a 0 indicating an empty cell, a 1 indicating an X, and a -1 indicating O. This encoding allows us to have a simple way of testing whether a given board configuration is a win for X or O, namely, if the values of a row, column, or diagonal add up to -3 or 3, respectively.

We give a complete C++ program for maintaining a Tic-Tac-Toe board for two players in Code Fragments 3.11 and 3.12. We show the resulting output in Figure 3.8. Note that this code is just for maintaining the Tic-Tac-Toe board and registering moves; it doesn't perform any strategy or allow someone to play Tic-Tac-Toe against the computer. The details of such a program are beyond the scope of this chapter, but it might nonetheless make a good project (see Exercise P-7.11).
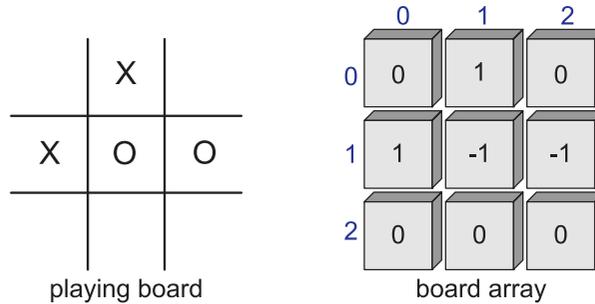
Figure 3.7: A Tic-Tac-Toe board and the array representing it.

```cpp
#include <cstdlib>                          // system definitions
#include <iostream>                         // I/O definitions
using namespace std;                        // make std:: accessible

const int X = 1, O = −1, EMPTY = 0;         // possible marks
int board[3][3];                            // playing board
int currentPlayer;                          // current player (X or O)

void clearBoard() {                         // clear the board
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      board[i][j] = EMPTY;                  // every cell is empty
  currentPlayer = X;                        // player X starts
}

void putMark(int i, int j) {                // mark row i column j
  board[i][j] = currentPlayer;              // mark with current player
  currentPlayer = −currentPlayer;           // switch players
}

bool isWin(int mark) {                              // is mark the winner?
  int win = 3*mark;                                 // +3 for X and -3 for O
  return ((board[0][0] + board[0][1] + board[0][2] == win)    // row 0
      || (board[1][0] + board[1][1] + board[1][2] == win)     // row 1
      || (board[2][0] + board[2][1] + board[2][2] == win)     // row 2
      || (board[0][0] + board[1][0] + board[2][0] == win)     // column 0
      || (board[0][1] + board[1][1] + board[2][1] == win)     // column 1
      || (board[0][2] + board[1][2] + board[2][2] == win)     // column 2
      || (board[0][0] + board[1][1] + board[2][2] == win)     // diagonal
      || (board[2][0] + board[1][1] + board[0][2] == win));    // diagonal
}
```

Code Fragment 3.11: A C++ program for playing Tic-Tac-Toe between two players. (Continues in Code Fragment 3.12.)

```cpp
int getWinner() {                          // who wins? (EMPTY means tie)
  if (isWin(X)) return X;
  else if (isWin(O)) return O;
  else return EMPTY;
}

void printBoard() {                        // print the board
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      switch (board[i][j]) {
      case X:     cout << "X"; break;
      case O:     cout << "O"; break;
      case EMPTY: cout << " "; break;
      }
      if (j < 2) cout << "|";              // column boundary
    }
    if (i < 2) cout << "\n-+-+-\n";        // row boundary
  }
}

int main() {                              // main program
  clearBoard();                           // clear the board
  putMark(0,0);      putMark(1,1);        // add the marks
  putMark(0,1);      putMark(0,2);
  putMark(2,0);      putMark(1,2);
  putMark(2,2);      putMark(2,1);
  putMark(1,0);
  printBoard();                           // print the final board
  int winner = getWinner();
  if (winner != EMPTY)                    // print the winner
    cout << " " << (winner == X ? 'X' : 'O') << " wins" << endl;
  else
    cout << " Tie" << endl;
  return EXIT_SUCCESS;
}
```

Code Fragment 3.12: A C++ program for playing Tic-Tac-Toe between two players. (Continued from Code Fragment 3.11.)

```
X|X|O
-+-+-
X|O|O
-+-+-
X|O|X   X wins
```

Figure 3.8: Output of the Tic-Tac-Toe program.

3.2 Singly Linked Lists

In the previous section, we presented the array data structure and discussed some of its applications. Arrays are nice and simple for storing things in a certain order, but they have drawbacks. They are not very adaptable. For instance, we have to fix the size *n* of an array in advance, which makes resizing an array difficult. (This drawback is remedied in STL vectors.) Insertions and deletions are difficult because elements need to be shifted around to make space for insertion or to fill empty positions after deletion. In this section, we explore an important alternate implementation of sequence, known as the singly linked list.

A ***linked list***, in its simplest form, is a collection of ***nodes*** that together form a linear ordering. As in the children's game "Follow the Leader," each node stores a pointer, called *next*, to the next node of the list. In addition, each node stores its associated element. (See Figure 3.9.)
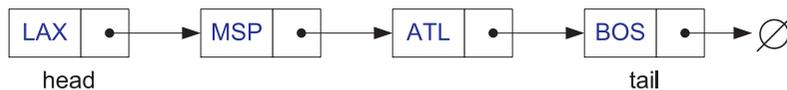
Figure 3.9: Example of a singly linked list of airport codes. The *next* pointers are shown as arrows. The null pointer is denoted by ∅.

The *next* pointer inside a node is a ***link*** or ***pointer*** to the next node of the list. Moving from one node to another by following a *next* reference is known as ***link hopping*** or ***pointer hopping***. The first and last nodes of a linked list are called the ***head*** and ***tail*** of the list, respectively. Thus, we can link-hop through the list, starting at the head and ending at the tail. We can identify the tail as the node having a null *next* reference. The structure is called a ***singly linked list*** because each node stores a single link.

Like an array, a singly linked list maintains its elements in a certain order, as determined by the chain of *next* links. Unlike an array, a singly linked list does not have a predetermined fixed size. It can be resized by adding or removing nodes.

3.2.1 Implementing a Singly Linked List

Let us implement a singly linked list of strings. We first define a class StringNode shown in Code Fragment 3.13. The node stores two values, the member *elem* stores the element stored in this node, which in this case is a character string. (Later, in Section 3.2.4, we describe how to define nodes that can store arbitrary types of elements.) The member *next* stores a pointer to the next node of the list. We make the linked list class a friend, so that it can access the node's private members.

```
class StringNode {                        // a node in a list of strings
private:
  string elem;                            // element value
  StringNode* next;                       // next item in the list

  friend class StringLinkedList;          // provide StringLinkedList access
};
```

Code Fragment 3.13: A node in a singly linked list of strings.

In Code Fragment 3.14, we define a class StringLinkedList for the actual linked list. It supports a number of member functions, including a constructor and destructor and functions for insertion and deletion. Their implementations are presented later. Its private data consists of a pointer to the head node of the list.

```
class StringLinkedList {                  // a linked list of strings
public:
  StringLinkedList();                     // empty list constructor
  ~StringLinkedList();                    // destructor
  bool empty() const;                     // is list empty?
  const string& front() const;            // get front element
  void addFront(const string& e);         // add to front of list
  void removeFront();                     // remove front item list
private:
  StringNode* head;                       // pointer to the head of list
};
```

Code Fragment 3.14: A class definition for a singly linked list of strings.

A number of simple member functions are shown in Code Fragment 3.15. The list constructor creates an empty list by setting the head pointer to NULL. The destructor repeatedly removes elements from the list. It exploits the fact that the function remove (presented below) destroys the node that it removes. To test whether the list is empty, we simply test whether the head pointer is NULL.

```
StringLinkedList::StringLinkedList()           // constructor
  : head(NULL) { }

StringLinkedList::~StringLinkedList()          // destructor
  { while (!empty()) removeFront(); }

bool StringLinkedList::empty() const           // is list empty?
  { return head == NULL; }

const string& StringLinkedList::front() const  // get front element
  { return head->elem; }
```

Code Fragment 3.15: Some simple member functions of class StringLinkedList.

3.2.2 Insertion to the Front of a Singly Linked List

We can easily insert an element at the head of a singly linked list. We first create a new node, and set its *elem* value to the desired string and set its *next* link to point to the current head of the list. We then set *head* to point to the new node. The process is illustrated in Figure 3.10.
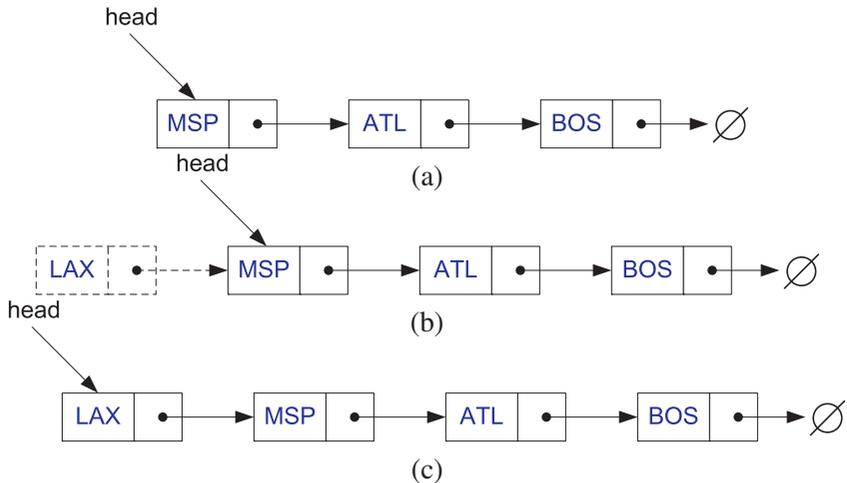
Figure 3.10: Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) creation of a new node; (c) after the insertion.

An implementation is shown in Code Fragment 3.16. Note that access to the private members *elem* and *next* of the StringNode class would normally be prohibited, but it is allowed here because StringLinkedList was declared to be a friend of StringNode.

```
void StringLinkedList::addFront(const string& e) {   // add to front of list
    StringNode* v = new StringNode;                  // create new node
    v−>elem = e;                                     // store data
    v−>next = head;                                  // head now follows v
    head = v;                                        // v is now the head
}
```

Code Fragment 3.16: Insertion to the front of a singly linked list.

3.2.3 Removal from the Front of a Singly Linked List

Next, we consider how to remove an element from the front of a singly linked list. We essentially undo the operations performed for insertion. We first save a pointer

to the old head node and advance the head pointer to the next node in the list. We then delete the old head node. This operation is illustrated in Figure 3.11.
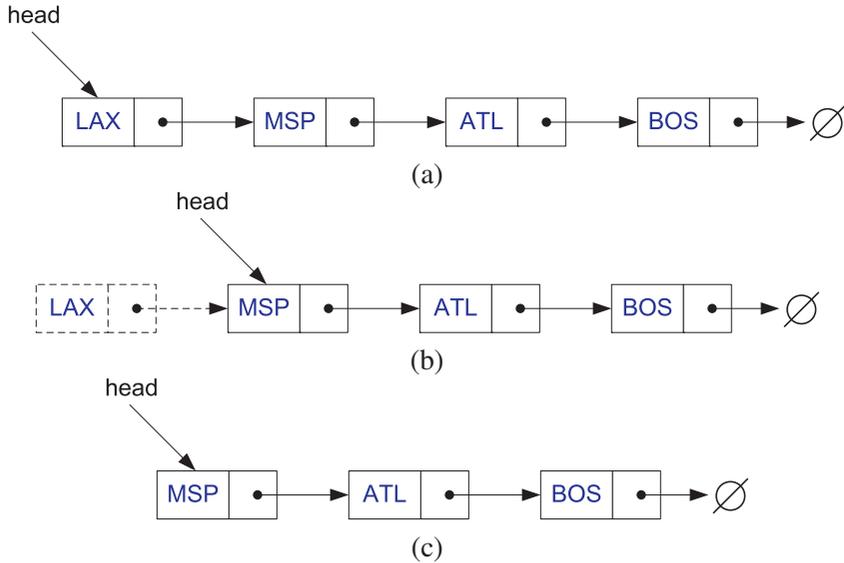
Figure 3.11: Removal of an element at the head of a singly linked list: (a) before the removal; (b) "linking out" the old new node; (c) after the removal.

An implementation of this operation is provided in Code Fragment 3.17. We assume that the user has checked that the list is nonempty before applying this operation. (A more careful implementation would throw an exception if the list were empty.) The function deletes the node in order to avoid any memory leaks. We do not return the value of the deleted node. If its value is desired, we can call the front function prior to the removal.

```
void StringLinkedList::removeFront() {          // remove front item
    StringNode* old = head;                     // save current head
    head = old->next;                           // skip over old head
    delete old;                                 // delete the old head
}
```

Code Fragment 3.17: Removal from the front of a singly linked list.

It is noteworthy that we cannot as easily delete the last node of a singly linked list, even if we had a pointer to it. In order to delete a node, we need to update the *next* link of the node immediately ***preceding*** the deleted node. Locating this node involves traversing the entire list and could take a long time. (We remedy this in Section 3.3 when we discuss doubly linked lists.)

3.2.4 Implementing a Generic Singly Linked List

The implementation of the singly linked list given in Section 3.2.1 assumes that the element type is a character string. It is easy to convert the implementation so that it works for an arbitrary element type through the use of C++'s template mechanism. The resulting generic singly linked list class is called SLinkedList.

We begin by presenting the node class, called SNode, in Code Fragment 3.18. The element type associated with each node is parameterized by the type variable E. In contrast to our earlier version in Code Fragment 3.13, references to the data type "string" have been replaced by "E." When referring to our templated node and list class, we need to include the suffix "<E>." For example, the class is SLinkedList<E> and the associated node is SNode<E>.

```
template <typename E>
class SNode {                              // singly linked list node
private:
  E elem;                                  // linked list element value
  SNode<E>* next;                          // next item in the list
  friend class SLinkedList<E>;             // provide SLinkedList access
};
```

Code Fragment 3.18: A node in a generic singly linked list.

The generic list class is presented in Code Fragment 3.19. As above, references to the specific element type "string" have been replaced by references to the generic type parameter "E." To keep things simple, we have omitted housekeeping functions such as a copy constructor.

```
template <typename E>
class SLinkedList {                        // a singly linked list
public:
  SLinkedList();                           // empty list constructor
  ~SLinkedList();                          // destructor
  bool empty() const;                      // is list empty?
  const E& front() const;                  // return front element
  void addFront(const E& e);               // add to front of list
  void removeFront();                      // remove front item list
private:
  SNode<E>* head;                          // head of the list
};
```

Code Fragment 3.19: A class definition for a generic singly linked list.

In Code Fragment 3.20, we present the class member functions. Note the similarity with Code Fragments 3.15 through 3.17. Observe that each definition is prefaced by the template specifier **template <typename E>**.

```
template <typename E>
SLinkedList<E>::SLinkedList()                    // constructor
  : head(NULL) { }

template <typename E>
bool SLinkedList<E>::empty() const               // is list empty?
  { return head == NULL; }

template <typename E>
const E& SLinkedList<E>::front() const            // return front element
  { return head->elem; }

template <typename E>
SLinkedList<E>::~SLinkedList()                    // destructor
  { while (!empty()) removeFront(); }

template <typename E>
void SLinkedList<E>::addFront(const E& e) {       // add to front of list
  SNode<E>* v = new SNode<E>;                     // create new node
  v->elem = e;                                    // store data
  v->next = head;                                 // head now follows v
  head = v;                                       // v is now the head
}

template <typename E>
void SLinkedList<E>::removeFront() {              // remove front item
  SNode<E>* old = head;                           // save current head
  head = old->next;                               // skip over old head
  delete old;                                     // delete the old head
}
```

Code Fragment 3.20: Other member functions for a generic singly linked list.

We can generate singly linked lists of various types by simply setting the template parameter as desired as shown in the following code fragment.

```
SLinkedList<string> a;                           // list of strings
a.addFront("MSP");
// ...
SLinkedList<int> b;                              // list of integers
b.addFront(13);
```

Code Fragment 3.21: Examples using the generic singly linked list class.

Because templated classes carry a relatively high notational burden, we often sacrifice generality for simplicity, and avoid the use of templated classes in some of our examples.

3.3 Doubly Linked Lists

As we saw in the previous section, removing an element at the tail of a singly linked list is not easy. Indeed, it is time consuming to remove any node other than the head in a singly linked list, since we do not have a quick way of accessing the node immediately preceding the one we want to remove. There are many applications where we do not have quick access to such a predecessor node. For such applications, it would be nice to have a way of going both directions in a linked list.

There is a type of linked list that allows us to go in both directions—forward and reverse—in a linked list. It is the *doubly linked* list. In addition to its element member, a node in a doubly linked list stores two pointers, a *next* link and a *prev* link, which point to the next node in the list and the previous node in the list, respectively. Such lists allow for a great variety of quick update operations, including efficient insertion and removal at any given position.

Header and Trailer Sentinels

To simplify programming, it is convenient to add special nodes at both ends of a doubly linked list: a *header* node just before the head of the list, and a *trailer* node just after the tail of the list. These "dummy" or *sentinel* nodes do not store any elements. They provide quick access to the first and last nodes of the list. In particular, the header's *next* pointer points to the first node of the list, and the *prev* pointer of the trailer node points to the last node of the list. An example is shown in Figure 3.12.
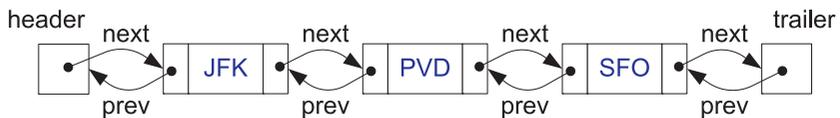
Figure 3.12: A doubly linked list with sentinels, *header* and *trailer*, marking the ends of the list. An empty list would have these sentinels pointing to each other. We do not show the null *prev* pointer for the *header* nor do we show the null *next* pointer for the *trailer*.

3.3.1 Insertion into a Doubly Linked List

Because of its double link structure, it is possible to insert a node at any position within a doubly linked list. Given a node *v* of a doubly linked list (which could possibly be the header, but not the trailer), let *z* be a new node that we wish to insert

immediately after *v*. Let *w* the be node following *v*, that is, *w* is the node pointed to by *v*'s next link. (This node exists, since we have sentinels.) To insert *z* after *v*, we link it into the current list, by performing the following operations:

- Make *z*'s *prev* link point to *v*
- Make *z*'s *next* link point to *w*
- Make *w*'s *prev* link point to *z*
- Make *v*'s *next* link point to *z*

This process is illustrated in Figure 3.13, where *v* points to the node JFK, *w* points to PVD, and *z* points to the new node BWI. Observe that this process works if *v* is any node ranging from the header to the node just prior to the trailer.
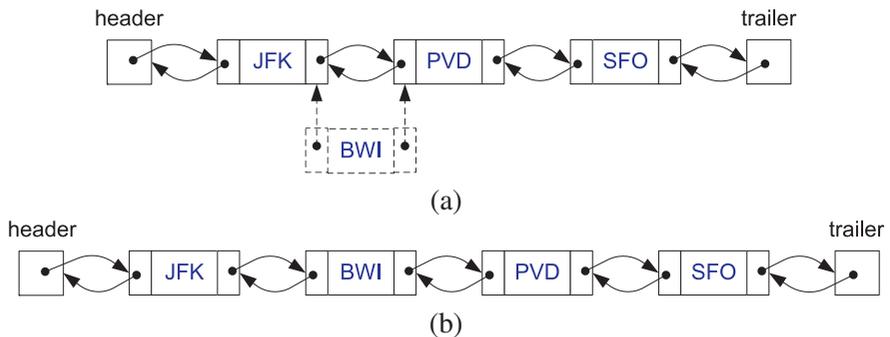
Figure 3.13: Adding a new node after the node storing JFK: (a) creating a new node with element BWI and linking it in; (b) after the insertion.

3.3.2 Removal from a Doubly Linked List

Likewise, it is easy to remove a node *v* from a doubly linked list. Let *u* be the node just prior to *v*, and *w* be the node just following *v*. (These nodes exist, since we have sentinels.) To remove node *v*, we simply have *u* and *w* point to each other instead of to *v*. We refer to this operation as the ***linking out*** of *v*. We perform the following operations.

- Make *w*'s *prev* link point to *u*
- Make *u*'s *next* link point to *w*
- Delete node *v*

This process is illustrated in Figure 3.14, where *v* is the node PVD, *u* is the node JFK, and *w* is the node SFO. Observe that this process works if *v* is any node from the header to the tail node (the node just prior to the trailer).
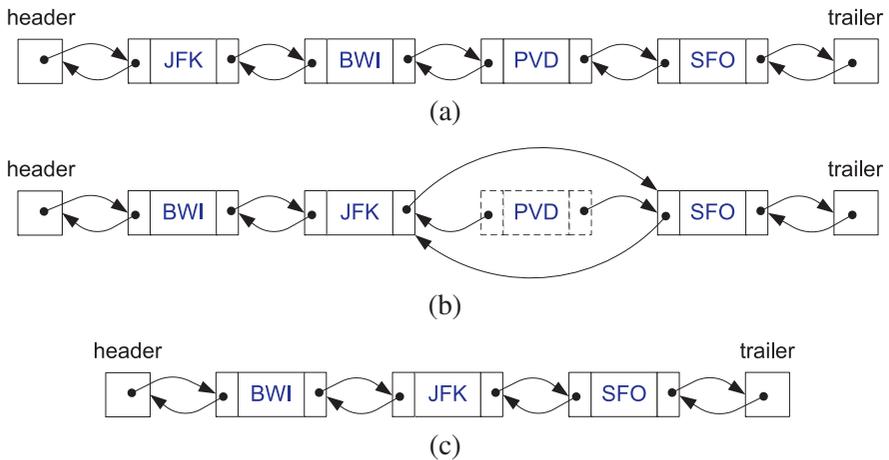
Figure 3.14: Removing the node storing PVD: (a) before the removal; (b) linking out the old node; (c) after node deletion.

3.3.3   A C++ Implementation

Let us consider how to implement a doubly linked list in C++. First, we present a C++ class for a node of the list in Code Fragment 3.22. To keep the code simple, we have chosen not to derive a templated class as we did in Section 3.2.1 for singly linked lists. Instead, we provide a **typedef** statement that defines the element type, called Elem. We define it to be a string, but any other type could be used instead. Each node stores an element. It also contains pointers to both the previous and next nodes of the list. We declare DLinkedList to be a friend, so it can access the node's private members.

```
typedef string Elem;                    // list element type
class DNode {                           // doubly linked list node
private:
  Elem elem;                            // node element value
  DNode* prev;                          // previous node in list
  DNode* next;                          // next node in list
  friend class DLinkedList;             // allow DLinkedList access
};
```

Code Fragment 3.22: C++ implementation of a doubly linked list node.

Next, we present the definition of the doubly linked list class, DLinkedList, in Code Fragment 3.23. In addition to a constructor and destructor, the public members consist of a function that indicates whether the list is currently empty

(meaning that it has no nodes other than the sentinels) and accessors to retrieve the front and back elements. We also provide methods for inserting and removing elements from the front and back of the list. There are two private data members, *header* and *trailer*, which point to the sentinels. Finally, we provide two protected utility member functions, add and remove. They are used internally by the class and by its subclasses, but they cannot be invoked from outside the class.

```cpp
class DLinkedList {                          // doubly linked list
public:
  DLinkedList();                             // constructor
  ~DLinkedList();                            // destructor
  bool empty() const;                        // is list empty?
  const Elem& front() const;                 // get front element
  const Elem& back() const;                  // get back element
  void addFront(const Elem& e);              // add to front of list
  void addBack(const Elem& e);               // add to back of list
  void removeFront();                        // remove from front
  void removeBack();                         // remove from back
private:                                     // local type definitions
  DNode* header;                             // list sentinels
  DNode* trailer;
protected:                                   // local utilities
  void add(DNode* v, const Elem& e);         // insert new node before v
  void remove(DNode* v);                     // remove node v
};
```

Code Fragment 3.23: Implementation of a doubly linked list class.

Let us begin by presenting the class constructor and destructor as shown in Code Fragment 3.24. The constructor creates the sentinel nodes and sets each to point to the other, and the destructor removes all but the sentinel nodes.

```cpp
DLinkedList::DLinkedList() {                 // constructor
  header = new DNode;                        // create sentinels
  trailer = new DNode;
  header->next = trailer;                    // have them point to each other
  trailer->prev = header;
}

DLinkedList::~DLinkedList() {                // destructor
  while (!empty()) removeFront();            // remove all but sentinels
  delete header;                             // remove the sentinels
  delete trailer;
}
```

Code Fragment 3.24: Class constructor and destructor.

Next, in Code Fragment 3.25 we show the basic class accessors. To determine whether the list is empty, we check that there is no node between the two sentinels. We do this by testing whether the trailer follows immediately after the header. To access the front element of the list, we return the element associated with the node that follows the list header. To access the back element, we return the element associated with node that precedes the trailer. Both operations assume that the list is nonempty. We could have enhanced these functions by throwing an exception if an attempt is made to access the front or back of an empty list, just as we did in Code Fragment 3.6.

```
bool DLinkedList::empty() const          // is list empty?
  { return (header->next == trailer); }

const Elem& DLinkedList::front() const    // get front element
  { return header->next->elem; }

const Elem& DLinkedList::back() const     // get back element
  { return trailer->prev->elem; }
```

Code Fragment 3.25: Accessor functions for the doubly linked list class.

In Section 3.3.1, we discussed how to insert a node into a doubly linked list. The local utility function add, which is shown in Code Fragment 3.26, implements this operation. In order to add a node to the front of the list, we create a new node, and insert it immediately after the header, or equivalently, immediately before the node that follows the header. In order to add a new node to the back of the list, we create a new node, and insert it immediately before the trailer.

```
                                         // insert new node before v
void DLinkedList::add(DNode* v, const Elem& e) {
  DNode* u = new DNode; u->elem = e;     // create a new node for e
  u->next = v;                           // link u in between v
  u->prev = v->prev;                     // ...and v->prev
  v->prev->next = v->prev = u;
}

void DLinkedList::addFront(const Elem& e) // add to front of list
  { add(header->next, e); }

void DLinkedList::addBack(const Elem& e)  // add to back of list
  { add(trailer, e); }
```

Code Fragment 3.26: Inserting a new node into a doubly linked list. The protected utility function add inserts a node z before an arbitrary node v. The public member functions addFront and addBack both invoke this utility function.

Observe that the above code works even if the list is empty (meaning that the only nodes are the header and trailer). For example, if addBack is invoked on an empty list, then the value of *trailer->prev* is a pointer to the list header. Thus, the node is added between the header and trailer as desired. One of the major advantages of providing sentinel nodes is to avoid handling of special cases, which would otherwise be needed.

Finally, let us discuss deletion. In Section 3.3.2, we showed how to remove an arbitrary node from a doubly linked list. In Code Fragment 3.27, we present local utility function remove, which performs the operation. In addition to linking out the node, it also deletes the node. The public member functions removeFront and removeBack are implemented by deleting the nodes immediately following the header and immediately preceding the trailer, respectively.

```
void DLinkedList::remove(DNode* v) {       // remove node v
  DNode* u = v->prev;                      // predecessor
  DNode* w = v->next;                      // successor
  u->next = w;                             // unlink v from list
  w->prev = u;
  delete v;
}

void DLinkedList::removeFront()            // remove from font
  { remove(header->next); }

void DLinkedList::removeBack()             // remove from back
  { remove(trailer->prev); }
```

Code Fragment 3.27: Removing a node from a doubly linked list. The local utility function remove removes the node *v*. The public member functions removeFront and removeBack invoke this utility function.

There are many more features that we could have added to our simple implementation of a doubly linked list. Although we have provided access to the ends of the list, we have not provided any mechanism for accessing or modifying elements in the middle of the list. Later, in Chapter 6, we discuss the concept of iterators, which provides a mechanism for accessing arbitrary elements of a list.

We have also performed no error checking in our implementation. It is the user's responsibility not to attempt to access or remove elements from an empty list. In a more robust implementation of a doubly linked list, we would design the member functions front, back, removeFront, and removeBack to throw an exception when an attempt is made to perform one of these functions on an empty list. Nonetheless, this simple implementation illustrates how easy it is to manipulate this useful data structure.

# 3.4 Circularly Linked Lists and List Reversal

In this section, we study some applications and extensions of linked lists.

3.4.1 Circularly Linked Lists

A *circularly linked list* has the same kind of nodes as a singly linked list. That is, each node in a circularly linked list has a next pointer and an element value. But, rather than having a head or tail, the nodes of a circularly linked list are linked into a cycle. If we traverse the nodes of a circularly linked list from any node by following **next** pointers, we eventually visit all the nodes and cycle back to the node from which we started.

Even though a circularly linked list has no beginning or end, we nevertheless need some node to be marked as a special node, which we call the *cursor*. The cursor node allows us to have a place to start from if we ever need to traverse a circularly linked list.

There are two positions of particular interest in a circular list. The first is the element that is referenced by the cursor, which is called the *back*, and the element immediately following this in the circular order, which is called the *front*. Although it may seem odd to think of a circular list as having a front and a back, observe that, if we were to cut the link between the node referenced by the cursor and this node's immediate successor, the result would be a singly linked list from the front node to the back node.
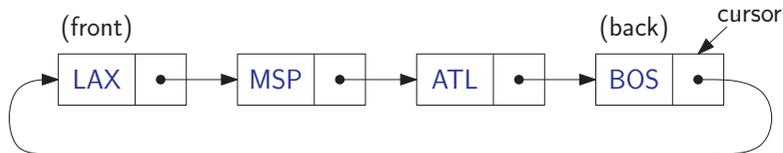
Figure 3.15: A circularly linked list. The node referenced by the cursor is called the back, and the node immediately following is called the front.

We define the following functions for a circularly linked list:

front(): Return the element referenced by the cursor; an error results if the list is empty.

back(): Return the element immediately after the cursor; an error results if the list is empty.

advance(): Advance the cursor to the next node in the list.

add(*e*): Insert a new node with element *e* immediately after the cursor; if the list is empty, then this node becomes the cursor and its *next* pointer points to itself.

remove(): Remove the node immediately after the cursor (not the cursor itself, unless it is the only node); if the list becomes empty, the cursor is set to *null*.

In Code Fragment 3.28, we show a C++ implementation of a node of a circularly linked list, assuming that each node contains a single string. The node structure is essentially identical to that of a singly linked list (recall Code Fragment 3.13). To keep the code simple, we have not implemented a templated class. Instead, we provide a **typedef** statement that defines the element type Elem to be the base type of the list, which in this case is a string.

```
typedef string Elem;                    // element type
class CNode {                           // circularly linked list node
private:
  Elem elem;                            // linked list element value
  CNode* next;                          // next item in the list

  friend class CircleList;              // provide CircleList access
};
```

Code Fragment 3.28: A node of a circularly linked list.

Next, in Code Fragment 3.29, we present the class definition for a circularly linked list called CircleList. In addition to the above functions, the class provides a constructor, a destructor, and a function to detect whether the list is empty. The private member consists of the cursor, which points to some node of the list.

```
class CircleList {                      // a circularly linked list
public:
  CircleList();                         // constructor
  ~CircleList();                        // destructor
  bool empty() const;                   // is list empty?
  const Elem& front() const;            // element at cursor
  const Elem& back() const;             // element following cursor
  void advance();                       // advance cursor
  void add(const Elem& e);              // add after cursor
  void remove();                        // remove node after cursor
private:
  CNode* cursor;                        // the cursor
};
```

Code Fragment 3.29: Implementation of a circularly linked list class.

Code Fragment 3.30 presents the class's constructor and destructor. The constructor generates an empty list by setting the cursor to NULL. The destructor iteratively removes nodes until the list is empty. We exploit the fact that the member function remove (given below) deletes the node that it removes.

```
CircleList::CircleList()                    // constructor
  : cursor(NULL) { }
CircleList::~CircleList()                    // destructor
  { while (!empty()) remove(); }
```

Code Fragment 3.30: The constructor and destructor.

We present a number of simple member functions in Code Fragment 3.31. To determine whether the list is empty, we test whether the cursor is NULL. The advance function advances the cursor to the next element.

```
bool CircleList::empty() const              // is list empty?
  { return cursor == NULL; }
const Elem& CircleList::back() const        // element at cursor
  { return cursor->elem; }
const Elem& CircleList::front() const       // element following cursor
  { return cursor->next->elem; }
void CircleList::advance()                   // advance cursor
  { cursor = cursor->next; }
```

Code Fragment 3.31: Simple member functions.

Next, let us consider insertion. Recall that insertions to the circularly linked list occur *after* the cursor. We begin by creating a new node and initializing its data member. If the list is empty, we create a new node that points to itself. We then direct the cursor to point to this element. Otherwise, we link the new node just after the cursor. The code is presented in Code Fragment 3.32.

```
void CircleList::add(const Elem& e) {        // add after cursor
  CNode* v = new CNode;                      // create a new node
  v->elem = e;
  if (cursor == NULL) {                      // list is empty?
    v->next = v;                             // v points to itself
    cursor = v;                              // cursor points to v
  }
  else {                                     // list is nonempty?
    v->next = cursor->next;                  // link in v after cursor
    cursor->next = v;
  }
}
```

Code Fragment 3.32: Inserting a node just after the cursor of a circularly linked list.

Finally, we consider removal. We assume that the user has checked that the list is nonempty before invoking this function. (A more careful implementation would throw an exception if the list is empty.) There are two cases. If this is the last node of the list (which can be tested by checking that the node to be removed points to itself) we set the cursor to NULL. Otherwise, we link the cursor's next pointer to skip over the removed node. We then delete the node. The code is presented in Code Fragment 3.33.

```
void CircleList::remove() {          // remove node after cursor
  CNode* old = cursor−>next;         // the node being removed
  if (old == cursor)                 // removing the only node?
    cursor = NULL;                   // list is now empty
  else
    cursor−>next = old−>next;        // link out the old node
  delete old;                        // delete the old node
}
```

Code Fragment 3.33: Removing the node following the cursor.

To keep the code simple, we have omitted error checking. In front, back, and advance, we should first test whether the list is empty, since otherwise the cursor pointer will be NULL. In the first two cases, we should throw some sort of exception. In the case of advance, if the list is empty, we can simply return.

Maintaining a Playlist for a Digital Audio Player

To help illustrate the use of our CircleList implementation of the circularly linked list, let us consider how to build a simple interface for maintaining a playlist for a digital audio player, also known as an MP3 player. The songs of the player are stored in a circular list. The cursor points to the current song. By advancing the cursor, we can move from one song to the next. We can also add new songs and remove songs by invoking the member functions insert and remove, respectively. Of course, a complete implementation would need to provide a method for playing the current song, but our purpose is to illustrate how the circularly linked list can be applied to this task.

To make this more concrete, suppose that you have a friend who loves retro music, and you want to create a playlist of songs from the bygone Disco Era. The main program is presented Code Fragment 3.34. We declare an object *playList* to be a CircleList. The constructor creates an empty playlist. We proceed to add three songs, "Stayin Alive," "Le Freak," and "Jive Talkin." The comments on the right show the current contents of the list in square brackets. The first entry of the list is the element immediately following the cursor (which is where insertion and removal occur), and the last entry in the list is cursor (which is indicated with an asterisk).

Suppose that we decide to replace "Stayin Alive" with "Disco Inferno." We advance the cursor twice so that "Stayin Alive" comes immediately after the cursor. We then remove this entry and insert its replacement.

```cpp
int main() {
  CircleList playList;                // []
  playList.add("Stayin Alive");       // [Stayin Alive*]
  playList.add("Le Freak");           // [Le Freak, Stayin Alive*]
  playList.add("Jive Talkin");        // [Jive Talkin, Le Freak, Stayin Alive*]

  playList.advance();                 // [Le Freak, Stayin Alive, Jive Talkin*]
  playList.advance();                 // [Stayin Alive, Jive Talkin, Le Freak*]
  playList.remove();                  // [Jive Talkin, Le Freak*]
  playList.add("Disco Inferno");      // [Disco Inferno, Jive Talkin, Le Freak*]
  return EXIT_SUCCESS;
}
```

Code Fragment 3.34: Using the CircleList class to implement a playlist for a digital audio player.

3.4.2 Reversing a Linked List

As another example of the manipulation of linked lists, we present a simple function for reversing the elements of a doubly linked list. Given a list L, our approach involves first copying the contents of L in reverse order into a temporary list T, and then copying the contents of T back into L (but without reversing).

To achieve the initial reversed copy, we repeatedly extract the first element of L and copy it to the front of T. (To see why this works, observe that the later an element appears in L, the earlier it will appear in T.) To copy the contents of T back to L, we repeatedly extract elements from the front of T, but this time we copy each one to the back of list L. Our C++ implementation is presented in Code Fragment 3.35.

```cpp
void listReverse(DLinkedList& L) {       // reverse a list
  DLinkedList T;                          // temporary list
  while (!L.empty()) {                    // reverse L into T
    string s = L.front();   L.removeFront();
    T.addFront(s);
  }
  while (!T.empty()) {                     // copy T back to L
    string s = T.front();   T.removeFront();
    L.addBack(s);
  }
}
```

Code Fragment 3.35: A function that reverses the contents of a doubly linked list L.

3.5   Recursion

We have seen that repetition can be achieved by writing loops, such as **for** loops and **while** loops. Another way to achieve repetition is through *recursion*, which occurs when a function refers to itself in its own definition. We have seen examples of functions calling other functions, so it should come as no surprise that most modern programming languages, including C++, allow a function to call itself. In this section, we see why this capability provides an elegant and powerful alternative for performing repetitive tasks.

The Factorial Function

To illustrate recursion, let us begin with a simple example of computing the value of the *factorial function*. The factorial of a positive integer n, denoted $n!$, is defined as the product of the integers from 1 to n. If $n = 0$, then $n!$ is defined as 1 by convention. More formally, for any integer $n \geq 0$,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. To make the connection with functions clearer, we use the notation $\text{factorial}(n)$ to denote $n!$.

The factorial function can be defined in a manner that suggests a recursive formulation. To see this, observe that

$$\text{factorial}(5) = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot \text{factorial}(4).$$

Thus, we can define $\text{factorial}(5)$ in terms of $\text{factorial}(4)$. In general, for a positive integer n, we can define $\text{factorial}(n)$ to be $n \cdot \text{factorial}(n-1)$. This leads to the following *recursive definition*

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{factorial}(n-1) & \text{if } n \geq 1. \end{cases}$$

This definition is typical of many recursive definitions. First, it contains one or more *base cases*, which are defined nonrecursively in terms of fixed quantities. In this case, $n = 0$ is the base case. It also contains one or more *recursive cases*, which are defined by appealing to the definition of the function being defined. Observe that there is no circularity in this definition because each time the function is invoked, its argument is smaller by one.

A Recursive Implementation of the Factorial Function

Let us consider a C++ implementation of the factorial function shown in Code Fragment 3.36 under the name recursiveFactorial. Notice that no looping was needed here. The repeated recursive invocations of the function take the place of looping.

```cpp
int recursiveFactorial(int n) {              // recursive factorial function
    if (n == 0) return 1;                    // basis case
    else return n * recursiveFactorial(n−1); // recursive case
}
```

Code Fragment 3.36: A recursive implementation of the factorial function.

We can illustrate the execution of a recursive function definition by means of a *recursion trace*. Each entry of the trace corresponds to a recursive call. Each new recursive function call is indicated by an arrow to the newly called function. When the function returns, an arrow showing this return is drawn, and the return value may be indicated with this arrow. An example of a trace is shown in Figure 3.16.

What is the advantage of using recursion? Although the recursive implementation of the factorial function is somewhat simpler than the iterative version, in this case there is no compelling reason for preferring recursion over iteration. For some problems, however, a recursive implementation can be significantly simpler and easier to understand than an iterative implementation. Such an example follows.
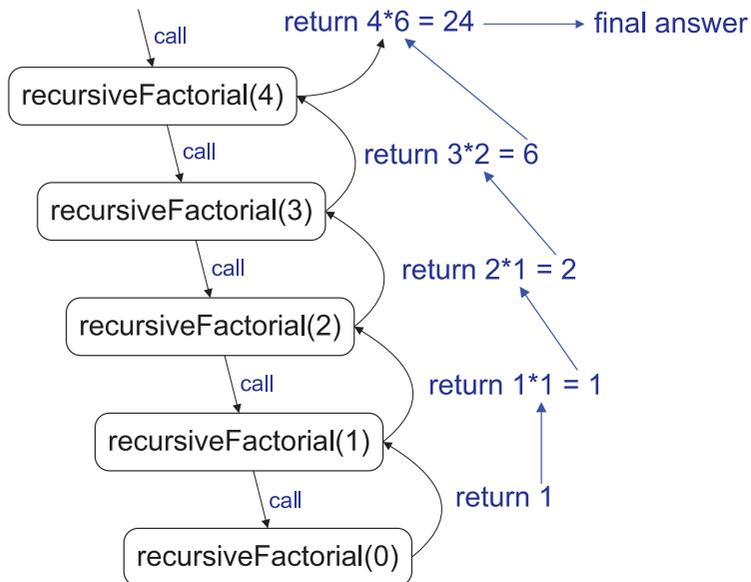
Figure 3.16: A recursion trace for the call recursiveFactorial(4).

Drawing an English Ruler

As a more complex example of the use of recursion, consider how to draw the markings of a typical English ruler. Such a ruler is broken into intervals, and each interval consists of a set of *ticks*, placed at intervals of 1/2 inch, 1/4 inch, and so on. As the size of the interval decreases by half, the tick length decreases by one. (See Figure 3.17.)

```
---- 0              ----- 0              --- 0
-                   -                    -
--                  --                   --
-                   -                    -
---                 ---                  --- 1
-                   -                    -
--                  --                   --
-                   -                    -
---- 1              ----                 --- 2
-                   -                    -
--                  --                   --
-                   -                    -
---                 ---                  --- 3
-                   -                    -
--                  --                   --
-                   -                    -
---- 2              ----- 1

(a)                 (b)                  (c)
```
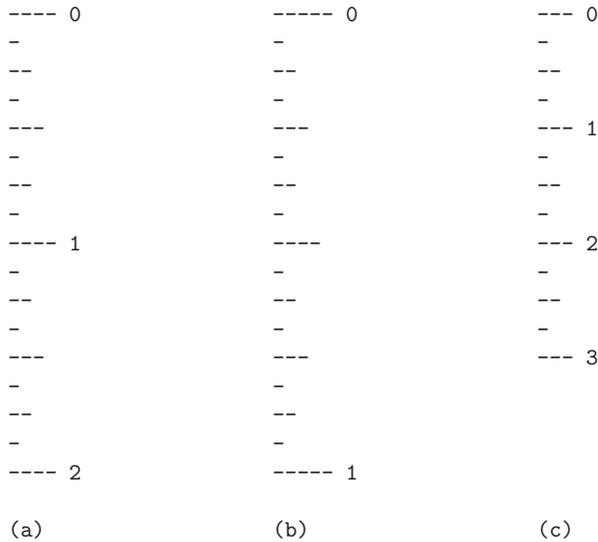
Figure 3.17: Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.

Each fraction of an inch also has a numeric label. The longest tick length is called the ***major tick length***. We won't worry about actual distances, however, and just print one tick per line.

A Recursive Approach to Ruler Drawing

Our approach to drawing such a ruler consists of three functions. The main function drawRuler draws the entire ruler. Its arguments are the total number of inches in the ruler, *nInches*, and the major tick length, *majorLength*. The utility function drawOneTick draws a single tick of the given length. It can also be given an optional integer label, which is printed if it is nonnegative.

The interesting work is done by the recursive function drawTicks, which draws the sequence of ticks within some interval. Its only argument is the tick length associated with the interval's central tick. Consider the English ruler with major tick length 5 shown in Figure 3.17(b). Ignoring the lines containing 0 and 1, let us consider how to draw the sequence of ticks lying between these lines. The central tick (at 1/2 inch) has length 4. Observe that the two patterns of ticks above and below this central tick are identical, and each has a central tick of length 3. In general, an interval with a central tick length $L \geq 1$ is composed of the following:

- An interval with a central tick length $L - 1$

- A single tick of length L

- An interval with a central tick length $L - 1$

With each recursive call, the length decreases by one. When the length drops to zero, we simply return. As a result, this recursive process always terminates. This suggests a recursive process in which the first and last steps are performed by calling the drawTicks$(L - 1)$ recursively. The middle step is performed by calling the function drawOneTick(L). This recursive formulation is shown in Code Fragment 3.37. As in the factorial example, the code has a base case (when $L = 0$). In this instance we make two recursive calls to the function.

```cpp
                                          // one tick with optional label
void drawOneTick(int tickLength, int tickLabel = −1) {
  for (int i = 0; i < tickLength; i++)
    cout << "-";
  if (tickLabel >= 0) cout << " " << tickLabel;
  cout << "\n";
}
void drawTicks(int tickLength) {            // draw ticks of given length
  if (tickLength > 0) {                     // stop when length drops to 0
    drawTicks(tickLength−1);                // recursively draw left ticks
    drawOneTick(tickLength);                // draw center tick
    drawTicks(tickLength−1);                // recursively draw right ticks
  }
}
void drawRuler(int nInches, int majorLength) {// draw the entire ruler
  drawOneTick(majorLength, 0);              // draw tick 0 and its label
  for (int i = 1; i <= nInches; i++) {
    drawTicks(majorLength−1);              // draw ticks for this inch
    drawOneTick(majorLength, i);            // draw tick i and its label
  }
}
```

Code Fragment 3.37: A recursive implementation of a function that draws a ruler.

Illustrating Ruler Drawing using a Recursion Trace

The recursive execution of the recursive drawTicks function, defined above, can be visualized using a recursion trace.

The trace for drawTicks is more complicated than in the factorial example, however, because each instance makes two recursive calls. To illustrate this, we show the recursion trace in a form that is reminiscent of an outline for a document. See Figure 3.18.

Figure 3.18: A partial recursion trace for the call drawTicks(3). The second pattern of calls for drawTicks(2) is not shown, but it is identical to the first.

Throughout this book, we see many other examples of how recursion can be used in the design of data structures and algorithms.

Further Illustrations of Recursion

As we discussed above, **recursion** is the concept of defining a function that makes a call to itself. When a function calls itself, we refer to this as a **recursive** call. We also consider a function M to be recursive if it calls another function that ultimately leads to a call back to M.

The main benefit of a recursive approach to algorithm design is that it allows us to take advantage of the repetitive structure present in many problems. By making our algorithm description exploit this repetitive structure in a recursive way, we can often avoid complex case analyses and nested loops. This approach can lead to more readable algorithm descriptions, while still being quite efficient.

In addition, recursion is a useful way for defining objects that have a repeated similar structural form, such as in the following examples.

Example 3.1: *Modern operating systems define file-system directories (which are also sometimes called "folders") in a recursive way. Namely, a file system consists of a top-level directory, and the contents of this directory consists of files and other directories, which in turn can contain files and other directories, and so on. The base directories in the file system contain only files, but by using this recursive definition, the operating system allows for directories to be nested arbitrarily deep (as long as there is enough space in memory).*

Example 3.2: *Much of the syntax in modern programming languages is defined in a recursive way. For example, we can define an argument list in C++ using the following notation:*

> *argument-list:*
>> *argument*
>> *argument-list , argument*

In other words, an argument list consists of either (i) an argument or (ii) an argument list followed by a comma and an argument. That is, an argument list consists of a comma-separated list of arguments. Similarly, arithmetic expressions can be defined recursively in terms of primitives (like variables and constants) and arithmetic expressions.

Example 3.3: *There are many examples of recursion in art and nature. One of the most classic examples of recursion used in art is in the Russian Matryoshka dolls. Each doll is made of solid wood or is hollow and contains another Matryoshka doll inside it.*

3.5.1   Linear Recursion

The simplest form of recursion is *linear recursion*, where a function is defined so that it makes at most one recursive call each time it is invoked. This type of recursion is useful when we view an algorithmic problem in terms of a first or last element plus a remaining set that has the same structure as the original set.

Summing the Elements of an Array Recursively

Suppose, for example, we are given an array, A, of n integers that we want to sum together. We can solve this summation problem using linear recursion by observing that the sum of all n integers in A is equal to $A[0]$, if $n = 1$, or the sum of the first $n - 1$ integers in A plus the last element in A. In particular, we can solve this summation problem using the recursive algorithm described in Code Fragment 3.38.

Algorithm LinearSum(A, n):
    Input: A integer array A and an integer $n \geq 1$, such that A has at least n elements
    Output: The sum of the first n integers in A
  if $n = 1$ **then**
    return $A[0]$
  else
    return LinearSum($A, n - 1$) $+ A[n - 1]$

Code Fragment 3.38: Summing the elements in an array using linear recursion.

This example also illustrates an important property that a recursive function should always possess—the function terminates. We ensure this by writing a non-recursive statement for the case $n = 1$. In addition, we always perform the recursive call on a smaller value of the parameter ($n - 1$) than that which we are given (n), so that, at some point (at the "bottom" of the recursion), we will perform the nonrecursive part of the computation (returning $A[0]$). In general, an algorithm that uses linear recursion typically has the following form:

- ***Test for base cases.*** We begin by testing for a set of base cases (there should be at least one). These base cases should be defined so that every possible chain of recursive calls eventually reaches a base case, and the handling of each base case should not use recursion.

- ***Recur.*** After testing for base cases, we then perform a single recursive call. This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step. Moreover, we should define each possible recursive call so that it makes progress towards a base case.

Analyzing Recursive Algorithms using Recursion Traces

We can analyze a recursive algorithm by using a visual tool known as a *recursion trace*. We used recursion traces, for example, to analyze and visualize the recursive factorial function of Section 3.5, and we similarly use recursion traces for the recursive sorting algorithms of Sections 11.1 and 11.2.

To draw a recursion trace, we create a box for each instance of the function and label it with the parameters of the function. Also, we visualize a recursive call by drawing an arrow from the box of the calling function to the box of the called function. For example, we illustrate the recursion trace of the LinearSum algorithm of Code Fragment 3.38 in Figure 3.19. We label each box in this trace with the parameters used to make this call. Each time we make a recursive call, we draw a line to the box representing the recursive call. We can also use this diagram to visualize stepping through the algorithm, since it proceeds by going from the call for n to the call for $n - 1$, to the call for $n - 2$, and so on, all the way down to the call for 1. When the final call finishes, it returns its value back to the call for 2, which adds in its value, and returns this partial sum to the call for 3, and so on, until the call for $n - 1$ returns its partial sum to the call for n.
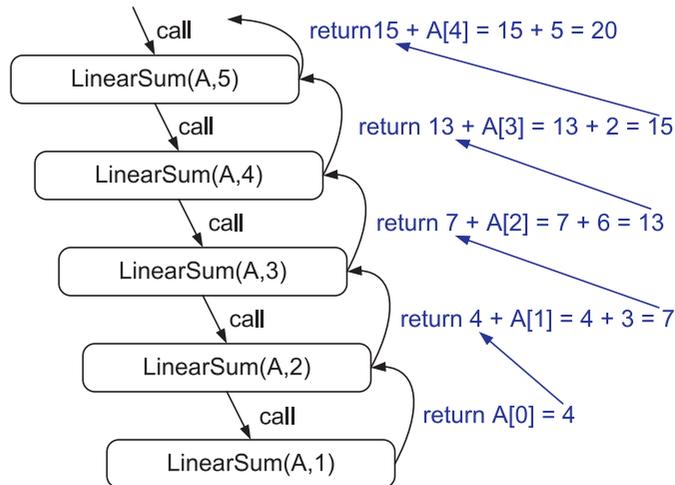
Figure 3.19: Recursion trace for an execution of LinearSum(A, n) with input parameters $A = \{4, 3, 6, 2, 5\}$ and $n = 5$.

From Figure 3.19, it should be clear that for an input array of size n, Algorithm LinearSum makes n calls. Hence, it takes an amount of time that is roughly proportional to n, since it spends a constant amount of time performing the nonrecursive part of each call. Moreover, we can also see that the memory space used by the algorithm (in addition to the array A) is also roughly proportional to n, since we need a constant amount of memory space for each of the n boxes in the trace at the

time we make the final recursive call (for $n = 1$).

Reversing an Array by Recursion

Next, let us consider the problem of reversing the n elements of an array, A, so that the first element becomes the last, the second element becomes second to the last, and so on. We can solve this problem using linear recursion, by observing that the reversal of an array can be achieved by swapping the first and last elements and then recursively reversing the remaining elements in the array. We describe the details of this algorithm in Code Fragment 3.39, using the convention that the first time we call this algorithm we do so as ReverseArray$(A, 0, n − 1)$.

Algorithm ReverseArray(A, i, j):
    Input: An array A and nonnegative integer indices i and j
    Output: The reversal of the elements in A starting at index i and ending at j
    if $i < j$ **then**
      Swap $A[i]$ and $A[j]$
      ReverseArray$(A, i + 1, j − 1)$
    return

Code Fragment 3.39: Reversing the elements of an array using linear recursion.

Note that, in this algorithm, we actually have two base cases, namely, when $i = j$ and when $i > j$. Moreover, in either case, we simply terminate the algorithm, since a sequence with zero elements or one element is trivially equal to its reversal. Furthermore, note that in the recursive step we are guaranteed to make progress towards one of these two base cases. If n is odd, we eventually reach the $i = j$ case, and if n is even, we eventually reach the $i > j$ case. The above argument immediately implies that the recursive algorithm of Code Fragment 3.39 is guaranteed to terminate.

Defining Problems in Ways that Facilitate Recursion

To design a recursive algorithm for a given problem, it is useful to think of the different ways we can subdivide this problem to define problems that have the same general structure as the original problem. This process sometimes means we need to redefine the original problem to facilitate similar-looking subproblems. For example, with the ReverseArray algorithm, we added the parameters i and j so that a recursive call to reverse the inner part of the array A would have the same structure (and same syntax) as the call to reverse all of A. Then, rather than initially calling the algorithm as ReverseArray(A), we call it initially as ReverseArray$(A, 0, n − 1)$. In general, if one has difficulty finding the repetitive structure needed to design a recursive algorithm, it is sometimes useful to work out the problem on a few concrete examples to see how the subproblems should be defined.

Tail Recursion

Using recursion can often be a useful tool for designing algorithms that have elegant, short definitions. But this usefulness does come at a modest cost. When we use a recursive algorithm to solve a problem, we have to use some of the memory locations in our computer to keep track of the state of each active recursive call. When computer memory is at a premium, then it is useful in some cases to be able to derive nonrecursive algorithms from recursive ones.

We can use the stack data structure, discussed in Section 5.1, to convert a recursive algorithm into a nonrecursive algorithm, but there are some instances when we can do this conversion more easily and efficiently. Specifically, we can easily convert algorithms that use *tail recursion*. An algorithm uses tail recursion if it uses linear recursion and the algorithm makes a recursive call as its very last operation. For example, the algorithm of Code Fragment 3.39 uses tail recursion to reverse the elements of an array.

It is not enough that the last statement in the function definition includes a recursive call, however. In order for a function to use tail recursion, the recursive call must be absolutely the last thing the function does (unless we are in a base case, of course). For example, the algorithm of Code Fragment 3.38 does not use tail recursion, even though its last statement includes a recursive call. This recursive call is not actually the last thing the function does. After it receives the value returned from the recursive call, it adds this value to $A[n-1]$ and returns this sum. That is, the last thing this algorithm does is an add, not a recursive call.

When an algorithm uses tail recursion, we can convert the recursive algorithm into a nonrecursive one, by iterating through the recursive calls rather than calling them explicitly. We illustrate this type of conversion by revisiting the problem of reversing the elements of an array. In Code Fragment 3.40, we give a nonrecursive algorithm that performs this task by iterating through the recursive calls of the algorithm of Code Fragment 3.39. We initially call this algorithm as IterativeReverseArray$(A, 0, n-1)$.

Algorithm IterativeReverseArray(A, i, j):

    Input: An array A and nonnegative integer indices i and j

    Output: The reversal of the elements in A starting at index i and ending at j

    while $i < j$ **do**

        Swap $A[i]$ and $A[j]$

        $i \leftarrow i+1$

        $j \leftarrow j-1$

    return

Code Fragment 3.40: Reversing the elements of an array using iteration.

3.5.2   Binary Recursion

When an algorithm makes two recursive calls, we say that it uses *binary recursion*. These calls can, for example, be used to solve two similar halves of some problem, as we did in Section 3.5 for drawing an English ruler. As another application of binary recursion, let us revisit the problem of summing the n elements of an integer array A. In this case, we can sum the elements in A by: (*i*) recursively summing the elements in the first half of A; (*ii*) recursively summing the elements in the second half of A; and (*iii*) adding these two values together. We give the details in the algorithm of Code Fragment 3.41, which we initially call as BinarySum($A, 0, n$).

Algorithm BinarySum(A, i, n):
    Input: An array A and integers i and n
    Output: The sum of the n integers in A starting at index i
    if $n = 1$ **then**
        return $A[i]$
    return BinarySum($A, i, \lceil n/2 \rceil$) + BinarySum($A, i + \lceil n/2 \rceil, \lfloor n/2 \rfloor$)

Code Fragment 3.41: Summing the elements in an array using binary recursion.

To analyze Algorithm BinarySum, we consider, for simplicity, the case where n is a power of two. The general case of arbitrary n is considered in Exercise R-4.5. Figure 3.20 shows the recursion trace of an execution of function BinarySum($0, 8$). We label each box with the values of parameters i and n, which represent the starting index and length of the sequence of elements to be summed, respectively. Notice that the arrows in the trace go from a box labeled (i, n) to another box labeled $(i, n/2)$ or $(i + n/2, n/2)$. That is, the value of parameter n is halved at each recursive call. Thus, the depth of the recursion, that is, the maximum number of function instances that are active at the same time, is $1 + \log_2 n$. Thus, Algorithm Binary-Sum uses an amount of additional space roughly proportional to this value. This is a big improvement over the space needed by the LinearSum function of Code Fragment 3.38. The running time of Algorithm BinarySum is still roughly proportional to n, however, since each box is visited in constant time when stepping through our algorithm and there are $2n - 1$ boxes.
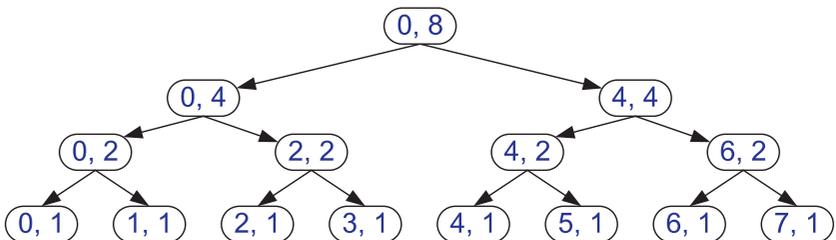
Figure 3.20: Recursion trace for the execution of BinarySum($0, 8$).

Computing Fibonacci Numbers via Binary Recursion

Let us consider the problem of computing the kth Fibonacci number. Recall from Section 2.2.3, that the Fibonacci numbers are recursively defined as follows:

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_i &= F_{i-1} + F_{i-2} \quad \text{for } i > 1
\end{aligned}
$$

By directly applying this definition, Algorithm BinaryFib, shown in Code Fragment 3.42, computes the sequence of Fibonacci numbers using binary recursion.

Algorithm BinaryFib(k):

    Input: Nonnegative integer k

    Output: The kth Fibonacci number F_k

  if $k \leq 1$ **then**

    return k

  else

    return BinaryFib($k-1$) + BinaryFib($k-2$)

Code Fragment 3.42: Computing the kth Fibonacci number using binary recursion.

Unfortunately, in spite of the Fibonacci definition looking like a binary recursion, using this technique is inefficient in this case. In fact, it takes an exponential number of calls to compute the kth Fibonacci number in this way. Specifically, let n_k denote the number of calls performed in the execution of BinaryFib(k). Then, we have the following values for the n_k's:

$$
\begin{aligned}
n_0 &= 1 \\
n_1 &= 1 \\
n_2 &= n_1 + n_0 + 1 = 1 + 1 + 1 = 3 \\
n_3 &= n_2 + n_1 + 1 = 3 + 1 + 1 = 5 \\
n_4 &= n_3 + n_2 + 1 = 5 + 3 + 1 = 9 \\
n_5 &= n_4 + n_3 + 1 = 9 + 5 + 1 = 15 \\
n_6 &= n_5 + n_4 + 1 = 15 + 9 + 1 = 25 \\
n_7 &= n_6 + n_5 + 1 = 25 + 15 + 1 = 41 \\
n_8 &= n_7 + n_6 + 1 = 41 + 25 + 1 = 67
\end{aligned}
$$

If we follow the pattern forward, we see that the number of calls more than doubles for each two consecutive indices. That is, n_4 is more than twice n_2, n_5 is more than twice n_3, n_6 is more than twice n_4, and so on. Thus, $n_k > 2^{k/2}$, which means that BinaryFib(k) makes a number of calls that are exponential in k. In other words, using binary recursion to compute Fibonacci numbers is very inefficient.

Computing Fibonacci Numbers via Linear Recursion

The main problem with the approach above, based on binary recursion, is that the computation of Fibonacci numbers is really a linearly recursive problem. It is not a good candidate for using binary recursion. We simply got tempted into using binary recursion because of the way the kth Fibonacci number, F_k, depends on the two previous values, F_{k-1} and F_{k-2}. But we can compute F_k much more efficiently using linear recursion.

In order to use linear recursion, however, we need to slightly redefine the problem. One way to accomplish this conversion is to define a recursive function that computes a pair of consecutive Fibonacci numbers (F_k, F_{k-1}) using the convention $F_{-1} = 0$. Then we can use the linearly recursive algorithm shown in Code Fragment 3.43.

Algorithm LinearFibonacci(k):

    Input: A nonnegative integer k

    Output: Pair of Fibonacci numbers (F_k, F_{k-1})

   if $k \le 1$ **then**

     return $(k, 0)$

   else

     $(i, j) \leftarrow$ LinearFibonacci($k - 1$)

     return $(i + j, i)$

Code Fragment 3.43: Computing the kth Fibonacci number using linear recursion.

The algorithm given in Code Fragment 3.43 shows that using linear recursion to compute Fibonacci numbers is much more efficient than using binary recursion. Since each recursive call to LinearFibonacci decreases the argument k by 1, the original call LinearFibonacci(k) results in a series of $k - 1$ additional calls. That is, computing the kth Fibonacci number via linear recursion requires k function calls. This performance is significantly faster than the exponential time needed by the algorithm based on binary recursion, which was given in Code Fragment 3.42. Therefore, when using binary recursion, we should first try to fully partition the problem in two (as we did for summing the elements of an array) or we should be sure that overlapping recursive calls are really necessary.

Usually, we can eliminate overlapping recursive calls by using more memory to keep track of previous values. In fact, this approach is a central part of a technique called ***dynamic programming***, which is related to recursion and is discussed in Section 12.2.

3.5.3 Multiple Recursion

Generalizing from binary recursion, we use ***multiple recursion*** when a function may make multiple recursive calls, with that number potentially being more than two. One of the most common applications of this type of recursion is used when we want to enumerate various configurations in order to solve a combinatorial puzzle. For example, the following are all instances of ***summation puzzles***.

$$
\begin{aligned}
pot + pan &= bib \\
dog + cat &= pig \\
boy + girl &= baby
\end{aligned}
$$

To solve such a puzzle, we need to assign a unique digit (that is, $0, 1, \ldots, 9$) to each letter in the equation, in order to make the equation true. Typically, we solve such a puzzle by using our human observations of the particular puzzle we are trying to solve to eliminate configurations (that is, possible partial assignments of digits to letters) until we can work though the feasible configurations left, testing for the correctness of each one.

If the number of possible configurations is not too large, however, we can use a computer to simply enumerate all the possibilities and test each one, without employing any human observations. In addition, such an algorithm can use multiple recursion to work through the configurations in a systematic way. We show pseudo-code for such an algorithm in Code Fragment 3.44. To keep the description general enough to be used with other puzzles, the algorithm enumerates and tests all k-length sequences without repetitions of the elements of a given set U. We build the sequences of k elements by the following steps:

1. Recursively generating the sequences of $k - 1$ elements

2. Appending to each such sequence an element not already contained in it.

Throughout the execution of the algorithm, we use the set U to keep track of the elements not contained in the current sequence, so that an element e has not been used yet if and only if e is in U.

Another way to look at the algorithm of Code Fragment 3.44 is that it enumerates every possible size-k ordered subset of U, and tests each subset for being a possible solution to our puzzle.

For summation puzzles, $U = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and each position in the sequence corresponds to a given letter. For example, the first position could stand for b, the second for o, the third for y, and so on.

Algorithm PuzzleSolve(*k*, *S*, *U*):

   Input: An integer *k*, sequence *S*, and set *U*

   Output: An enumeration of all *k*-length extensions to *S* using elements in *U*
     without repetitions

  for each *e* in *U* **do**

    Remove *e* from *U* {*e* is now being used}

    Add *e* to the end of *S*

    if *k* = 1 **then**

      Test whether *S* is a configuration that solves the puzzle

      if *S* solves the puzzle **then**

        return "Solution found: " *S*

    else

      PuzzleSolve(*k* − 1, *S*, *U*)

    Add *e* back to *U* {*e* is now unused}

    Remove *e* from the end of *S*

Code Fragment 3.44: Solving a combinatorial puzzle by enumerating and testing all possible configurations.

In Figure 3.21, we show a recursion trace of a call to PuzzleSolve(3, *S*, *U*), where *S* is empty and *U* = {*a*, *b*, *c*}. During the execution, all the permutations of the three characters are generated and tested. Note that the initial call makes three recursive calls, each of which in turn makes two more. If we had executed PuzzleSolve(3, *S*, *U*) on a set *U* consisting of four elements, the initial call would have made four recursive calls, each of which would have a trace looking like the one in Figure 3.21.
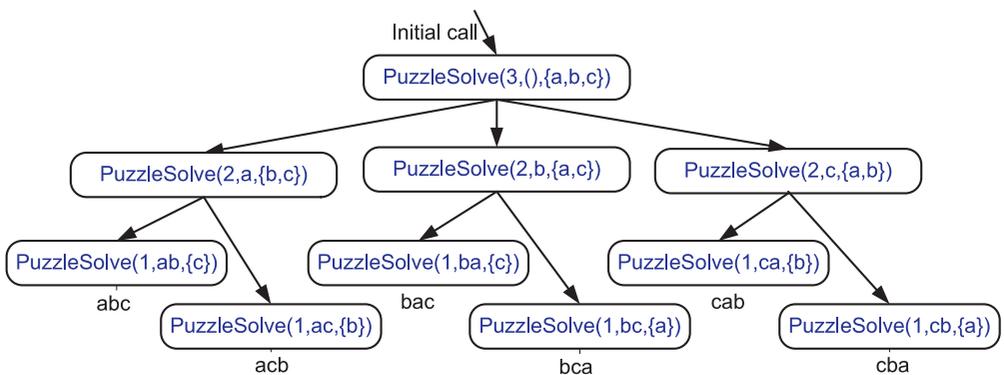
Figure 3.21: Recursion trace for an execution of PuzzleSolve(3, *S*, *U*), where *S* is empty and *U* = {*a*, *b*, *c*}. This execution generates and tests all permutations of *a*, *b*, and *c*. We show the permutations generated directly below their respective boxes.

3.6 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-3.1 Modify the implementation of class Scores so that at most $\lceil maxEnt/2 \rceil$ of the scores can come from any one single player.

R-3.2 Suppose that two entries of an array A are equal to each other. After running the insertion-sort algorithm of Code Fragment 3.7, will they appear in the same relative order in the final sorted order or in reverse order? Explain your answer.

R-3.3 Give a C++ code fragment that, given a $n \times n$ matrix M of type **float**, replaces M with its transpose. Try to do this without the use of a temporary matrix.

R-3.4 Describe a way to use recursion to compute the sum of all the elements in a $n \times n$ (two-dimensional) array of integers.

R-3.5 Give a recursive definition of a singly linked list.

R-3.6 Add a function size() to our C++ implementation of a singly link list. Can you design this function so that it runs in $O(1)$ time?

R-3.7 Give an algorithm for finding the penultimate (second to last) node in a singly linked list where the last element is indicated by a null next link.

R-3.8 Give a fully generic implementation of the doubly linked list data structure of Section 3.3.3 by using a templated class.

R-3.9 Give a more robust implementation of the doubly linked list data structure of Section 3.3.3, which throws an appropriate exception if an illegal operation is attempted.

R-3.10 Describe a nonrecursive function for finding, by link hopping, the middle node of a doubly linked list with header and trailer sentinels. (Note: This function must only use link hopping; it cannot use a counter.) What is the running time of this function?

R-3.11 Describe a recursive algorithm for finding the maximum element in an array A of n elements. What is your running time and space usage?

R-3.12 Draw the recursion trace for the execution of function ReverseArray$(A,0,4)$ (Code Fragment 3.39) on array $A = \{4,3,6,2,5\}$.

R-3.13 Draw the recursion trace for the execution of function PuzzleSolve$(3,S,U)$ (Code Fragment 3.44), where S is empty and $U = \{a,b,c,d\}$.

R-3.14 Write a short C++ function that repeatedly selects and removes a random entry from an n-element array until the array holds no more entries. Assume that you have access to a function random(k), which returns a random integer in the range from 0 to k.

R-3.15 Give a fully generic implementation of the circularly linked list data structure of Section 3.4.1 by using a templated class.

R-3.16 Give a more robust implementation of the circularly linked list data structure of Section 3.4.1, which throws an appropriate exception if an illegal operation is attempted.

R-3.17 Write a short C++ function to count the number of nodes in a circularly linked list.

Creativity

C-3.1 In the Tic-Tac-Toe example, we used 1 for player X and -1 for player O. Explain how to modify the program's counting trick to decide the winner if we had used 1 for player X and 4 for player O instead. Could we use any combination of values a and b for the two players? Explain.

C-3.2 Give C++ code for performing add(e) and remove(i) functions for game entries stored in an array a, as in class Scores in Section 3.1.1, except this time, don't maintain the game entries in order. Assume that we still need to keep n entries stored in indices 0 to $n-1$. Try to implement the add and remove functions without using any loops, so that the number of steps they perform does not depend on n.

C-3.3 Let A be an array of size $n \geq 2$ containing integers from 1 to $n-1$, inclusive, with exactly one repeated. Describe a fast algorithm for finding the integer in A that is repeated.

C-3.4 Let B be an array of size $n \geq 6$ containing integers from 1 to $n-5$, inclusive, with exactly five repeated. Describe a good algorithm for finding the five integers in B that are repeated.

C-3.5 Suppose you are designing a multi-player game that has $n \geq 1000$ players, numbered 1 to n, interacting in an enchanted forest. The winner of this game is the first player who can meet all the other players at least once (ties are allowed). Assuming that there is a function meet(i, j), which is called each time a player i meets a player j (with $i \neq j$), describe a way to keep track of the pairs of meeting players and who is the winner.

C-3.6 Give a recursive algorithm to compute the product of two positive integers, m and n, using only addition and subtraction.

C-3.7 Describe a fast recursive algorithm for reversing a singly linked list L, so that the ordering of the nodes becomes opposite of what it was before.

C-3.8 Describe a good algorithm for concatenating two singly linked lists L and M, with header sentinels, into a single list L' that contains all the nodes of L followed by all the nodes of M.

C-3.9 Give a fast algorithm for concatenating two doubly linked lists L and M, with header and trailer sentinel nodes, into a single list L'.

C-3.10 Describe in detail how to swap two nodes x and y (and not just their contents) in a singly linked list L given references only to x and y. Repeat this exercise for the case when L is a doubly linked list. Which algorithm takes more time?

C-3.11 Describe in detail an algorithm for reversing a singly linked list L using only a constant amount of additional space and not using any recursion.

C-3.12 In the ***Towers of Hanoi*** puzzle, we are given a platform with three pegs, a, b, and c, sticking out of it. On peg a is a stack of n disks, each larger than the next, so that the smallest is on the top and the largest is on the bottom. The puzzle is to move all the disks from peg a to peg c, moving one disk at a time, so that we never place a larger disk on top of a smaller one. Describe a recursive algorithm for solving the Towers of Hanoi puzzle for arbitrary n.

(Hint: Consider first the subproblem of moving all but the nth disk from peg a to another peg using the third as "temporary storage.")

C-3.13 Describe a recursive function for converting a string of digits into the integer it represents. For example, `"13531"` represents the integer $13,531$.

C-3.14 Describe a recursive algorithm that counts the number of nodes in a singly linked list.

C-3.15 Write a recursive C++ program that will output all the subsets of a set of n elements (without repeating any subsets).

C-3.16 Write a short recursive C++ function that finds the minimum and maximum values in an array of **int** values without using any loops.

C-3.17 Describe a recursive algorithm that will check if an array A of integers contains an integer $A[i]$ that is the sum of two integers that appear earlier in A, that is, such that $A[i] = A[j] + A[k]$ for $j, k < i$.

C-3.18 Write a short recursive C++ function that will rearrange an array of **int** values so that all the even values appear before all the odd values.

C-3.19 Write a short recursive C++ function that takes a character string s and outputs its reverse. So for example, the reverse of `"pots&pans"` would be `"snap&stop"`.

C-3.20 Write a short recursive C++ function that determines if a string s is a palindrome, that is, it is equal to its reverse. For example, `"racecar"` and `"gohangasalamiimalasagnahog"` are palindromes.

C-3.21 Use recursion to write a C++ function for determining if a string s has more vowels than consonants.

C-3.22 Suppose you are given two circularly linked lists, L and M, that is, two lists of nodes such that each node has a nonnull next node. Describe a fast algorithm for telling if L and M are really the same list of nodes but with different (cursor) starting points.

C-3.23 Given a circularly linked list L containing an even number of nodes, describe how to split L into two circularly linked lists of half the size.

Projects

P-3.1 Write a C++ function that takes two three-dimensional integer arrays and adds them componentwise.

P-3.2 Write a C++ program for a matrix class that can add and multiply arbitrary two-dimensional arrays of integers. Do this by overloading the addition ("+") and multiplication ("∗") operators.

P-3.3 Write a class that maintains the top 10 scores for a game application, implementing the add and remove functions of Section 3.1.1, but use a singly linked list instead of an array.

P-3.4 Perform the previous project but use a doubly linked list. Moreover, your implementation of remove(i) should make the fewest number of pointer hops to get to the game entry at index i.

P-3.5 Perform the previous project but use a linked list that is both circularly linked and doubly linked.

P-3.6 Write a program for solving summation puzzles by enumerating and testing all possible configurations. Using your program, solve the three puzzles given in Section 3.5.3.

P-3.7 Write a program that can perform encryption and decryption using an arbitrary substitution cipher. In this case, the encryption array is a random shuffling of the letters in the alphabet. Your program should generate a random encryption array, its corresponding decryption array, and use these to encode and decode a message.

P-3.8 Write a program that can solve instances of the Tower of Hanoi problem (from Exercise C-3.12).

Chapter Notes

The fundamental data structures of arrays and linked lists, as well as recursion, discussed in this chapter, belong to the folklore of computer science. They were first chronicled in the computer science literature by Knuth in his seminal book on *Fundamental Algorithms* [59].

Chapter

4

Analysis Tools
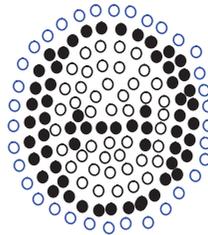
Contents

4.1   The Seven Functions Used in This Book

In this section, we briefly discuss the seven most important functions used in the analysis of algorithms. We use only these seven simple functions for almost all the analysis we do in this book. In fact, sections that use a function other than one of these seven are marked with a star ($\star$) to indicate that they are optional. In addition to these seven fundamental functions, Appendix A contains a list of other useful mathematical facts that apply in the context of data structure and algorithm analysis.

4.1.1   The Constant Function

The simplest function we can think of is the ***constant function***. This is the function,

$$f(n) = c,$$

for some fixed constant c, such as $c = 5$, $c = 27$, or $c = 2^{10}$. That is, for any argument n, the constant function $f(n)$ assigns the value c. In other words, it doesn't matter what the value of n is, $f(n)$ is always be equal to the constant value c.

Since we are most interested in integer functions, the most fundamental constant function is $g(n) = 1$, and this is the typical constant function we use in this book. Note that any other constant function, $f(n) = c$, can be written as a constant c times $g(n)$. That is, $f(n) = cg(n)$ in this case.

As simple as it is, the constant function is useful in algorithm analysis because it characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers.

4.1.2   The Logarithm Function

One of the interesting and sometimes even surprising aspects of the analysis of data structures and algorithms is the ubiquitous presence of the ***logarithm function***, $f(n) = \log_b n$, for some constant $b > 1$. This function is defined as follows:

$$x = \log_b n \quad \text{if and only if} \quad b^x = n.$$

By definition, $\log_b 1 = 0$. The value b is known as the ***base*** of the logarithm.

Computing the logarithm function exactly for any integer n involves the use of calculus, but we can use an approximation that is good enough for our purposes without calculus. In particular, we can easily compute the smallest integer greater than or equal to $\log_a n$, since this number is equal to the number of times

we can divide n by a until we get a number less than or equal to 1. For example, this evaluation of $\log_3 27$ is 3, since $27/3/3/3 = 1$. Likewise, this evaluation of $\log_4 64$ is 3, since $64/4/4/4 = 1$, and this approximation to $\log_2 12$ is 4, since $12/2/2/2/2 = 0.75 \leq 1$. This base-2 approximation arises in algorithm analysis, since a common operation in many algorithms is to repeatedly divide an input in half.

Indeed, since computers store integers in binary, the most common base for the logarithm function in computer science is 2. In fact, this base is so common that we typically leave it off when it is 2. That is, for us,

$$\log n = \log_2 n.$$

We note that most handheld calculators have a button marked LOG, but this is typically for calculating the logarithm base 10, not base 2.

There are some important rules for logarithms, similar to the exponent rules.

Proposition 4.1 (Logarithm Rules): *Given real numbers $a > 0$, $b > 1$, $c > 0$ and $d > 1$, we have:*

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_d a)/\log_d b$
5. $b^{\log_d a} = a^{\log_d b}$

Also, as a notational shorthand, we use $\log^c n$ to denote the function $(\log n)^c$. Rather than show how we could derive each of the identities above which all follow from the definition of logarithms and exponents, let us illustrate these identities with a few examples instead.

Example 4.2: *We demonstrate below some interesting applications of the logarithm rules from Proposition 4.1 (using the usual convention that the base of a logarithm is 2 if it is omitted).*

- $\log(2n) = \log 2 + \log n = 1 + \log n$, *by rule 1*
- $\log(n/2) = \log n - \log 2 = \log n - 1$, *by rule 2*
- $\log n^3 = 3 \log n$, *by rule 3*
- $\log 2^n = n \log 2 = n \cdot 1 = n$, *by rule 3*
- $\log_4 n = (\log n)/\log 4 = (\log n)/2$, *by rule 4*
- $2^{\log n} = n^{\log 2} = n^1 = n$, *by rule 5*

As a practical matter, we note that rule 4 gives us a way to compute the base-2 logarithm on a calculator that has a base-10 logarithm button, LOG, for

$$\log_2 n = \text{LOG } n / \text{LOG } 2.$$

4.1.3   The Linear Function

Another simple yet important function is the ***linear function***,

$$f(n) = n.$$

That is, given an input value n, the linear function f assigns the value n itself.

This function arises in algorithm analysis any time we have to do a single basic operation for each of n elements. For example, comparing a number x to each element of an array of size n requires n comparisons. The linear function also represents the best running time we can hope to achieve for any algorithm that processes a collection of n objects that are not already in the computer's memory, since reading in the n objects itself requires n operations.

4.1.4   The N-Log-N Function

The next function we discuss in this section is the ***n-log-n function***,

$$f(n) = n \log n.$$

That is, the function that assigns to an input n the value of n times the logarithm base 2 of n. This function grows a little faster than the linear function and a lot slower than the quadratic function. Thus, as we show on several occasions, if we can improve the running time of solving some problem from quadratic to n-log-n, we have an algorithm that runs much faster in general.

4.1.5   The Quadratic Function

Another function that appears quite often in algorithm analysis is the ***quadratic function***,

$$f(n) = n^2.$$

That is, given an input value n, the function f assigns the product of n with itself (in other words, "n squared").

The main reason why the quadratic function appears in the analysis of algorithms is that there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. Thus, in such cases, the algorithm performs $n \cdot n = n^2$ operations.

Nested Loops and the Quadratic Function

The quadratic function can also arise in the context of nested loops where the first iteration of a loop uses one operation, the second uses two operations, the third uses three operations, and so on. That is, the number of operations is

$$1 + 2 + 3 + \cdots + (n-2) + (n-1) + n.$$

In other words, this is the total number of operations that are performed by the nested loop if the number of operations performed inside the loop increases by one with each iteration of the outer loop. This quantity also has an interesting history.

In 1787, a German schoolteacher decided to keep his 9- and 10-year-old pupils occupied by adding up the integers from 1 to 100. But almost immediately one of the children claimed to have the answer! The teacher was suspicious, for the student had only the answer on his slate. But the answer was correct—5,050—and the student, Carl Gauss, grew up to be one of the greatest mathematicians of his time. It is widely suspected that young Gauss used the following identity.

Proposition 4.3: *For any integer $n \geq 1$, we have:*

$$1 + 2 + 3 + \cdots + (n-2) + (n-1) + n = \frac{n(n+1)}{2}.$$

We give two "visual" justifications of Proposition 4.3 in Figure 4.1.
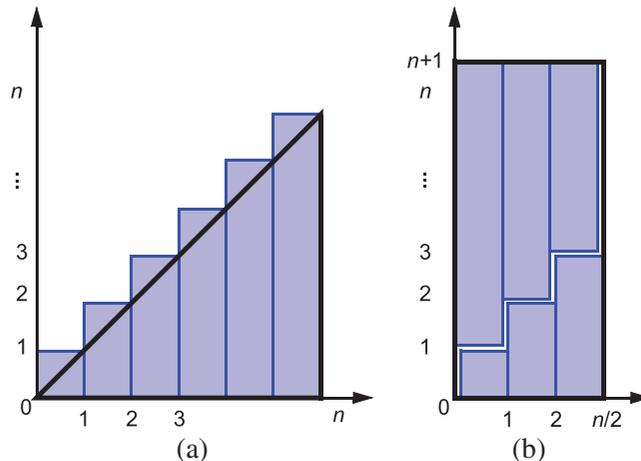
Figure 4.1: Visual justifications of Proposition 4.3. Both illustrations visualize the identity in terms of the total area covered by n unit-width rectangles with heights $1, 2, \ldots, n$. In (a), the rectangles are shown to cover a big triangle of area $n^2/2$ (base n and height n) plus n small triangles of area $1/2$ each (base 1 and height 1). In (b), which applies only when n is even, the rectangles are shown to cover a big rectangle of base $n/2$ and height $n+1$.

The lesson to be learned from Proposition 4.3 is that if we perform an algorithm with nested loops such that the operations in the inner loop increase by one each time, then the total number of operations is quadratic in the number of times, n, we perform the outer loop. In particular, the number of operations is $n^2/2 + n/2$, in this case, which is a little more than a constant factor $(1/2)$ times the quadratic function n^2. In other words, such an algorithm is only slightly better than an algorithm that uses n operations each time the inner loop is performed. This observation might at first seem nonintuitive, but it is nevertheless true as shown in Figure 4.1.

4.1.6   The Cubic Function and Other Polynomials

Continuing our discussion of functions that are powers of the input, we consider the **cubic function**,

$$f(n) = n^3,$$

which assigns to an input value n the product of n with itself three times. This function appears less frequently in the context of algorithm analysis than the constant, linear, and quadratic functions previously mentioned, but it does appear from time to time.

Polynomials

Interestingly, the functions we have listed so far can be viewed as all being part of a larger class of functions, the **polynomials**.

A **polynomial** function is a function of the form,

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d,$$

where $a_0, a_1, \ldots, a_d$ are constants, called the **coefficients** of the polynomial, and $a_d \neq 0$. Integer d, which indicates the highest power in the polynomial, is called the **degree** of the polynomial.

For example, the following functions are all polynomials:

- $f(n) = 2 + 5n + n^2$
- $f(n) = 1 + n^3$
- $f(n) = 1$
- $f(n) = n$
- $f(n) = n^2$

Therefore, we could argue that this book presents just four important functions used in algorithm analysis, but we stick to saying that there are seven, since the constant, linear, and quadratic functions are too important to be lumped in with other polynomials. Running times that are polynomials with degree, d, are generally better than polynomial running times of larger degree.

Summations

A notation that appears again and again in the analysis of data structures and algorithms is the **summation**, which is defined as

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b),$$

where a and b are integers and $a \leq b$. Summations arise in data structure and algorithm analysis because the running times of loops naturally give rise to summations.

Using a summation, we can rewrite the formula of Proposition 4.3 as

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Likewise, we can write a polynomial $f(n)$ of degree d with coefficients $a_0, \ldots, a_d$ as

$$f(n) = \sum_{i=0}^{d} a_i n^i.$$

Thus, the summation notation gives us a shorthand way of expressing sums of increasing terms that have a regular structure.

4.1.7 The Exponential Function

Another function used in the analysis of algorithms is the **exponential function**,

$$f(n) = b^n,$$

where b is a positive constant, called the **base**, and the argument n is the **exponent**. That is, function $f(n)$ assigns to the input argument n the value obtained by multiplying the base b by itself n times. In algorithm analysis, the most common base for the exponential function is $b = 2$. For instance, if we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the nth iteration is 2^n. In addition, an integer word containing n bits can represent all the nonnegative integers less than 2^n. Thus, the exponential function with base 2 is quite common. The exponential function is also referred to as **exponent function**.

We sometimes have other exponents besides n, however; hence, it is useful for us to know a few handy rules for working with exponents. In particular, the following **exponent rules** are quite helpful.

Proposition 4.4 (Exponent Rules): *Given positive integers a, b, and c, we have:*
1. $(b^a)^c = b^{ac}$
2. $b^a b^c = b^{a+c}$
3. $b^a / b^c = b^{a-c}$

For example, we have the following:
- $256 = 16^2 = (2^4)^2 = 2^{4 \cdot 2} = 2^8 = 256$ (Exponent Rule 1)
- $243 = 3^5 = 3^{2+3} = 3^2 3^3 = 9 \cdot 27 = 243$ (Exponent Rule 2)
- $16 = 1024/64 = 2^{10}/2^6 = 2^{10-6} = 2^4 = 16$ (Exponent Rule 3)

We can extend the exponential function to exponents that are fractions or real numbers and to negative exponents, as follows. Given a positive integer k, we define $b^{1/k}$ to be kth root of b, that is, the number r such that $r^k = b$. For example, $25^{1/2} = 5$, since $5^2 = 25$. Likewise, $27^{1/3} = 3$ and $16^{1/4} = 2$. This approach allows us to define any power whose exponent can be expressed as a fraction, since $b^{a/c} = (b^a)^{1/c}$, by Exponent Rule 1. For example, $9^{3/2} = (9^3)^{1/2} = 729^{1/2} = 27$. Thus, $b^{a/c}$ is really just the cth root of the integral exponent b^a.

We can further extend the exponential function to define b^x for any real number x, by computing a series of numbers of the form $b^{a/c}$ for fractions a/c that get progressively closer and closer to x. Any real number x can be approximated arbitrarily close by a fraction a/c; hence, we can use the fraction a/c as the exponent of b to get arbitrarily close to b^x. So, for example, the number 2^π is well defined. Finally, given a negative exponent d, we define $b^d = 1/b^{-d}$, which corresponds to applying Exponent Rule 3 with $a = 0$ and $c = -d$.

Geometric Sums

Suppose we have a loop where each iteration takes a multiplicative factor longer than the previous one. This loop can be analyzed using the following proposition.

Proposition 4.5: *For any integer $n \geq 0$ and any real number a such that $a > 0$ and $a \neq 1$, consider the summation*

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n$$

(remembering that $a^0 = 1$ if $a > 0$). This summation is equal to

$$\frac{a^{n+1} - 1}{a - 1}.$$

Summations as shown in Proposition 4.5 are called ***geometric*** summations, because each term is geometrically larger than the previous one if $a > 1$. For example, everyone working in computing should know that

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1,$$

since this is the largest integer that can be represented in binary notation using n bits.

4.1.8 Comparing Growth Rates

To sum up, Table 4.1 shows each of the seven common functions used in algorithm analysis in order.

constant	*logarithm*	*linear*	*n-log-n*	*quadratic*	*cubic*	*exponential*
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Table 4.1: Classes of functions. Here we assume that $a > 1$ is a constant.

Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or n-log-n time. Algorithms with quadratic or cubic running times are less practical, but algorithms with exponential running times are infeasible for all but the smallest sized inputs. Plots of the seven functions are shown in Figure 4.2.
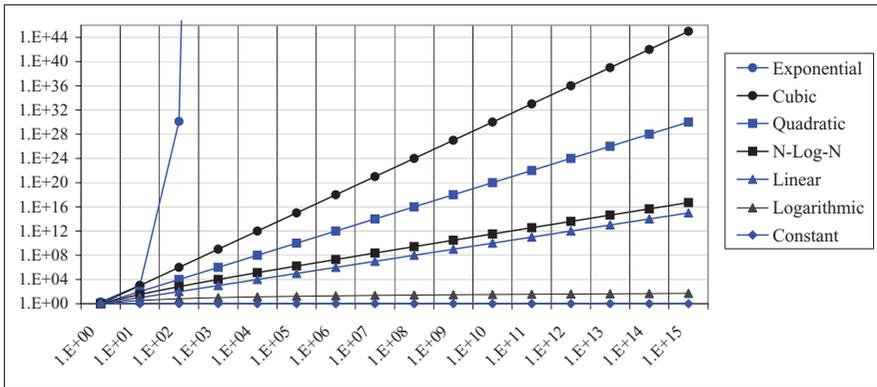
Figure 4.2: Growth rates for the seven fundamental functions used in algorithm analysis. We use base $a = 2$ for the exponential function. The functions are plotted in a log-log chart, to compare the growth rates primarily as slopes. Even so, the exponential function grows too fast to display all its values on the chart. Also, we use the scientific notation for numbers, where aE+b denotes $a10^b$.

The Ceiling and Floor Functions

One additional comment concerning the functions above is in order. The value of a logarithm is typically not an integer, yet the running time of an algorithm is usually expressed by means of an integer quantity, such as the number of operations performed. Thus, the analysis of an algorithm may sometimes involve the use of the *floor function* and *ceiling function*, which are defined respectively as follows:

- $\lfloor x \rfloor$ = the largest integer less than or equal to x
- $\lceil x \rceil$ = the smallest integer greater than or equal to x

4.2   Analysis of Algorithms

In a classic story, the famous mathematician Archimedes was asked to determine if a golden crown commissioned by the king was indeed pure gold, and not part silver, as an informant had claimed. Archimedes discovered a way to perform this analysis while stepping into a (Greek) bath. He noted that water spilled out of the bath in proportion to the amount of him that went in. Realizing the implications of this fact, he immediately got out of the bath and ran naked through the city shouting, "Eureka, eureka!," for he had discovered an analysis tool (displacement), which, when combined with a simple scale, could determine if the king's new crown was good or not. That is, Archimedes could dip the crown and an equal-weight amount of gold into a bowl of water to see if they both displaced the same amount. This discovery was unfortunate for the goldsmith, however, for when Archimedes did his analysis, the crown displaced more water than an equal-weight lump of pure gold, indicating that the crown was not, in fact, pure gold.

In this book, we are interested in the design of "good" data structures and algorithms. Simply put, a *data structure* is a systematic way of organizing and accessing data, and an *algorithm* is a step-by-step procedure for performing some task in a finite amount of time. These concepts are central to computing, but to be able to classify some data structures and algorithms as "good," we must have precise ways of analyzing them.

The primary analysis tool we use in this book involves characterizing the running times of algorithms and data structure operations, with space usage also being of interest. Running time is a natural measure of "goodness," since time is a precious resource—computer solutions should run as fast as possible.

In general, the running time of an algorithm or data structure method increases with the input size, although it may also vary for different inputs of the same size. Also, the running time is affected by the hardware environment (as reflected in the processor, clock rate, memory, disk, etc.) and software environment (as reflected in the operating system, programming language, compiler, interpreter, etc.) in which the algorithm is implemented, compiled, and executed. All other factors being equal, the running time of the same algorithm on the same input data is smaller if the computer has, say, a much faster processor or if the implementation is done in a program compiled into native machine code instead of an interpreted implementation run on a virtual machine. Nevertheless, in spite of the possible variations that come from different environmental factors, we would like to focus on the relationship between the running time of an algorithm and the size of its input.

We are interested in characterizing an algorithm's running time as a function of the input size. But what is the proper way of measuring it?

4.2.1 Experimental Studies

If an algorithm has been implemented, we can study its running time by executing it on various test inputs and recording the actual time spent in each execution. Fortunately, such measurements can be taken in an accurate manner by using system calls that are built into the language or operating system (for example, by using the clock() function or calling the run-time environment with profiling enabled). Such tests assign a specific running time to a specific input size, but we are interested in determining the general dependence of running time on the size of the input. In order to determine this dependence, we should perform several experiments on many different test inputs of various sizes. Then we can visualize the results of such experiments by plotting the performance of each run of the algorithm as a point with x-coordinate equal to the input size, n, and y-coordinate equal to the running time, t. (See Figure 4.3.) From this visualization and the data that supports it, we can perform a statistical analysis that seeks to fit the best function of the input size to the experimental data. To be meaningful, this analysis requires that we choose good sample inputs and test enough of them to be able to make sound statistical claims about the algorithm's running time.
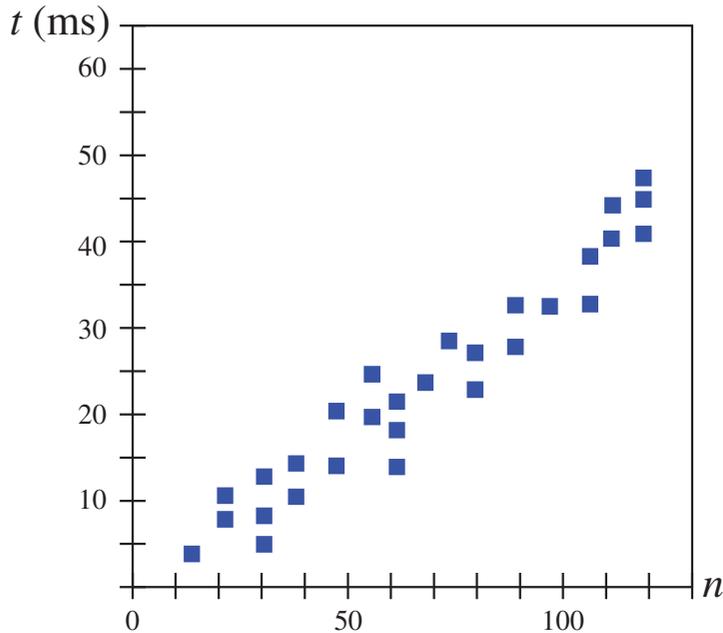
Figure 4.3: Results of an experimental study on the running time of an algorithm. A dot with coordinates (n,t) indicates that on an input of size n, the running time of the algorithm is t milliseconds (ms).

While experimental studies of running times are useful, they have three major limitations:

- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- We have difficulty comparing the experimental running times of two algorithms unless the experiments were performed in the same hardware and software environments.
- We have to fully implement and execute an algorithm in order to study its running time experimentally.

This last requirement is obvious, but it is probably the most time consuming aspect of performing an experimental analysis of an algorithm. The other limitations impose serious hurdles too, of course. Thus, we would ideally like to have an analysis tool that allows us to avoid performing experiments.

In the rest of this chapter, we develop a general way of analyzing the running times of algorithms that:

- Takes into account all possible inputs.
- Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent from the hardware and software environment.
- Can be performed by studying a high-level description of the algorithm without actually implementing it or running experiments on it.

This methodology aims at associating, with each algorithm, a function $f(n)$ that characterizes the running time of the algorithm as a function of the input size n. Typical functions that are encountered include the seven functions mentioned earlier in this chapter.

4.2.2   Primitive Operations

As noted above, experimental analysis is valuable, but it has its limitations. If we wish to analyze a particular algorithm without performing experiments on its running time, we can perform an analysis directly on the high-level pseudo-code instead. We define a set of ***primitive operations*** such as the following:

- Assigning a value to a variable
- Calling a function
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a function

Counting Primitive Operations

Specifically, a primitive operation corresponds to a low-level instruction with an execution time that is constant. Instead of trying to determine the specific execution time of each primitive operation, we simply *count* how many primitive operations are executed, and use this number t as a measure of the running time of the algorithm.

This operation count correlates to an actual running time in a specific computer, since each primitive operation corresponds to a constant-time instruction, and there are only a fixed number of primitive operations. The implicit assumption in this approach is that the running times of different primitive operations is fairly similar. Thus, the number, t, of primitive operations an algorithm performs is proportional to the actual running time of that algorithm.

An algorithm may run faster on some inputs than it does on others of the same size. Thus, we may wish to express the running time of an algorithm as the function of the input size obtained by taking the average over all possible inputs of the same size. Unfortunately, such an *average-case* analysis is typically quite challenging. It requires us to define a probability distribution on the set of inputs, which is often a difficult task. Figure 4.4 schematically shows how, depending on the input distribution, the running time of an algorithm can be anywhere between the worst-case time and the best-case time. For example, what if inputs are really only of types "A" or "D"?
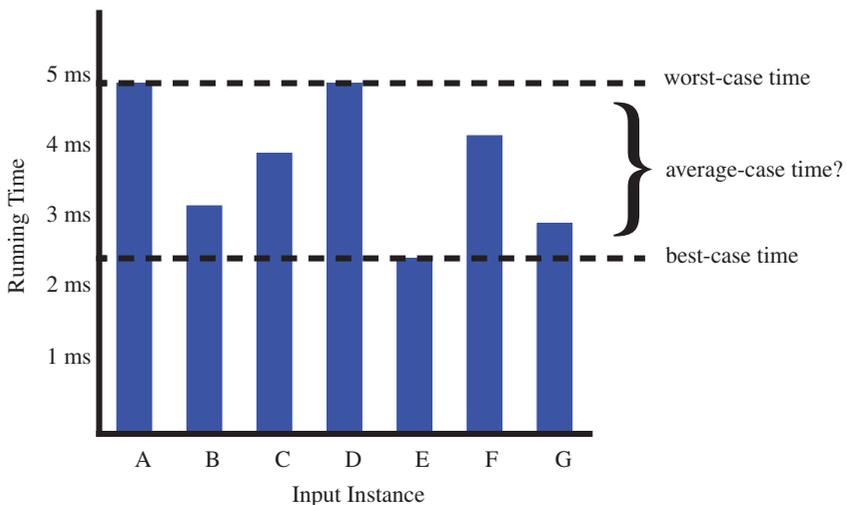
Figure 4.4: The difference between best-case and worst-case time. Each bar represents the running time of some algorithm on a different possible input.

Focusing on the Worst Case

An average-case analysis usually requires that we calculate expected running times based on a given input distribution, which usually involves sophisticated probability theory. Therefore, for the remainder of this book, unless we specify otherwise, we characterize running times in terms of the *worst case*, as a function of the input size, n, of the algorithm.

Worst-case analysis is much easier than average-case analysis, as it requires only the ability to identify the worst-case input, which is often simple. Also, this approach typically leads to better algorithms. Making the standard of success for an algorithm to perform well in the worst case necessarily requires that it does well on *every* input. That is, designing for the worst case leads to stronger algorithmic "muscles," much like a track star who always practices by running up an incline.

4.2.3 Asymptotic Notation

In general, each basic step in a pseudo-code description or a high-level language implementation corresponds to a small number of primitive operations (except for function calls, of course). Thus, we can perform a simple analysis of an algorithm written in pseudo-code that estimates the number of primitive operations executed up to a constant factor, by pseudo-code steps (but we must be careful, since a single line of pseudo-code may denote a number of steps in some cases).

In algorithm analysis, we focus on the growth rate of the running time as a function of the input size n, taking a "big-picture" approach. It is often enough just to know that the running time of an algorithm such as arrayMax, shown in Code Fragment 4.1, *grows proportionally to* n, with its true running time being n times a constant factor that depends on the specific computer.

We analyze algorithms using a mathematical notation for functions that disregards constant factors. Namely, we characterize the running times of algorithms by using functions that map the size of the input, n, to values that correspond to the main factor that determines the growth rate in terms of n. This approach allows us to focus on the "big-picture" aspects of an algorithm's running time.

Algorithm arrayMax(A, n):
   Input: An array A storing $n \geq 1$ integers.
   Output: The maximum element in A.

  $currMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $currMax < A[i]$ **then**
 $currMax \leftarrow A[i]$
  return $currMax$

Code Fragment 4.1: Algorithm arrayMax.

The "Big-Oh" Notation

Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq cg(n), \quad \text{for} \quad n \geq n_0.$$

This definition is often referred to as the "big-Oh" notation, for it is sometimes pronounced as "$f(n)$ is **big-Oh** of $g(n)$." Alternatively, we can also say "$f(n)$ is **order of** $g(n)$." (This definition is illustrated in Figure 4.5.)
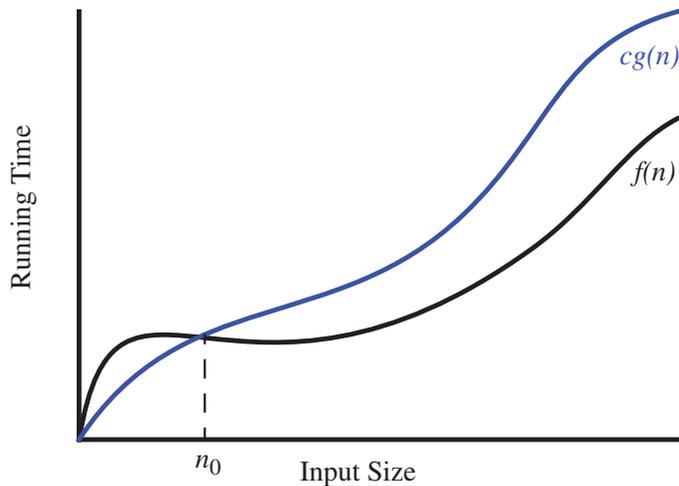
Figure 4.5: The "big-Oh" notation. The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Example 4.6: *The function $8n - 2$ is $O(n)$.*

Justification: By the big-Oh definition, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $8n - 2 \leq cn$ for every integer $n \geq n_0$. It is easy to see that a possible choice is $c = 8$ and $n_0 = 1$. Indeed, this is one of infinitely many choices available because any real number greater than or equal to 8 works for c, and any integer greater than or equal to 1 works for n_0. ■

The big-Oh notation allows us to say that a function $f(n)$ is "less than or equal to" another function $g(n)$ up to a constant factor and in the ***asymptotic*** sense as n grows toward infinity. This ability comes from the fact that the definition uses "$\leq$" to compare $f(n)$ to a $g(n)$ times a constant, c, for the asymptotic cases when $n \geq n_0$.

Characterizing Running Times using the Big-Oh Notation

The big-Oh notation is used widely to characterize running times and space bounds in terms of some parameter n, which varies from problem to problem, but is always defined as a chosen measure of the "size" of the problem. For example, if we are interested in finding the largest element in an array of integers, as in the arrayMax algorithm, we should let n denote the number of elements of the array. Using the big-Oh notation, we can write the following mathematically precise statement on the running time of algorithm arrayMax for ***any*** computer.

Proposition 4.7: *The Algorithm* arrayMax, *for computing the maximum element in an array of n integers, runs in $O(n)$ time.*

Justification: The number of primitive operations executed by algorithm array-Max in each iteration is a constant. Hence, since each primitive operation runs in constant time, we can say that the running time of algorithm arrayMax on an input of size n is at most a constant times n, that is, we may conclude that the running time of algorithm arrayMax is $O(n)$.   ■

Some Properties of the Big-Oh Notation

The big-Oh notation allows us to ignore constant factors and lower order terms and focus on the main components of a function that affect its growth.

Example 4.8: $5n^4 + 3n^3 + 2n^2 + 4n + 1$ *is $O(n^4)$.*

Justification: Note that $5n^4 + 3n^3 + 2n^2 + 4n + 1 \le (5+3+2+4+1)n^4 = cn^4$, for $c = 15$, when $n \ge n_0 = 1$.   ■

In fact, we can characterize the growth rate of any polynomial function.

Proposition 4.9: *If $f(n)$ is a polynomial of degree d, that is,*

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d,$$

and $a_d > 0$, then $f(n)$ is $O(n^d)$.

Justification: Note that, for $n \ge 1$, we have $1 \le n \le n^2 \le \cdots \le n^d$; hence,

$$a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d \le (a_0 + a_1 + a_2 + \cdots + a_d)n^d.$$

Therefore, we can show $f(n)$ is $O(n^d)$ by defining $c = a_0 + a_1 + \cdots + a_d$ and $n_0 = 1$.
                                     ■

Thus, the highest-degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial. We consider some additional properties of the big-Oh notation in the exercises. Let us consider some further examples here, however, focusing on combinations of the seven fundamental functions used in algorithm design.

Example 4.10: $5n^2 + 3n\log n + 2n + 5$ is $O(n^2)$.

Justification: $5n^2 + 3n\log n + 2n + 5 \leq (5 + 3 + 2 + 5)n^2 = cn^2$, for $c = 15$, when $n \geq n_0 = 2$ (note that $n\log n$ is zero for $n = 1$). ∎

Example 4.11: $20n^3 + 10n\log n + 5$ is $O(n^3)$.

Justification: $20n^3 + 10n\log n + 5 \leq 35n^3$, for $n \geq 1$. ∎

Example 4.12: $3\log n + 2$ is $O(\log n)$.

Justification: $3\log n + 2 \leq 5\log n$, for $n \geq 2$. Note that $\log n$ is zero for $n = 1$. That is why we use $n \geq n_0 = 2$ in this case. ∎

Example 4.13: 2^{n+2} is $O(2^n)$.

Justification: $2^{n+2} = 2^n 2^2 = 4 \cdot 2^n$; hence, we can take $c = 4$ and $n_0 = 1$ in this case. ∎

Example 4.14: $2n + 100\log n$ is $O(n)$.

Justification: $2n + 100\log n \leq 102n$, for $n \geq n_0 = 2$; hence, we can take $c = 102$ in this case. ∎

Characterizing Functions in Simplest Terms

In general, we should use the big-Oh notation to characterize a function as closely as possible. While it is true that the function $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or even $O(n^4)$, it is more accurate to say that $f(n)$ is $O(n^3)$. Consider, by way of analogy, a scenario where a hungry traveler driving along a long country road happens upon a local farmer walking home from a market. If the traveler asks the farmer how much longer he must drive before he can find some food, it may be truthful for the farmer to say, "certainly no longer than 12 hours," but it is much more accurate (and helpful) for him to say, "you can find a market just a few minutes drive up this road." Thus, even with the big-Oh notation, we should strive as much as possible to tell the whole truth.

It is also considered poor taste to include constant factors and lower order terms in the big-Oh notation. For example, it is not fashionable to say that the function $2n^2$ is $O(4n^2 + 6n\log n)$, although this is completely correct. We should strive instead to describe the function in the big-Oh in *simplest terms*.

The seven functions listed in Section 4.1 are the most common functions used in conjunction with the big-Oh notation to characterize the running times and space usage of algorithms. Indeed, we typically use the names of these functions to refer to the running times of the algorithms they characterize. So, for example, we would say that an algorithm that runs in worst-case time $4n^2 + n\log n$ is a *quadratic-time* algorithm, since it runs in $O(n^2)$ time. Likewise, an algorithm running in time at most $5n + 20\log n + 4$ would be called a *linear-time* algorithm.

Big-Omega

Just as the big-Oh notation provides an asymptotic way of saying that a function is "less than or equal to" another function, the following notations provide an asymptotic way of saying that a function grows at a rate that is "greater than or equal to" that of another.

Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. We say that $f(n)$ is $\Omega(g(n))$ (pronounced "$f(n)$ is big-Omega of $g(n)$") if $g(n)$ is $O(f(n))$, that is, there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \geq cg(n), \quad \text{for} \quad n \geq n_0.$$

This definition allows us to say asymptotically that one function is greater than or equal to another, up to a constant factor.

Example 4.15: $3n \log n + 2n$ is $\Omega(n \log n)$.

Justification: $3n \log n + 2n \geq 3n \log n$, for $n \geq 2$.   ∎

Big-Theta

In addition, there is a notation that allows us to say that two functions grow at the same rate, up to constant factors. We say that $f(n)$ is $\Theta(g(n))$ (pronounced "$f(n)$ is big-Theta of $g(n)$") if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$, that is, there are real constants $c' > 0$ and $c'' > 0$, and an integer constant $n_0 \geq 1$ such that

$$c'g(n) \leq f(n) \leq c''g(n), \quad \text{for} \quad n \geq n_0.$$

Example 4.16: $3n \log n + 4n + 5 \log n$ is $\Theta(n \log n)$.

Justification: $3n \log n \leq 3n \log n + 4n + 5 \log n \leq (3 + 4 + 5)n \log n$ for $n \geq 2$. ∎

4.2.4   Asymptotic Analysis

Suppose two algorithms solving the same problem are available: an algorithm A, which has a running time of $O(n)$, and an algorithm B, which has a running time of $O(n^2)$. Which algorithm is better? We know that n is $O(n^2)$, which implies that algorithm A is **asymptotically better** than algorithm B, although for a small value of n, B may have a lower running time than A.

We can use the big-Oh notation to order classes of functions by asymptotic growth rate. Our seven functions are ordered by increasing growth rate in the sequence below, that is, if a function $f(n)$ precedes a function $g(n)$ in the sequence, then $f(n)$ is $O(g(n))$:

$$1 \quad \log n \quad n \quad n \log n \quad n^2 \quad n^3 \quad 2^n.$$

We illustrate the growth rates of some important functions in Table 4.2.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Table 4.2: Selected values of fundamental functions in algorithm analysis.

We further illustrate the importance of the asymptotic viewpoint in Table 4.3. This table explores the maximum size allowed for an input instance that is processed by an algorithm in 1 second, 1 minute, and 1 hour. It shows the importance of good algorithm design, because an asymptotically slow algorithm is beaten in the long run by an asymptotically faster algorithm, even if the constant factor for the asymptotically faster algorithm is worse.

Running Time (μs)	*Maximum Problem Size (n)*		
	1 second	*1 minute*	*1 hour*
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Table 4.3: Maximum size of a problem that can be solved in 1 second, 1 minute, and 1 hour, for various running times measured in microseconds.

The importance of good algorithm design goes beyond just what can be solved effectively on a given computer, however. As shown in Table 4.4, even if we achieve a dramatic speed-up in hardware, we still cannot overcome the handicap of an asymptotically slow algorithm. This table shows the new maximum problem size achievable for any fixed amount of time, assuming algorithms with the given running times are now run on a computer 256 times faster than the previous one.

Running Time	*New Maximum Problem Size*
$400n$	$256m$
$2n^2$	$16m$
2^n	$m + 8$

Table 4.4: Increase in the maximum size of a problem that can be solved in a fixed amount of time by using a computer that is 256 times faster than the previous one. Each entry is a function of m, the previous maximum problem size.

4.2.5   Using the Big-Oh Notation

Having made the case of using the big-Oh notation for analyzing algorithms, let us briefly discuss a few issues concerning its use. It is considered poor taste, in general, to say "$f(n) \leq O(g(n))$," since the big-Oh already denotes the "less-than-or-equal-to" concept. Likewise, although common, it is not fully correct to say "$f(n) = O(g(n))$" (with the usual understanding of the "=" relation), since there is no way to make sense of the statement "$O(g(n)) = f(n)$." In addition, it is completely wrong to say "$f(n) \geq O(g(n))$" or "$f(n) > O(g(n))$," since the $g(n)$ in the big-Oh expresses an upper bound on $f(n)$. It is best to say,

> "$f(n)$ **is** $O(g(n))$."

For the more mathematically inclined, it is also correct to say,

> "$f(n) \in O(g(n))$,"

for the big-Oh notation is, technically speaking, denoting a whole collection of functions. In this book, we stick to presenting big-Oh statements as "$f(n)$ **is** $O(g(n))$." Even with this interpretation, there is considerable freedom in how we can use arithmetic operations with the big-Oh notation, and with this freedom comes a certain amount of responsibility.

Some Words of Caution

A few words of caution about asymptotic notation are in order at this point. First, note that the use of the big-Oh and related notations can be somewhat misleading should the constant factors they "hide" be very large. For example, while it is true that the function $10^{100}n$ is $O(n)$, if this is the running time of an algorithm being compared to one whose running time is $10n \log n$, we prefer the $O(n \log n)$ time algorithm, even though the linear-time algorithm is asymptotically faster. This preference is because the constant factor, 10^{100}, which is called "one googol," is believed by many astronomers to be an upper bound on the number of atoms in the observable universe. So we are unlikely to ever have a real-world problem that has this number as its input size. Thus, even when using the big-Oh notation, we should at least be somewhat mindful of the constant factors and lower order terms we are "hiding."

The observation above raises the issue of what constitutes a "fast" algorithm. Generally speaking, any algorithm running in $O(n \log n)$ time (with a reasonable constant factor) should be considered efficient. Even an $O(n^2)$ time method may be fast enough in some contexts, that is, when n is small. But an algorithm running in $O(2^n)$ time should almost never be considered efficient.

Exponential Running Times

There is a famous story about the inventor of the game of chess. He asked only that his king pay him 1 grain of rice for the first square on the board, 2 grains for the second, 4 grains for the third, 8 for the fourth, and so on. It is an interesting test of programming skills to write a program to compute exactly the number of grains of rice the king would have to pay. In fact, any C++ program written to compute this number in a single integer value causes an integer overflow to occur (although the run-time machine probably won't complain).

If we must draw a line between efficient and inefficient algorithms, therefore, it is natural to make this distinction be that between those algorithms running in polynomial time and those running in exponential time. That is, make the distinction between algorithms with a running time that is $O(n^c)$, for some constant $c > 1$, and those with a running time that is $O(b^n)$, for some constant $b > 1$. Like so many notions we have discussed in this section, this too should be taken with a "grain of salt," for an algorithm running in $O(n^{100})$ time should probably not be considered "efficient." Even so, the distinction between polynomial-time and exponential-time algorithms is considered a robust measure of tractability.

To summarize, the asymptotic notations of big-Oh, big-Omega, and big-Theta provide a convenient language for us to analyze data structures and algorithms. As mentioned earlier, these notations provide convenience because they let us concentrate on the "big picture" rather than low-level details.

Two Examples of Asymptotic Algorithm Analysis

We conclude this section by analyzing two algorithms that solve the same problem but have rather different running times. The problem we are interested in is the one of computing the so-called ***prefix averages*** of a sequence of numbers. Namely, given an array X storing n numbers, we want to compute an array A such that $A[i]$ is the average of elements $X[0],\ldots,X[i]$, for $i = 0,\ldots,n-1$, that is,

$$A[i] = \frac{\sum_{j=0}^{i} X[j]}{i+1}.$$

Computing prefix averages has many applications in economics and statistics. For example, given the year-by-year returns of a mutual fund, an investor typically wants to see the fund's average annual returns for the last year, the last three years, the last five years, and the last ten years. Likewise, given a stream of daily Web usage logs, a Web site manager may wish to track average usage trends over various time periods.

A Quadratic-Time Algorithm

Our first algorithm for the prefix averages problem, called prefixAverages1, is shown in Code Fragment 4.2. It computes every element of A separately, following the definition.

Algorithm prefixAverages1(X):
  Input: An n-element array X of numbers.
  Output: An n-element array A of numbers such that $A[i]$ is
     the average of elements $X[0], \ldots, X[i]$.
  Let A be an array of n numbers.
  for $i \leftarrow 0$ **to** $n-1$ **do**
     $a \leftarrow 0$
     for $j \leftarrow 0$ **to** i **do**
        $a \leftarrow a + X[j]$
     $A[i] \leftarrow a/(i+1)$
  return array A

Code Fragment 4.2: Algorithm prefixAverages1.

Let us analyze the prefixAverages1 algorithm.

- Initializing and returning array A at the beginning and end can be done with a constant number of primitive operations per element and takes $O(n)$ time.

- There are two nested **for** loops that are controlled by counters i and j, respectively. The body of the outer loop, controlled by counter i, is executed n times for $i = 0, \ldots, n-1$. Thus, statements $a = 0$ and $A[i] = a/(i+1)$ are executed n times each. This implies that these two statements, plus the incrementing and testing of counter i, contribute a number of primitive operations proportional to n, that is, $O(n)$ time.

- The body of the inner loop, which is controlled by counter j, is executed $i+1$ times, depending on the current value of the outer loop counter i. Thus, statement $a = a + X[j]$ in the inner loop is executed $1 + 2 + 3 + \cdots + n$ times. By recalling Proposition 4.3, we know that $1 + 2 + 3 + \cdots + n = n(n+1)/2$, which implies that the statement in the inner loop contributes $O(n^2)$ time. A similar argument can be done for the primitive operations associated with the incrementing and testing counter j, which also take $O(n^2)$ time.

The running time of algorithm prefixAverages1 is given by the sum of three terms. The first and the second term are $O(n)$, and the third term is $O(n^2)$. By a simple application of Proposition 4.9, the running time of prefixAverages1 is $O(n^2)$.

A Linear-Time Algorithm

In order to compute prefix averages more efficiently, we can observe that two consecutive averages $A[i-1]$ and $A[i]$ are similar:

$$\begin{aligned} A[i-1] &= (X[0]+X[1]+\cdots+X[i-1])/i \\ A[i] &= (X[0]+X[1]+\cdots+X[i-1]+X[i])/(i+1). \end{aligned}$$

If we denote with S_i the **prefix sum** $X[0]+X[1]+\cdots+X[i]$, we can compute the prefix averages as $A[i] = S_i/(i+1)$. It is easy to keep track of the current prefix sum while scanning array X with a loop. We are now ready to present Algorithm prefixAverages2 in Code Fragment 4.3.

Algorithm prefixAverages2(X):
    Input: An n-element array X of numbers.
    Output: An n-element array A of numbers such that $A[i]$ is
        the average of elements $X[0],\ldots,X[i]$.
  Let A be an array of n numbers.
  $s \leftarrow 0$
  for $i \leftarrow 0$ **to** $n-1$ **do**
    $s \leftarrow s + X[i]$
    $A[i] \leftarrow s/(i+1)$
  return array A

Code Fragment 4.3: Algorithm prefixAverages2.

The analysis of the running time of algorithm prefixAverages2 follows:

- Initializing and returning array A at the beginning and end can be done with a constant number of primitive operations per element, and takes $O(n)$ time.
- Initializing variable s at the beginning takes $O(1)$ time.
- There is a single **for** loop, which is controlled by counter i. The body of the loop is executed n times, for $i = 0,\ldots,n-1$. Thus, statements $s = s + X[i]$ and $A[i] = s/(i+1)$ are executed n times each. This implies that these two statements plus the incrementing and testing of counter i contribute a number of primitive operations proportional to n, that is, $O(n)$ time.

The running time of algorithm prefixAverages2 is given by the sum of three terms. The first and the third term are $O(n)$, and the second term is $O(1)$. By a simple application of Proposition 4.9, the running time of prefixAverages2 is $O(n)$, which is much better than the quadratic-time algorithm prefixAverages1.

4.2.6   A Recursive Algorithm for Computing Powers

As a more interesting example of algorithm analysis, let us consider the problem of raising a number x to an arbitrary nonnegative integer, n. That is, we wish to compute the **power function** $p(x,n)$, defined as $p(x,n) = x^n$. This function has an immediate recursive definition based on linear recursion:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{otherwise} \end{cases}$$

This definition leads immediately to a recursive algorithm that uses $O(n)$ function calls to compute $p(x,n)$. We can compute the power function much faster than this, however, by using the following alternative definition, also based on linear recursion, which employs a squaring technique:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, (n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x, n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

To illustrate how this definition works, consider the following examples:

$$\begin{aligned} 2^4 &= 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16 \\ 2^5 &= 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32 \\ 2^6 &= 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64 \\ 2^7 &= 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128 \end{aligned}$$

This definition suggests the algorithm of Code Fragment 4.4.

Algorithm Power(x,n):
   Input: A number x and integer $n \geq 0$
   Output: The value x^n
  if $n = 0$ **then**
    return 1
  if n is odd **then**
    $y \leftarrow$ Power$(x, (n-1)/2)$
    return $x \cdot y \cdot y$
  else
    $y \leftarrow$ Power$(x, n/2)$
    return $y \cdot y$

Code Fragment 4.4: Computing the power function using linear recursion.

To analyze the running time of the algorithm, we observe that each recursive call of function Power(x,n) divides the exponent, n, by two. Thus, there are $O(\log n)$ recursive calls, not $O(n)$. That is, by using linear recursion and the squaring technique, we reduce the running time for the computation of the power function from $O(n)$ to $O(\log n)$, which is a big improvement.

4.2.7   Some More Examples of Algorithm Analysis

Now that we have the big-Oh notation for doing algorithm analysis, let us give some more examples of simple algorithms that can have their running times characterized using this notation. Moreover, in keeping with our earlier promise, we illustrate below how each of the seven functions given earlier in this chapter can be used to characterize the running time of an example algorithm.

A Constant-Time Method

To illustrate a constant-time algorithm, consider the following C++ function, which returns the *size* of an STL vector, that is, the current number of cells in the array:

```cpp
int capacity(const vector<int>& arr) {
  return arr.size();
}
```

This is a very simple algorithm, because the size of a vector is stored as a member variable in the vector object, so it takes only a constant-time lookup to return this value. Thus, the capacity function runs in $O(1)$ time; that is, the running time of this function is independent of the value of n, the size of the array.

Revisiting the Method for Finding the Maximum in an Array

For our next example, let us reconsider a simple problem studied earlier, finding the largest value in an array of integers. We assume that the array is stored as an STL vector. This can be done in C++ as follows:

```cpp
int findMax(const vector<int>& arr) {
  int max = arr[0];
  for (int i = 1; i < arr.size(); i++) {
    if (max < arr[i]) max = arr[i];
  }
  return max;
}
```

This function, which amounts to a C++ implementation of the arrayMax algorithm of Section 4.2.3, compares each of the n elements in the input array to a current maximum, and each time it finds an element larger than the current maximum, it updates the current maximum to be this value. Thus, it spends a constant amount of time for each of the n elements in the array; hence, as with the pseudo-code version of the arrayMax algorithm, the running time of this algorithm is $O(n)$.

Further Analysis of the Maximum-Finding Algorithm

A more interesting question, with respect to the above maximum-finding algorithm, is to ask how many times we update the current maximum value. Note that this statement is executed only if we encounter a value of the array that is larger than our current maximum. In the worst case, this condition could be true each time we perform the test. For instance, this situation would occur if the input array is given to us in sorted order. Thus, in the worst-case, the statement $max = arr[i]$ is performed $n - 1$ times, hence $O(n)$ times.

But what if the input array is given to us in random order, with all orders equally likely; what would be the expected number of times we updated the maximum value in this case? To answer this question, note that we update the current maximum in the ith iteration only if the ith element in the array is bigger than all the elements that precede it. But if the array is given to us in random order, the probability that the ith element is larger than all elements that precede it is $1/i$; hence, the expected number of times we update the maximum in this case is $H_n = \sum_{i=1}^{n} 1/i$, which is known as the nth ***Harmonic number***. It turns out (see Proposition A.16) that H_n is $O(\log n)$. Therefore, the expected number of times the maximum is updated when the above maximum-finding algorithm is run on a random array is $O(\log n)$.

Three-Way Set Disjointness

Suppose we are given three sets, A, B, and C, with these sets stored in three different integer arrays, a, b, and c, respectively. The ***three-way set disjointness*** problem is to determine if these three sets are disjoint, that is, whether there is no element x such that $x \in A$, $x \in B$, and $x \in C$. A simple C++ function to determine this property is given below:

```cpp
bool areDisjoint(const vector<int>& a, const vector<int>& b,
                 const vector<int>& c) {
  for (int i = 0; i < a.size(); i++)
    for (int j = 0; j < b.size(); j++)
      for (int k = 0; k < c.size(); k++)
        if ((a[i] == b[j]) && (b[j] == c[k])) return false;
  return true;
}
```

This simple algorithm loops through each possible triple of indices i, j, and k to check if the respective elements indexed in a, b, and c are equal. Thus, if each of these arrays is of size n, then the worst-case running time of this function is $O(n^3)$. Moreover, the worst case is achieved when the sets are disjoint, since in this case we go through all n^3 triples of valid indices, i, j, and k. Such a running time would generally not be considered very efficient, but, fortunately, there is a better way to solve this problem, which we explore in Exercise C-4.3.

Recursion Run Amok

The next few example algorithms we study are for solving the ***element uniqueness problem***, in which we are given a range, $i, i+1, \ldots, j$, of indices for an array, A, which we assume is given as an STL vector. We want to determine if the elements of this range, $A[i], A[i+1], \ldots, A[j]$, are all unique, that is, there is no repeated element in this group of array entries. The first algorithm we give for solving the element uniqueness problem is a recursive one. But it uses recursion in a very inefficient manner, as shown in the following C++ implementation.

```
bool isUnique(const vector<int>& arr, int start, int end) {
  if (start >= end) return true;
  if (!isUnique(arr, start, end−1))
    return false;
  if (!isUnique(arr, start+1, end))
    return false;
  return (arr[start] != arr[end]);
}
```

You should first convince yourself that the function is correct. To analyze this recursive algorithm's running time, let us first determine how much time we spend outside of recursive calls in any invocation of this function. Note, in particular, that there are no loops—just comparisons, arithmetic operations, array element references, and function returns. Thus, the nonrecursive part of each function invocation runs in constant time, that is, $O(1)$ time; hence, to determine the worst-case running time of this function we only need to determine the worst-case total number of calls we make to the isUnique function.

Let n denote the number of entries under consideration, that is, let

$$n = end - start + 1.$$

If $n = 1$, then the running time of the isUnique is $O(1)$, since there are no recursive calls for this case. To characterize the running time of the general case, the important observation to make is that in order to solve a problem of size n, the isUnique function makes two recursive calls on problems of size $n-1$. Thus, in the worst case, a call for a range of size n makes two calls on ranges of size $n-1$, which each make two calls on ranges of size $n-2$, which each make two calls on ranges of size $n-3$, and so on. Thus, in the worst case, the total number of function calls is given by the geometric summation

$$1 + 2 + 4 + \cdots + 2^{n-1},$$

which is equal to $2^n - 1$ by Proposition 4.5. Thus, the worst-case running time of function isUnique is $O(2^n)$. This is an incredibly inefficient method for solving the element uniqueness problem. Its inefficiency comes not from the fact that it uses recursion—it comes from the fact that it uses recursion poorly, which is something we address in Exercise C-4.2.

An Iterative Method for Solving the Element Uniqueness Problem

We can do much better than the above exponential-time method by using the following iterative algorithm:

```cpp
bool isUniqueLoop(const vector<int>& arr, int start, int end) {
  if (start >= end) return true;
  for (int i = start; i < end; i++)
    for (int j = i+1; j <= end; j++)
      if (arr[i] == arr[j]) return false;
  return true;
}
```

This function solves the element uniqueness problem by looping through all distinct pairs of indices, i and j, and checking if any of them indexes a pair of elements that are equal to each other. It does this using two nested **for** loops, such that the first iteration of the outer loop causes $n-1$ iterations of the inner loop, the second iteration of the outer loop causes $n-2$ iterations of the inner loop, the third iteration of the outer loop causes $n-3$ iterations of the inner loop, and so on. Thus, the worst-case running time of this function is proportional to

$$1+2+3+\cdots+(n-1),$$

which is $O(n^2)$ as we saw earlier in this chapter (Proposition 4.3).

Using Sorting as a Problem-Solving Tool

An even better algorithm for the element uniqueness problem is based on using sorting as a problem-solving tool. In this case, by sorting an array of elements, we are guaranteed that any duplicate elements will be placed next to each other. Thus, it suffices to sort the array and look for duplicates among consecutive elements. A C++ implementation of this algorithm follows.

```cpp
bool isUniqueSort(const vector<int>& arr, int start, int end) {
  if (start >= end) return true;
  vector<int> buf(arr);                        // duplicate copy of arr
  sort(buf.begin()+start, buf.begin()+end);    // sort the subarray
  for (int i = start; i < end; i++)            // check for duplicates
    if (buf[i] == buf[i+1]) return false;
  return true;
}
```

The function sort is provided by the STL. On most systems, it runs in $O(n\log n)$ time. Since the other steps run in $O(n)$ time, the entire algorithm runs in $O(n\log n)$ time. Incidentally, we can solve the element uniqueness problem even faster, at least in terms of its average-case running time, by using the hash table data structure we explore in Section 9.2.

4.3 Simple Justification Techniques

Sometimes, we want to make claims about an algorithm, such as showing that it is correct or that it runs fast. In order to rigorously make such claims, we must use mathematical language, and in order to back up such claims, we must justify or **prove** our statements. Fortunately, there are several simple ways to do this.

4.3.1 By Example

Some claims are of the generic form, "There is an element x in a set S that has property P." To justify such a claim, we only need to produce a particular x in S that has property P. Likewise, some hard-to-believe claims are of the generic form, "Every element x in a set S has property P." To justify that such a claim is false, we only need to produce a particular x from S that does not have property P. Such an instance is called a **counterexample**.

Example 4.17: *Professor Amongus claims that every number of the form $2^i - 1$ is a prime, when i is an integer greater than 1. Professor Amongus is wrong.*

Justification: To prove Professor Amongus is wrong, we find a counter-example. Fortunately, we need not look too far, for $2^4 - 1 = 15 = 3 \cdot 5$. ∎

4.3.2 The "Contra" Attack

Another set of justification techniques involves the use of the negative. The two primary such methods are the use of the **contrapositive** and the **contradiction**. The use of the contrapositive method is like looking through a negative mirror. To justify the statement "if p is true, then q is true" we establish that "if q is not true, then p is not true" instead. Logically, these two statements are the same, but the latter, which is called the **contrapositive** of the first, may be easier to think about.

Example 4.18: *Let a and b be integers. If ab is even, then a is even or b is even.*

Justification: To justify this claim, consider the contrapositive, "If a is odd and b is odd, then ab is odd." So, suppose $a = 2i + 1$ and $b = 2j + 1$, for some integers i and j. Then $ab = 4ij + 2i + 2j + 1 = 2(2ij + i + j) + 1$; hence, ab is odd. ∎

Besides showing a use of the contrapositive justification technique, the previous example also contains an application of **DeMorgan's Law**. This law helps us deal with negations, for it states that the negation of a statement of the form "p or q" is "not p and not q." Likewise, it states that the negation of a statement of the form "p and q" is "not p or not q."

Contradiction

Another negative justification technique is justification by ***contradiction***, which also often involves using DeMorgan's Law. In applying the justification by contradiction technique, we establish that a statement q is true by first supposing that q is false and then showing that this assumption leads to a contradiction (such as $2 \neq 2$ or $1 > 3$). By reaching such a contradiction, we show that no consistent situation exists with q being false, so q must be true. Of course, in order to reach this conclusion, we must be sure our situation is consistent before we assume q is false.

Example 4.19: *Let a and b be integers. If ab is odd, then a is odd and b is odd.*

Justification: Let ab be odd. We wish to show that a is odd and b is odd. So, with the hope of leading to a contradiction, let us assume the opposite, namely, suppose a is even or b is even. In fact, without loss of generality, we can assume that a is even (since the case for b is symmetric). Then $a = 2i$ for some integer i. Hence, $ab = (2i)b = 2(ib)$, that is, ab is even. But this is a contradiction: ab cannot simultaneously be odd and even. Therefore a is odd and b is odd. ∎

4.3.3 Induction and Loop Invariants

Most of the claims we make about a running time or a space bound involve an integer parameter n (usually denoting an intuitive notion of the "size" of the problem). Moreover, most of these claims are equivalent to saying some statement $q(n)$ is true "for all $n \geq 1$." Since this is making a claim about an infinite set of numbers, we cannot justify this exhaustively in a direct fashion.

Induction

We can often justify claims such as those above as true, however, by using the technique of ***induction***. This technique amounts to showing that, for any particular $n \geq 1$, there is a finite sequence of implications that starts with something known to be true and ultimately leads to showing that $q(n)$ is true. Specifically, we begin a justification by induction by showing that $q(n)$ is true for $n = 1$ (and possibly some other values $n = 2, 3, \ldots, k$, for some constant k). Then we justify that the inductive "step" is true for $n > k$, namely, we show "if $q(i)$ is true for $i < n$, then $q(n)$ is true." The combination of these two pieces completes the justification by induction.

Proposition 4.20: *Consider the Fibonacci function $F(n)$, where we define $F(1) = 1$, $F(2) = 2$, and $F(n) = F(n-1) + F(n-2)$ for $n > 2$. (See Section 2.2.3.) We claim that $F(n) < 2^n$.*

Justification: We show our claim is right by induction.
Base cases: *($n \leq 2$).* $F(1) = 1 < 2 = 2^1$ and $F(2) = 2 < 4 = 2^2$.
Induction step: *($n > 2$).* Suppose our claim is true for $n' < n$. Consider $F(n)$. Since $n > 2$, $F(n) = F(n-1) + F(n-2)$. Moreover, since $n - 1 < n$ and $n - 2 < n$, we can apply the inductive assumption (sometimes called the "inductive hypothesis") to imply that $F(n) < 2^{n-1} + 2^{n-2}$, since

$$2^{n-1} + 2^{n-2} < 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n.$$

■

Let us do another inductive argument, this time for a fact we have seen before.

Proposition 4.21: *(which is the same as Proposition 4.3)*

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Justification: We justify this equality by induction.
Base case: $n = 1$. Trivial, for $1 = n(n+1)/2$, if $n = 1$.
Induction step: $n \geq 2$. Assume the claim is true for $n' < n$. Consider n.

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i.$$

By the induction hypothesis, then

$$\sum_{i=1}^{n} i = n + \frac{(n-1)n}{2},$$

which we can simplify as

$$n + \frac{(n-1)n}{2} = \frac{2n + n^2 - n}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2}.$$

■

We may sometimes feel overwhelmed by the task of justifying something true for *all* $n \geq 1$. We should remember, however, the concreteness of the inductive technique. It shows that, for any particular n, there is a finite step-by-step sequence of implications that starts with something true and leads to the truth about n. In short, the inductive argument is a formula for building a sequence of direct justifications.

Loop Invariants

The final justification technique we discuss in this section is the ***loop invariant***. To prove some statement S about a loop is correct, define S in terms of a series of smaller statements $S_0, S_1, \ldots, S_k$, where:

1. The ***initial*** claim, S_0, is true before the loop begins.
2. If S_{i-1} is true before iteration i, then S_i is true after iteration i.
3. The final statement, S_k, implies the statement S that we wish to be true.

Let us give a simple example of using a loop-invariant argument to justify the correctness of an algorithm. In particular, let us consider using a loop invariant to justify the correctness of arrayFind, shown in Code Fragment 4.5, for finding an element x in an array A.

Algorithm arrayFind(x, A):
    Input: An element x and an n-element array, A.
    Output: The index i such that $x = A[i]$ or -1 if no element of A is equal to x.
    $i \leftarrow 0$
    while $i < n$ **do**
      if $x = A[i]$ **then**
        return i
      else
        $i \leftarrow i + 1$
    return -1

Code Fragment 4.5: Algorithm arrayFind for finding a given element in an array.

To show that arrayFind is correct, we inductively define a series of statements, S_i, that lead to the correctness of our algorithm. Specifically, we claim the following is true at the beginning of iteration i of the **while** loop:
    S_i: x is not equal to any of the first i elements of A.
This claim is true at the beginning of the first iteration of the loop, since there are no elements among the first 0 in A (this kind of a trivially true claim is said to hold ***vacuously***). In iteration i, we compare element x to element $A[i]$ and return the index i if these two elements are equal, which is clearly correct and completes the algorithm in this case. If the two elements x and $A[i]$ are not equal, then we have found one more element not equal to x and we increment the index i. Thus, the claim S_i is true for this new value of i; hence, it is true at the beginning of the next iteration. If the while-loop terminates without ever returning an index in A, then we have $i = n$. That is, S_n is true—there are no elements of A equal to x. Therefore, the algorithm correctly returns -1 to indicate that x is not in A.

4.4 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-4.1 There is a well-known city (which will go nameless here) whose inhabitants have the reputation of enjoying a meal only if that meal is the best they have ever experienced in their life. Otherwise, they hate it. Assuming meal quality is distributed uniformly across a person's life, what is the expected number of times inhabitants of this city are happy with their meals?

R-4.2 Give a pseudo-code description of the $O(n)$-time algorithm for computing the power function $p(x,n)$. Also, draw the recursion trace of this algorithm for the computation of $p(2,5)$.

R-4.3 Give a C++ description of Algorithm Power for computing the power function $p(x,n)$ (Code Fragment 4.4).

R-4.4 Draw the recursion trace of the Power algorithm (Code Fragment 4.4, which computes the power function $p(x,n)$) for computing $p(2,9)$.

R-4.5 Analyze the running time of Algorithm BinarySum (Code Fragment 3.41) for arbitrary values of the input parameter n.

R-4.6 Graph the functions $8n$, $4n\log n$, $2n^2$, n^3, and 2^n using a logarithmic scale for the x- and y-axes. That is, if the function is $f(n)$ is y, plot this as a point with x-coordinate at $\log n$ and y-coordinate at $\log y$.

R-4.7 The number of operations executed by algorithms A and B is $8n\log n$ and $2n^2$, respectively. Determine n_0 such that A is better than B for $n \geq n_0$.

R-4.8 The number of operations executed by algorithms A and B is $40n^2$ and $2n^3$, respectively. Determine n_0 such that A is better than B for $n \geq n_0$.

R-4.9 Give an example of a function that is plotted the same on a log-log scale as it is on a standard scale.

R-4.10 Explain why the plot of the function n^c is a straight line with slope c on a log-log scale.

R-4.11 What is the sum of all the even numbers from 0 to $2n$, for any positive integer n?

R-4.12 Show that the following two statements are equivalent:
(a) The running time of algorithm A is always $O(f(n))$.
(b) In the worst case, the running time of algorithm A is $O(f(n))$.

R-4.13 Order the following functions by asymptotic growth rate.

$$4n\log n + 2n \quad 2^{10} \quad 2^{\log n}$$
$$3n + 100\log n \quad 4n \quad 2^n$$
$$n^2 + 10n \quad n^3 \quad n\log n$$

R-4.14 Show that if $d(n)$ is $O(f(n))$, then $ad(n)$ is $O(f(n))$, for any constant $a > 0$.

R-4.15 Show that if $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then the product $d(n)e(n)$ is $O(f(n)g(n))$.

R-4.16 Give a big-Oh characterization, in terms of n, of the running time of the Ex1 function shown in Code Fragment 4.6.

R-4.17 Give a big-Oh characterization, in terms of n, of the running time of the Ex2 function shown in Code Fragment 4.6.

R-4.18 Give a big-Oh characterization, in terms of n, of the running time of the Ex3 function shown in Code Fragment 4.6.

R-4.19 Give a big-Oh characterization, in terms of n, of the running time of the Ex4 function shown in Code Fragment 4.6.

R-4.20 Give a big-Oh characterization, in terms of n, of the running time of the Ex5 function shown in Code Fragment 4.6.

R-4.21 Bill has an algorithm, find2D, to find an element x in an $n \times n$ array A. The algorithm find2D iterates over the rows of A, and calls the algorithm arrayFind, of Code Fragment 4.5, on each row, until x is found or it has searched all rows of A. What is the worst-case running time of find2D in terms of n? What is the worst-case running time of find2D in terms of N, where N is the total size of A? Would it be correct to say that Find2D is a linear-time algorithm? Why or why not?

R-4.22 For each function $f(n)$ and time t in the following table, determine the largest size n of a problem P that can be solved in time t if the algorithm for solving P takes $f(n)$ microseconds (one entry is already completed).

	1 Second	1 Hour	1 Month	1 Century
$\log n$	$\approx 10^{300000}$			
n				
$n\log n$				
n^2				
2^n				

R-4.23 Show that if $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n) + e(n)$ is $O(f(n) + g(n))$.

Algorithm Ex1(A):

  Input: An array A storing $n \geq 1$ integers.

  Output: The sum of the elements in A.

 $s \leftarrow A[0]$

 for $i \leftarrow 1$ **to** $n - 1$ **do**

  $s \leftarrow s + A[i]$

 return s

Algorithm Ex2(A):

  Input: An array A storing $n \geq 1$ integers.

  Output: The sum of the elements at even cells in A.

 $s \leftarrow A[0]$

 for $i \leftarrow 2$ **to** $n - 1$ **by** increments of 2 **do**

  $s \leftarrow s + A[i]$

 return s

Algorithm Ex3(A):

  Input: An array A storing $n \geq 1$ integers.

  Output: The sum of the prefix sums in A.

 $s \leftarrow 0$

 for $i \leftarrow 0$ **to** $n - 1$ **do**

  $s \leftarrow s + A[0]$

  for $j \leftarrow 1$ **to** i **do**

   $s \leftarrow s + A[j]$

 return s

Algorithm Ex4(A):

  Input: An array A storing $n \geq 1$ integers.

  Output: The sum of the prefix sums in A.

 $s \leftarrow A[0]$

 $t \leftarrow s$

 for $i \leftarrow 1$ **to** $n - 1$ **do**

  $s \leftarrow s + A[i]$

  $t \leftarrow t + s$

 return t

Algorithm Ex5(A, B):

  Input: Arrays A and B each storing $n \geq 1$ integers.

  Output: The number of elements in B equal to the sum of prefix sums in A.

 $c \leftarrow 0$

 for $i \leftarrow 0$ **to** $n - 1$ **do**

  $s \leftarrow 0$

  for $j \leftarrow 0$ **to** $n - 1$ **do**

   $s \leftarrow s + A[0]$

   for $k \leftarrow 1$ **to** j **do**

    $s \leftarrow s + A[k]$

  if $B[i] = s$ **then**

   $c \leftarrow c + 1$

 return c

Code Fragment 4.6: Some algorithms.

R-4.24 Show that if $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n) - e(n)$ is *not necessarily* $O(f(n) - g(n))$.

R-4.25 Show that if $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.

R-4.26 Show that $O(\max\{f(n), g(n)\}) = O(f(n) + g(n))$.

R-4.27 Show that $f(n)$ is $O(g(n))$ if and only if $g(n)$ is $\Omega(f(n))$.

R-4.28 Show that if $p(n)$ is a polynomial in n, then $\log p(n)$ is $O(\log n)$.

R-4.29 Show that $(n+1)^5$ is $O(n^5)$.

R-4.30 Show that 2^{n+1} is $O(2^n)$.

R-4.31 Show that n is $O(n \log n)$.

R-4.32 Show that n^2 is $\Omega(n \log n)$.

R-4.33 Show that $n \log n$ is $\Omega(n)$.

R-4.34 Show that $\lceil f(n) \rceil$ is $O(f(n))$, if $f(n)$ is a positive nondecreasing function that is always greater than 1.

R-4.35 Algorithm A executes an $O(\log n)$-time computation for each entry of an n-element array. What is the worst-case running time of Algorithm A?

R-4.36 Given an n-element array X, Algorithm B chooses $\log n$ elements in X at random and executes an $O(n)$-time calculation for each. What is the worst-case running time of Algorithm B?

R-4.37 Given an n-element array X of integers, Algorithm C executes an $O(n)$-time computation for each even number in X, and an $O(\log n)$-time computation for each odd number in X. What are the best-case and worst-case running times of Algorithm C?

R-4.38 Given an n-element array X, Algorithm D calls Algorithm E on each element $X[i]$. Algorithm E runs in $O(i)$ time when it is called on element $X[i]$. What is the worst-case running time of Algorithm D?

R-4.39 Al and Bob are arguing about their algorithms. Al claims his $O(n \log n)$-time method is *always* faster than Bob's $O(n^2)$-time method. To settle the issue, they perform a set of experiments. To Al's dismay, they find that if $n < 100$, the $O(n^2)$-time algorithm runs faster, and only when $n \geq 100$ is the $O(n \log n)$-time one better. Explain how this is possible.

Creativity

C-4.1 Describe a recursive algorithm to compute the integer part of the base-2 logarithm of n using only addition and integer division.

C-4.2 Describe an efficient recursive method for solving the element uniqueness problem, which runs in time that is at most $O(n^2)$ in the worst case without using sorting.

C-4.3 Assuming it is possible to sort n numbers in $O(n \log n)$ time, show that it is possible to solve the three-way set disjointness problem in $O(n \log n)$ time.

C-4.4 Describe an efficient algorithm for finding the 10 largest elements in an array of size n. What is the running time of your algorithm?

C-4.5 Suppose you are given an n-element array A containing distinct integers that are listed in increasing order. Given a number k, describe a recursive algorithm to find two integers in A that sum to k, if such a pair exists. What is the running time of your algorithm?

C-4.6 Given an n-element unsorted array A of n integers and an integer k, describe a recursive algorithm for rearranging the elements in A so that all elements less than or equal to k come before any elements larger than k. What is the running time of your algorithm?

C-4.7 Communication security is extremely important in computer networks, and one way many network protocols achieve security is to encrypt messages. Typical *cryptographic* schemes for the secure transmission of messages over such networks are based on the fact that no efficient algorithms are known for factoring large integers. Hence, if we can represent a secret message by a large prime number p, we can transmit, over the network, the number $r = p \cdot q$, where $q > p$ is another large prime number that acts as the *encryption key*. An eavesdropper who obtains the transmitted number r on the network would have to factor r in order to figure out the secret message p.

Using factoring to figure out a message is very difficult without knowing the encryption key q. To understand why, consider the following naive factoring algorithm:

> **for** $p = 2, \cdots, r - 1$ **do**
>   **if** p divides r **then**
>   **return** "The secret message is p!"

  a. Suppose that the eavesdropper uses the above algorithm and has a computer that can carry out in 1 microsecond (1 millionth of a second) a division between two integers of up to 100 bits each. Give an estimate of the time that it will take in the worst case to decipher the secret message p if the transmitted message r has 100 bits.

    b. What is the worst-case time complexity of the above algorithm? Since the input to the algorithm is just one large number r, assume that the input size n is the number of bytes needed to store r, that is, $n = \lfloor (\log_2 r)/8 \rfloor + 1$, and that each division takes time $O(n)$.

C-4.8 Give an example of a positive function $f(n)$ such that $f(n)$ is neither $O(n)$ nor $\Omega(n)$.

C-4.9 Show that $\sum_{i=1}^{n} i^2$ is $O(n^3)$.

C-4.10 Show that $\sum_{i=1}^{n} i/2^i < 2$.

(Hint: Try to bound this sum term by term with a geometric progression.)

C-4.11 Show that $\log_b f(n)$ is $\Theta(\log f(n))$ if $b > 1$ is a constant.

C-4.12 Describe a method for finding both the minimum and maximum of n numbers using fewer than $3n/2$ comparisons.

(Hint: First construct a group of candidate minimums and a group of candidate maximums.)

C-4.13 Bob built a Web site and gave the URL only to his n friends, which he numbered from 1 to n. He told friend number i that he/she can visit the Web site at most i times. Now Bob has a counter, C, keeping track of the total number of visits to the site (but not the identities of who visits). What is the minimum value for C such that Bob should know that one of his friends has visited his/her maximum allowed number of times?

C-4.14 Al says he can prove that all sheep in a flock are the same color:

Base case: One sheep. It is clearly the same color as itself.

Induction step: A flock of n sheep. Take a sheep, a, out. The remaining $n-1$ are all the same color by induction. Now put sheep a back in and take out a different sheep, b. By induction, the $n-1$ sheep (now with a) are all the same color. Therefore, all the sheep in the flock are the same color.

What is wrong with Al's "justification"?

C-4.15 Consider the following "justification" that the Fibonacci function, $F(n)$ (see Proposition 4.20) is $O(n)$:

Base case ($n \le 2$): $F(1) = 1$ and $F(2) = 2$.

Induction step ($n > 2$): Assume the claim true for $n' < n$. Consider n. $F(n) = F(n-1) + F(n-2)$. By induction, $F(n-1)$ is $O(n-1)$ and $F(n-2)$ is $O(n-2)$. Then, $F(n)$ is $O((n-1)+(n-2))$, by the identity presented in Exercise R-4.23. Therefore, $F(n)$ is $O(n)$.

What is wrong with this "justification"?

C-4.16 Let $p(x)$ be a polynomial of degree n, that is, $p(x) = \sum_{i=0}^{n} a_i x^i$.

(a) Describe a simple $O(n^2)$ time method for computing $p(x)$.

(b) Now consider a rewriting of $p(x)$ as

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + x a_n) \cdots))),$$

which is known as ***Horner's method***. Using the big-Oh notation, characterize the number of arithmetic operations this method executes.

C-4.17 Consider the Fibonacci function, $F(n)$ (see Proposition 4.20). Show by induction that $F(n)$ is $\Omega((3/2)^n)$.

C-4.18 Given a set $A = \{a_1, a_2, \ldots, a_n\}$ of n integers, describe, in pseudo-code, an efficient method for computing each of partial sums $s_k = \sum_{i=1}^{k} a_i$, for $k = 1, 2, \ldots, n$. What is the running time of this method?

C-4.19 Draw a visual justification of Proposition 4.3 analogous to that of Figure 4.1(b) for the case when n is odd.

C-4.20 An array A contains $n-1$ unique integers in the range $[0, n-1]$, that is, there is one number from this range that is not in A. Design an $O(n)$-time algorithm for finding that number. You are only allowed to use $O(1)$ additional space besides the array A itself.

C-4.21 Let S be a set of n lines in the plane such that no two are parallel and no three meet in the same point. Show, by induction, that the lines in S determine $\Theta(n^2)$ intersection points.

C-4.22 Show that the summation $\sum_{i=1}^{n} \lceil \log_2 i \rceil$ is $O(n \log n)$.

C-4.23 An evil king has n bottles of wine, and a spy has just poisoned one of them. Unfortunately, they don't know which one it is. The poison is very deadly; just one drop diluted even a billion to one will still kill. Even so, it takes a full month for the poison to take effect. Design a scheme for determining exactly which one of the wine bottles was poisoned in just one month's time while expending $O(\log n)$ taste testers.

C-4.24 An array A contains n integers taken from the interval $[0, 4n]$, with repetitions allowed. Describe an efficient algorithm for determining an integer value k that occurs the most often in A. What is the running time of your algorithm?

C-4.25 Describe, in pseudo-code, a method for multiplying an $n \times m$ matrix A and an $m \times p$ matrix B. Recall that the product $C = AB$ is defined so that $C[i][j] = \sum_{k=1}^{m} A[i][k] \cdot B[k][j]$. What is the running time of your method?

C-4.26 Suppose each row of an $n \times n$ array A consists of 1's and 0's such that, in any row i of A, all the 1's come before any 0's. Also suppose that the number of 1's in row i is at least the number in row $i+1$, for $i = 0, 1, \ldots, n-2$. Assuming A is already in memory, describe a method running in $O(n)$ time (not $O(n^2)$) for counting the number of 1's in A.

C-4.27 Describe a recursive function for computing the nth ***Harmonic number***, $H_n = \sum_{i=1}^{n} 1/i$.

Projects

P-4.1 Implement prefixAverages1 and prefixAverages2 from Section 4.2.5, and perform an experimental analysis of their running times. Visualize their running times as a function of the input size with a log-log chart.

P-4.2 Perform a careful experimental analysis that compares the relative running times of the functions shown in Code Fragments 4.6.

P-4.3 Perform an experimental analysis to test the hypothesis that the STL function, sort, runs in $O(n \log n)$ time on average.

P-4.4 Perform an experimental analysis to determine the largest value of n for each of the three algorithms given in the chapter for solving the element uniqueness problem such that the given algorithm runs in one minute or less.

Chapter Notes

The big-Oh notation has prompted several comments about its proper use [15, 43, 58]. Knuth [59, 58] defines it using the notation $f(n) = O(g(n))$, but says this "equality" is only "one way." We have chosen to take a more standard view of equality and view the big-Oh notation as a set, following Brassard [15]. The reader interested in studying average-case analysis is referred to the book chapter by Vitter and Flajolet [101]. We found the story about Archimedes in [78]. For some additional mathematical tools, please refer to Appendix A.

Chapter

5

Stacks, Queues, and Deques

Contents

5.1   Stacks

A *stack* is a container of objects that are inserted and removed according to the ***last-in first-out*** (***LIFO***) principle. Objects can be inserted into a stack at any time, but only the most recently inserted (that is, "last") object can be removed at any time. The name "stack" is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser. In this case, the fundamental operations involve the "pushing" and "popping" of plates on the stack. When we need a new plate from the dispenser, we "pop" the top plate off the stack, and when we add a plate, we "push" it down on the stack to become the new top plate. Perhaps an even more amusing metaphor would be a PEZ® candy dispenser, which stores mint candies in a spring-loaded container that "pops" out the top-most candy in the stack when the top of the dispenser is lifted. (See Figure 5.1.) Stacks are a fundamental data structure. They are used in many applications, including the following.

Example 5.1: *Internet Web browsers store the addresses of recently visited sites on a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.*

Example 5.2: *Text editors usually provide an "undo" mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.*
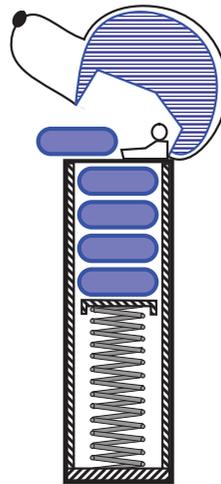
Figure 5.1: A schematic drawing of a PEZ® dispenser; a physical implementation of the stack ADT. (PEZ® is a registered trademark of PEZ Candy, Inc.)

5.1.1 The Stack Abstract Data Type

Stacks are the simplest of all data structures, yet they are also among the most important, since they are used in a host of different applications that include many more sophisticated data structures. Formally, a stack is an abstract data type (ADT) that supports the following operations:

push(e): Insert element e at the top of the stack.

pop(): Remove the top element from the stack; an error occurs if the stack is empty.

top(): Return a reference to the top element on the stack, without removing it; an error occurs if the stack is empty.

Additionally, let us also define the following supporting functions:

size(): Return the number of elements in the stack.

empty(): Return true if the stack is empty and false otherwise.

Example 5.3: *The following table shows a series of stack operations and their effects on an initially empty stack of integers.*

Operation	*Output*	*Stack Contents*
push(5)	–	(5)
push(3)	–	(5, 3)
pop()	–	(5)
push(7)	–	(5, 7)
pop()	–	(5)
top()	5	(5)
pop()	–	()
pop()	*"error"*	()
top()	*"error"*	()
empty()	*true*	()
push(9)	–	(9)
push(7)	–	(9, 7)
push(3)	–	(9, 7, 3)
push(5)	–	(9, 7, 3, 5)
size()	4	(9, 7, 3, 5)
pop()	–	(9, 7, 3)
push(8)	–	(9, 7, 3, 8)
pop()	–	(9, 7, 3)
top()	3	(9, 7, 3)

5.1.2   The STL Stack

The Standard Template Library provides an implementation of a stack. The un-
derlying implementation is based on the STL vector class, which is presented in
Sections 1.5.5 and 6.1.4. In order to declare an object of type stack, it is neces-
sary to first include the definition file, which is called "stack." As with the STL
vector, the class stack is part of the std namespace, so it is necessary either to use
"std::stack" or to provide a "**using**" statement. The stack class is templated with
the class of the individual elements. For example, the code fragment below declares
a stack of integers.

```
#include <stack>
using std::stack;                    // make stack accessible
stack<int> myStack;                  // a stack of integers
```

We refer to the type of individual elements as the stack's ***base type***. As with STL
vectors, an STL stack dynamically resizes itself as new elements are pushed on.

The STL stack class supports the same operators as our interface. Below, we
list the principal member functions. Let *s* be declared to be an STL vector, and let
e denote a single object whose type is the same as the base type of the stack. (For
example, *s* is a vector of integers, and *e* is an integer.)

size(): Return the number of elements in the stack.

empty(): Return true if the stack is empty and false otherwise.

push(*e*): Push *e* onto the top of the stack.

pop(): Pop the element at the top of the stack.

top(): Return a reference to the element at the top of the stack.

There is one significant difference between the STL implementation and our
own definitions of the stack operations. In the STL implementation, the result of
applying either of the operations top or pop to an empty stack is undefined. In
particular, no exception is thrown. Even though no exception is thrown, it may
very likely result in your program aborting. Thus, it is up to the programmer to be
sure that no such illegal accesses are attempted.

5.1.3   A C++ Stack Interface

Before discussing specific implementations of the stack, let us first consider how
to define an abstract data type for a stack. When defining an abstract data type,
our principal concern is specifying the ***Application Programming Interface*** (API),
or simply ***interface***, which describes the names of the public members that the
ADT must support and how they are to be declared and used. An interface is not
a complete description of all the public members. For example, it does not include

the private data members. Rather, it is a list of members that any implementation must provide. The C++ programming language does not provide a simple method for defining interfaces, and therefore, the interface defined here is not an official C++ class. It is offered principally for the purpose of illustration.

The informal interface for the stack ADT is given in Code Fragment 5.1. This interface defines a class template. Recall from Section 2.3 that such a definition implies that the base type of element being stored in the stack will be provided by the user. In Code Fragment 5.1, this element type is indicated by E. For example, E may be any fundamental type (such as **int**, **char**, **bool**, and **double**), any built-in or user-defined class (such as string), or a pointer to any of these.

```
template <typename E>
class Stack {                                   // an interface for a stack
public:
    int size() const;                           // number of items in stack
    bool empty() const;                         // is the stack empty?
    const E& top() const throw(StackEmpty);     // the top element
    void push(const E& e);                      // push x onto the stack
    void pop() throw(StackEmpty);               // remove the top element
};
```

Code Fragment 5.1: An informal Stack interface (not a complete C++ class).

Observe that the member functions size, empty, and top are all declared to be **const**, which informs the compiler that they do not alter the contents of the stack. The member function top returns a constant reference to the top of the stack, which means that its value may be read but not written.

Note that pop does not return the element that was popped. If the user wants to know this value, it is necessary to perform a top operation first, and save the value. The member function push takes a constant reference to an object of type E as its argument. Recall from Section 1.4 that this is the most efficient way of passing objects to a function.

An error condition occurs when calling either of the functions pop or top on an empty stack. This is signaled by throwing an exception of type StackEmpty, which is defined in Code Fragment 5.2.

```
// Exception thrown on performing top or pop of an empty stack.
class StackEmpty : public RuntimeException {
public:
    StackEmpty(const string& err) : RuntimeException(err) {}
};
```

Code Fragment 5.2: Exception thrown by functions pop and top when called on an empty stack. This class is derived from RuntimeException from Section 2.4.

5.1.4   A Simple Array-Based Stack Implementation

We can implement a stack by storing its elements in an array. Specifically, the stack in this implementation consists of an N-element array S plus an integer variable t that gives the index of the top element in array S. (See Figure 5.2.)
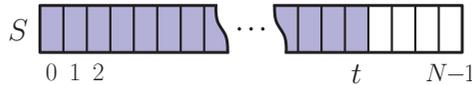
Figure 5.2: Realization of a stack by means of an array S. The top element in the stack is stored in the cell $S[t]$.

Recalling that arrays in C++ start at index 0, we initialize t to -1, and use this value for t to identify when the stack is empty. Likewise, we can use this variable to determine the number of elements in a stack $(t+1)$. We also introduce a new type of exception, called StackFull, to signal the error condition that arises if we try to insert a new element and the array S is full. Exception StackFull is specific to our implementation of a stack and is not defined in the stack ADT. Given this new exception, we can then implement the stack ADT functions as described in Code Fragment 5.3.

Algorithm size():
    return $t+1$
Algorithm empty():
    return $(t < 0)$
Algorithm top():
    if empty() **then**
       throw StackEmpty exception
    return $S[t]$
Algorithm push(e):
    if size() $= N$ **then**
       throw StackFull exception
    $t \leftarrow t+1$
    $S[t] \leftarrow e$
Algorithm pop():
    if empty() **then**
       throw StackEmpty exception
    $t \leftarrow t-1$

Code Fragment 5.3: Implementation of a stack by means of an array.

The correctness of the functions in the array-based implementation follows immediately from the definition of the functions themselves. Table 5.1 shows the

running times for member functions in a realization of a stack by an array. Each of the stack functions in the array realization executes a constant number of statements involving arithmetic operations, comparisons, and assignments. Thus, in this implementation of the Stack ADT, each function runs in constant time, that is, they each run in $O(1)$ time.

Operation	*Time*
size	$O(1)$
empty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Table 5.1: Performance of an array-based stack. The space usage is $O(N)$, where N is the array's size. Note that the space usage is independent from the number $n \leq N$ of elements that are actually in the stack.

A C++ Implementation of a Stack

In this section, we present a concrete C++ implementation of the above pseudo-code specification by means of a class, called ArrayStack. Our approach is to store the elements of a stack in an array. To keep the code simple, we have omitted the standard housekeeping utilities, such as a destructor, an assignment operator, and a copy constructor. We leave their implementations as an exercise.

We begin by providing the ArrayStack class definition in Code Fragment 5.4.

```
template <typename E>
class ArrayStack {
  enum { DEF_CAPACITY = 100 };          // default stack capacity
public:
  ArrayStack(int cap = DEF_CAPACITY);   // constructor from capacity
  int size() const;                     // number of items in the stack
  bool empty() const;                   // is the stack empty?
  const E& top() const throw(StackEmpty); // get the top element
  void push(const E& e) throw(StackFull); // push element onto stack
  void pop() throw(StackEmpty);         // pop the stack
  // ...housekeeping functions omitted
private:                                // member data
  E* S;                                 // array of stack elements
  int capacity;                         // stack capacity
  int t;                                // index of the top of the stack
};
```

Code Fragment 5.4: The class ArrayStack, which implements the Stack interface.

In addition to the member functions required by the interface, we also provide a constructor, that is given the desired capacity of the stack as its only argument. If no argument is given, the default value given by DEF_CAPACITY is used. This is an example of using default arguments in function calls. We use an enumeration to define this default capacity value. This is the simplest way of defining symbolic integer constants within a C++ class. Our class is templated with the element type, denoted by E. The stack's storage, denoted *S*, is a dynamically allocated array of type E, that is, a pointer to E.

Next, we present the implementations of the ArrayStack member functions in Code Fragment 5.5. The constructor allocates the array storage, whose size is set to the default capacity. The members *capacity* and *t* are also set to their initial values. In spite of the syntactical complexities of defining templated member functions in C++, the remaining member functions are straightforward implementations of their definitions in Code 5.3. Observe that functions top and pop first check that the stack is not empty, and otherwise, they throw an exception. Similarly, push first checks that the stack is not full, and otherwise, it throws an exception.

```cpp
template <typename E> ArrayStack<E>::ArrayStack(int cap)
  : S(new E[cap]), capacity(cap), t(−1) { } // constructor from capacity

template <typename E> int ArrayStack<E>::size() const
  { return (t + 1); }                        // number of items in the stack

template <typename E> bool ArrayStack<E>::empty() const
  { return (t < 0); }                        // is the stack empty?

template <typename E>                        // return top of stack
const E& ArrayStack<E>::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S[t];
}

template <typename E>                        // push element onto the stack
void ArrayStack<E>::push(const E& e) throw(StackFull) {
  if (size() == capacity) throw StackFull("Push to full stack");
  S[++t] = e;
}

template <typename E>                        // pop the stack
void ArrayStack<E>::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  −−t;
}
```

Code Fragment 5.5: Implementations of the member functions of class ArrayStack (excluding housekeeping functions).

Example Output

In Code Fragment 5.6 below, we present an example of the use of our ArrayStack class. To demonstrate the flexibility of our implementation, we show two stacks of different base types. The instance *A* is a stack of integers of the default capacity (100). The instance *B* is a stack of character strings of capacity 10.

```
ArrayStack<int> A;                    // A = [], size = 0
A.push(7);                            // A = [7*], size = 1
A.push(13);                           // A = [7, 13*], size = 2
cout << A.top() << endl; A.pop();     // A = [7*], outputs: 13
A.push(9);                            // A = [7, 9*], size = 2
cout << A.top() << endl;              // A = [7, 9*], outputs: 9
cout << A.top() << endl; A.pop();     // A = [7*], outputs: 9
ArrayStack<string> B(10);             // B = [], size = 0
B.push("Bob");                        // B = [Bob*], size = 1
B.push("Alice");                      // B = [Bob, Alice*], size = 2
cout << B.top() << endl; B.pop();     // B = [Bob*], outputs: Alice
B.push("Eve");                        // B = [Bob, Eve*], size = 2
```

Code Fragment 5.6: An example of the use of the ArrayStack class. The contents of the stack are shown in the comment following the operation. The top of the stack is indicated by an asterisk ("*").

Note that our implementation, while simple and efficient, could be enhanced in a number of ways. For example, it assumes a fixed upper bound N on the ultimate size of the stack. In Code Fragment 5.4, we chose the default capacity value $N = 100$ more or less arbitrarily (although the user can set the capacity in the constructor). An application may actually need much less space than the given initial size, and this would be wasteful of memory. Alternatively, an application may need more space than this, in which case our stack implementation might "crash" if too many elements are pushed onto the stack.

Fortunately, there are other implementations that do not impose an arbitrary size limitation. One such method is to use the STL stack class, which was introduced earlier in this chapter. The STL stack is also based on the STL vector class, and it offers the advantage that it is automatically expanded when the stack overflows its current storage limits. In practice, the STL stack would be the easiest and most practical way to implement an array-based stack. Later in this chapter, we see other methods that use space proportional to the actual size of the stack.

In instances where we have a good estimate on the number of items needing to go in the stack, the array-based implementation is hard to beat from the perspective of speed and simplicity. Stacks serve a vital role in a number of computing applications, so it is helpful to have a fast stack ADT implementation, such as the simple array-based implementation.

5.1.5 Implementing a Stack with a Generic Linked List

In this section, we show how to implement the stack ADT using a singly linked list. Our approach is to use the generic singly linked list, called SLinkedList, which was presented earlier in Section 3.2.4. The definition of our stack, called LinkedStack, is presented in Code Fragment 5.7.

To avoid the syntactic messiness inherent in C++ templated classes, we have chosen not to implement a fully generic templated class. Instead, we have opted to define a type for the stack's elements, called Elem. In this example, we define Elem to be of type string. We leave the task of producing a truly generic implementation as an exercise. (See Exercise R-5.7.)

```cpp
typedef string Elem;                        // stack element type
class LinkedStack {                          // stack as a linked list
public:
  LinkedStack();                             // constructor
  int size() const;                          // number of items in the stack
  bool empty() const;                        // is the stack empty?
  const Elem& top() const throw(StackEmpty); // the top element
  void push(const Elem& e);                  // push element onto stack
  void pop() throw(StackEmpty);              // pop the stack
private:                                     // member data
  SLinkedList<Elem> S;                       // linked list of elements
  int n;                                     // number of elements
};
```

Code Fragment 5.7: The class LinkedStack, a linked list implementation of a stack.

The principal data member of the class is the generic linked list of type Elem, called S. Since the SLinkedList class does not provide a member function size, we store the current size in a member variable, n.

In Code Fragment 5.8, we present the implementations of the constructor and the size and empty functions. Our constructor creates the initial stack and initializes n to zero. We do not provide an explicit destructor, relying instead on the SLinkedList destructor to deallocate the linked list S.

```cpp
LinkedStack::LinkedStack()
  : S(), n(0) { }                            // constructor

int LinkedStack::size() const
  { return n; }                              // number of items in the stack

bool LinkedStack::empty() const
  { return n == 0; }                         // is the stack empty?
```

Code Fragment 5.8: Constructor and size functions for the LinkedStack class.

The definitions of the stack operations, top, push, and pop, are presented in Code Fragment 5.9. Which side of the list, head or tail, should we chose for the top of the stack? Since SLinkedList can insert and delete elements in constant time only at the head, the head is clearly the better choice. Therefore, the member function top returns *S*.front(). The functions push and pop invoke the functions addFront and removeFront, respectively, and update the number of elements.

```
                                              // get the top element
const Elem& LinkedStack::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S.front();
}
void LinkedStack::push(const Elem& e) {    // push element onto stack
  ++n;
  S.addFront(e);
}
                                              // pop the stack
void LinkedStack::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  −−n;
  S.removeFront();
}
```

Code Fragment 5.9: Principal operations for the LinkedStack class.

5.1.6   Reversing a Vector Using a Stack

We can use a stack to reverse the elements in a vector, thereby producing a nonrecursive algorithm for the array-reversal problem introduced in Section 3.5.1. The basic idea is to push all the elements of the vector in order into a stack and then fill the vector back up again by popping the elements off of the stack. In Code Fragment 5.10, we give a C++ implementation of this algorithm.

```
template <typename E>
void reverse(vector<E>& V) {                // reverse a vector
  ArrayStack<E> S(V.size());
  for (int i = 0; i < V.size(); i++)        // push elements onto stack
    S.push(V[i]);
  for (int i = 0; i < V.size(); i++) {      // pop them in reverse order
    V[i] = S.top();  S.pop();
  }
}
```

Code Fragment 5.10: A generic function that uses a stack to reverse a vector.

For example, if the input vector to function reverse contained the five strings [Jack, Kate, Hurley, Jin, Michael], then on returning from the function, the vector would contain [Michael, Jin, Hurley, Kate, Jack].

5.1.7   Matching Parentheses and HTML Tags

In this section, we explore two related applications of stacks. The first is matching parentheses and grouping symbols in arithmetic expressions. Arithmetic expressions can contain various pairs of grouping symbols, such as

- Parentheses: "(" and ")"
- Braces: "{" and "}"
- Brackets: "[" and "]"
- Floor function symbols: "⌊" and "⌋"
- Ceiling function symbols: "⌈" and "⌉,"

and each opening symbol must match with its corresponding closing symbol. For example, a left bracket symbol ("[") must match with a corresponding right bracket ("]") as in the following expression:

- Correct: $()(()){([()])}$
- Correct: $((()(()){([()])}))$
- Incorrect: $)(()){([()])}$
- Incorrect: $({[])}$
- Incorrect: $($

We leave the precise definition of matching of grouping symbols to Exercise R-5.8.

An Algorithm for Parentheses Matching

An important problem in processing arithmetic expressions is to make sure their grouping symbols match up correctly. We can use a stack S to perform the matching of grouping symbols in an arithmetic expression with a single left-to-right scan. The algorithm tests that left and right symbols match up and also that the left and right symbols are both of the same type.

Suppose we are given a sequence $X = x_0 x_1 x_2 \ldots x_{n-1}$, where each x_i is a **token** that can be a grouping symbol, a variable name, an arithmetic operator, or a number. The basic idea behind checking that the grouping symbols in S match correctly, is to process the tokens in X in order. Each time we encounter an opening symbol, we push that symbol onto S, and each time we encounter a closing symbol, we pop the top symbol from the stack S (assuming S is not empty) and we check that these two symbols are of corresponding types. (For example, if the symbol "(" was pushed, the symbol ")" should be its match.) If the stack is empty after we have processed the whole sequence, then the symbols in X match.

Assuming that the push and pop operations are implemented to run in constant time, this algorithm runs in $O(n)$ total time. We give a pseudo-code description of this algorithm in Code Fragment 5.11.

Algorithm ParenMatch(X, n):
    Input: An array X of n tokens, each of which is either a grouping symbol, a
        variable, an arithmetic operator, or a number
    Output: **true** if and only if all the grouping symbols in X match
    Let S be an empty stack
    for $i \leftarrow 0$ to $n - 1$ **do**
      if $X[i]$ is an opening grouping symbol **then**
        S.push($X[i]$)
      else if $X[i]$ is a closing grouping symbol **then**
        if S.empty() **then**
          return false   {nothing to match with}
        if S.top() does not match the type of $X[i]$ **then**
          return false   {wrong type}
        S.pop()
    if S.empty() **then**
      return true   {every symbol matched}
    else
      return false   {some symbols were never matched}

Code Fragment 5.11: Algorithm for matching grouping symbols in an arithmetic expression.

Matching Tags in an HTML Document

Another application in which matching is important is in the validation of HTML documents. HTML is the standard format for hyperlinked documents on the Internet. In an HTML document, portions of text are delimited by ***HTML tags***. A simple opening HTML tag has the form "<name>" and the corresponding closing tag has the form "</name>." Commonly used HTML tags include:

- body: document body
- h1: section header
- center: center justify
- p: paragraph
- ol: numbered (ordered) list
- li: list item

We show a sample HTML document and a possible rendering in Figure 5.3. Our goal is to write a program to check that the tags properly match.

A very similar approach to that given in Code Fragment 5.11 can be used to match the tags in an HTML document. We push each opening tag on a stack, and when we encounter a closing tag, we pop the stack and verify that the two tags match.

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even
as a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine.   The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

(a)   (b)

Figure 5.3: HTML tags: (a) an HTML document; (b) its rendering.

In Code Fragments 5.12 through 5.14, we present a C++ program for matching tags in an HTML document read from the standard input stream. For simplicity, we assume that all tags are syntactically well formed.

First, the procedure getHtmlTags reads the input line by line, extracts all the tags as strings, and stores them in a vector, which it returns.

```cpp
vector<string> getHtmlTags() {                  // store tags in a vector
  vector<string> tags;                          // vector of html tags
  while (cin) {                                 // read until end of file
    string line;
    getline(cin, line);                         // input a full line of text
    int pos = 0;                                // current scan position
    int ts = line.find("<", pos);               // possible tag start
    while (ts != string::npos) {                // repeat until end of string
      int te = line.find(">", ts+1);            // scan for tag end
      tags.push_back(line.substr(ts, te-ts+1)); // append tag to the vector
      pos = te + 1;                             // advance our position
      ts = line.find("<", pos);
    }
  }
  return tags;                                  // return vector of tags
}
```

Code Fragment 5.12: Get a vector of HTML tags from the input, and store them in a vector of strings.

Given the example shown in Figure 5.3(a), this procedure would return the following vector:

```
<body>, <center>, <h1>, </h1>, </center>, ..., </body>
```

In Code Fragment 5.12, we employ a variable *pos*, which maintains the current position in the input line. We use the built-in string member function find to locate the first occurrence of "<" that follows the current position. (Recall the discussion of string operations from Section 1.5.5.) This tag start position is stored in the variable *ts*. We then find the next occurrence of ">," and store this tag end position in *te*. The tag itself consists of the substring of length $te - ts + 1$ starting at position *ts*. This is pushed onto the vector *tags*. We then update the current position to be $te + 1$ and repeat until we find no further occurrences of "<." This occurs when the find function returns the special value string::npos.

Next, the procedure isHtmlMatched, shown in Code Fragments 5.12, implements the process of matching the tags.

```cpp
                                            // check for matching tags
bool isHtmlMatched(const vector<string>& tags) {
  LinkedStack S;                            // stack for opening tags
  typedef vector<string>::const_iterator Iter;// iterator type
                                            // iterate through vector
  for (Iter p = tags.begin(); p != tags.end(); ++p) {
    if (p->at(1) != '/')                    // opening tag?
      S.push(*p);                           // push it on the stack
    else {                                  // else must be closing tag
      if (S.empty()) return false;          // nothing to match - failure
      string open = S.top().substr(1);      // opening tag excluding '<'
      string close = p->substr(2);          // closing tag excluding '</'
      if (open.compare(close) != 0) return false; // fail to match
      else S.pop();                         // pop matched element
    }
  }
  if (S.empty()) return true;               // everything matched - good
  else return false;                        // some unmatched - bad
}
```

Code Fragment 5.13: Check whether HTML tags stored in the vector tags are matched.

We create a stack, called *S*, in which we store the opening tags. We then iterate through the vector of tags. If the second character tag string is not "/," then this is an opening tag, and it is pushed onto the stack. Otherwise, it is a closing tag, and we check that it matches the tag on top of the stack. To compare the opening and closing tags, we use the string substr member function to strip the first character off the opening tag (thus removing the "<") and the first two characters off the closing tag (thus removing the "</"). We check that these two substrings are equal, using

the built-in string function compare. When the loop terminates, we know that every closing tag matches its corresponding opening tag. To finish the job, we need to check that there were no unmatched opening tags. We test this by checking that the stack is now empty.

Finally, the main program is presented in Code Fragment 5.14. It invokes the function getHtmlTags to read the tags, and then it passes these to isHtmlMatched to test them.

```
int main() {                                    // main HTML tester
  if (isHtmlMatched(getHtmlTags()))             // get tags and test them
    cout << "The input file is a matched HTML document." << endl;
  else
    cout << "The input file is not a matched HTML document." << endl;
}
```

Code Fragment 5.14: The main program to test whether the input file consists of matching HTML tags.

5.2   Queues

Another fundamental data structure is the *queue*, which is a close relative of the stack. A queue is a container of elements that are inserted and removed according to the *first-in first-out* (*FIFO*) principle. Elements can be inserted in a queue at any time, but only the element that has been in the queue the longest can be removed at any time. We usually say that elements enter the queue at the *rear* and are removed from the *front*. The metaphor for this terminology is a line of people waiting to get on an amusement park ride. People enter at the rear of the line and get on the ride from the front of the line.

5.2.1   The Queue Abstract Data Type

Formally, the queue abstract data type defines a container that keeps elements in a sequence, where element access and deletion are restricted to the first element in the sequence, which is called the *front* of the queue, and element insertion is restricted to the end of the sequence, which is called the *rear* of the queue. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in first-out (FIFO) principle.

The *queue* abstract data type (ADT) supports the following operations:

enqueue(*e*): Insert element *e* at the rear of the queue.

dequeue(): Remove element at the front of the queue; an error occurs if the queue is empty.

front(): Return, but do not remove, a reference to the front ele-
ment in the queue; an error occurs if the queue is empty.

The queue ADT also includes the following supporting member functions:

size(): Return the number of elements in the queue.

empty(): Return true if the queue is empty and false otherwise.

We illustrate the operations in the queue ADT in the following example.

Example 5.4: *The following table shows a series of queue operations and their effects on an initially empty queue, Q, of integers.*

Operation	*Output*	*front ← Q ← rear*
enqueue(5)	–	(5)
enqueue(3)	–	(5,3)
front()	5	(5,3)
size()	2	(5,3)
dequeue()	–	(3)
enqueue(7)	–	(3,7)
dequeue()	–	(7)
front()	7	(7)
dequeue()	–	()
dequeue()	*"error"*	()
empty()	*true*	()

5.2.2 The STL Queue

The Standard Template Library provides an implementation of a queue. As with the STL stack, the underlying implementation is based on the STL vector class (Sections 1.5.5 and 6.1.4). In order to declare an object of type queue, it is necessary to first include the definition file, which is called "queue." As with the STL vector, the class queue is part of the std namespace, so it is necessary either to use "std::queue" or to provide an appropriate "**using**" statement. The queue class is templated with the base type of the individual elements. For example, the code fragment below declares a queue of floats.

```
#include <queue>
using std::queue;                    // make queue accessible
queue<float> myQueue;                // a queue of floats
```

As with instances of STL vectors and stacks, an STL queue dynamically resizes itself as new elements are added.

The STL queue supports roughly the same operators as our interface, but the syntax and semantics are slightly different. Below, we list the principal member

functions. Let *q* be declared to be an STL queue, and let *e* denote a single object whose type is the same as the base type of the queue. (For example, *q* is a queue of floats, and *e* is a float.)

size(): Return the number of elements in the queue.

empty(): Return true if the queue is empty and false otherwise.

push(*e*): Enqueue *e* at the rear of the queue.

pop(): Dequeue the element at the front of the queue.

front(): Return a reference to the element at the queue's front.

back(): Return a reference to the element at the queue's rear.

Unlike our queue interface, the STL queue provides access to both the front and back of the queue. Similar to the STL stack, the result of applying any of the operations front, back, or pop to an empty STL queue is undefined. Unlike our interface, no exception is thrown, but it may very likely result in the program aborting. It is up to the programmer to be sure that no such illegal accesses are attempted.

5.2.3 A C++ Queue Interface

Our interface for the queue ADT is given in Code Fragment 5.15. As with the stack ADT, the class is templated. The queue's base element type E is provided by the user.

```
template <typename E>
class Queue {                                  // an interface for a queue
public:
  int size() const;                            // number of items in queue
  bool empty() const;                          // is the queue empty?
  const E& front() const throw(QueueEmpty);    // the front element
  void enqueue (const E& e);                   // enqueue element at rear
  void dequeue() throw(QueueEmpty);            // dequeue element at front
};
```

Code Fragment 5.15: An informal Queue interface (not a complete C++ class).

Note that the size and empty functions have the same meaning as their counterparts in the Stack ADT. These two member functions and front are known as *accessor* functions, for they return a value and do not change the contents of the data structure. Also note the use of the exception QueueEmpty to indicate the error state of an empty queue.

The member functions size, empty, and front are all declared to be **const**, which informs the compiler that they do not alter the contents of the queue. Note

that the member function front returns a constant reference to the top of the queue.

An error condition occurs when calling either of the functions front or dequeue on an empty queue. This is signaled by throwing an exception QueueEmpty, which is defined in Code Fragment 5.16.

```
class QueueEmpty : public RuntimeException {
public:
  QueueEmpty(const string& err) : RuntimeException(err) { }
};
```

Code Fragment 5.16: Exception thrown by functions front or dequeue when called on an empty queue. This class is derived from RuntimeException from Section 2.4.

5.2.4 A Simple Array-Based Implementation

We present a simple realization of a queue by means of an array, Q, with capacity N, for storing its elements. The main issue with this implementation is deciding how to keep track of the front and rear of the queue.

One possibility is to adapt the approach we used for the stack implementation. In particular, let $Q[0]$ be the front of the queue and have the queue grow from there. This is not an efficient solution, however, for it requires that we move all the elements forward one array cell each time we perform a dequeue operation. Such an implementation would therefore require $\Theta(n)$ time to perform the dequeue function, where n is the current number of elements in the queue. If we want to achieve constant time for each queue function, we need a different approach.

Using an Array in a Circular Way

To avoid moving objects once they are placed in Q, we define three variables, f, r, n, which have the following meanings:

- f is the index of the cell of Q storing the front of the queue. If the queue is nonempty, this is the index of the element to be removed by dequeue.

- r is an index of the cell of Q following the rear of the queue. If the queue is not full, this is the index where the element is inserted by enqueue.

- n is the current number of elements in the queue.

Initially, we set $n = 0$ and $f = r = 0$, indicating an empty queue. When we dequeue an element from the front of the queue, we decrement n and increment f to the next cell in Q. Likewise, when we enqueue an element, we increment r and increment n. This allows us to implement the enqueue and dequeue functions in constant time

Nonetheless, there is still a problem with this approach. Consider, for example, what happens if we repeatedly enqueue and dequeue a single element N different times. We would have $f = r = N$. If we were then to try to insert the element just one more time, we would get an array-out-of-bounds error, even though there is plenty of room in the queue in this case. To avoid this problem and be able to utilize all of the array Q, we let the f and r indices "wrap around" the end of Q. That is, we now view Q as a "circular array" that goes from $Q[0]$ to $Q[N-1]$ and then immediately back to $Q[0]$ again. (See Figure 5.4.)
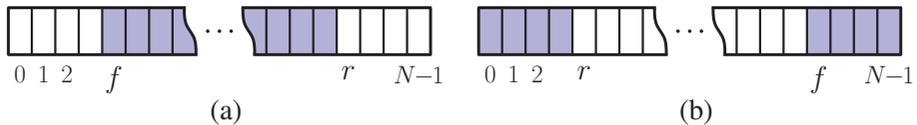
Figure 5.4: Using array Q in a circular fashion: (a) the "normal" configuration with $f \leq r$; (b) the "wrapped around" configuration with $r < f$. The cells storing queue elements are shaded.

Using the Modulo Operator to Implement a Circular Array

Implementing this circular view of Q is actually pretty easy. Each time we increment f or r, we simply need to compute this increment as "$(f+1) \bmod N$" or "$(r+1) \bmod N$," respectively, where the operator "mod" is the ***modulo*** operator. This operator is computed for a positive number by taking the remainder after an integral division. For example, 48 divided by 5 is 9 with remainder 3, so $48 \bmod 5 = 3$. Specifically, given integers x and y, such that $x \geq 0$ and $y > 0$, $x \bmod y$ is the unique integer $0 \leq r < y$ such that $x = qy + r$, for some integer q. Recall that C++ uses "%" to denote the modulo operator.

We present our implementation in Code Fragment 5.17. Note that we have introduced a new exception, called QueueFull, to signal that no more elements can be inserted in the queue. Our implementation of a queue by means of an array is similar to that of a stack, and is left as an exercise.

The array-based queue implementation is quite efficient. All of the operations of the queue ADT are performed in $O(1)$ time. The space usage is $O(N)$, where N is the size of the array, determined at the time the queue is created. Note that the space usage is independent from the number $n < N$ of elements that are actually in the queue.

As with the array-based stack implementation, the only real disadvantage of the array-based queue implementation is that we artificially set the capacity of the queue to be some number N. In a real application, we may actually need more or less queue capacity than this, but if we have a good estimate of the number of

Algorithm size():
    return n
Algorithm empty():
    return $(n = 0)$
Algorithm front():
    if empty() **then**
        throw QueueEmpty exception
    return $Q[f]$
Algorithm dequeue():
    if empty() **then**
        throw QueueEmpty exception
    $f \leftarrow (f + 1) \bmod N$
    $n = n - 1$
Algorithm enqueue(e):
    if size() $= N$ **then**
        throw QueueFull exception
    $Q[r] \leftarrow e$
    $r \leftarrow (r + 1) \bmod N$
    $n = n + 1$

Code Fragment 5.17: Implementation of a queue using a circular array.

elements that will be in the queue at the same time, then the array-based implementation is quite efficient. One such possible application of a queue is dynamic memory allocation in C++, which is discussed in Chapter 14.

5.2.5 Implementing a Queue with a Circularly Linked List

In this section, we present a C++ implementation of the queue ADT using a linked representation. Recall that we delete from the head of the queue and insert at the rear. Thus, unlike our linked stack of Code Fragment 5.7, we cannot use our singly linked list class, since it provides efficient access only to one side of the list. Instead, our approach is to use the circularly linked list, called CircleList, which was introduced earlier in Section 3.4.1.

Recall that CircleList maintains a pointer, called the ***cursor***, which points to one node of the list. Also recall that CircleList provides two member functions, back and front. The function back returns a reference to the element to which the cursor points, and the function front returns a reference to the element that immediately follows it in the circular list. In order to implement a queue, the element referenced by back will be the rear of the queue and the element referenced by front will be the front. (Why would it not work to reverse matters using the back of the circular list

as the front of the queue and the front of the circular list as the rear of the queue?)

Also recall that CircleList supports the following modifier functions. The function add inserts a new node just after the cursor, the function remove removes the node immediately following the cursor, and the function advance moves the cursor forward to the next node of the circular list.

In order to implement the queue operation enqueue, we first invoke the function add, which inserts a new element just after the cursor, that is, just after the rear of the queue. We then invoke advance, which advances the cursor to this new element, thus making the new node the rear of the queue. The process is illustrated in Figure 5.5.

(a)

(b)

(c)
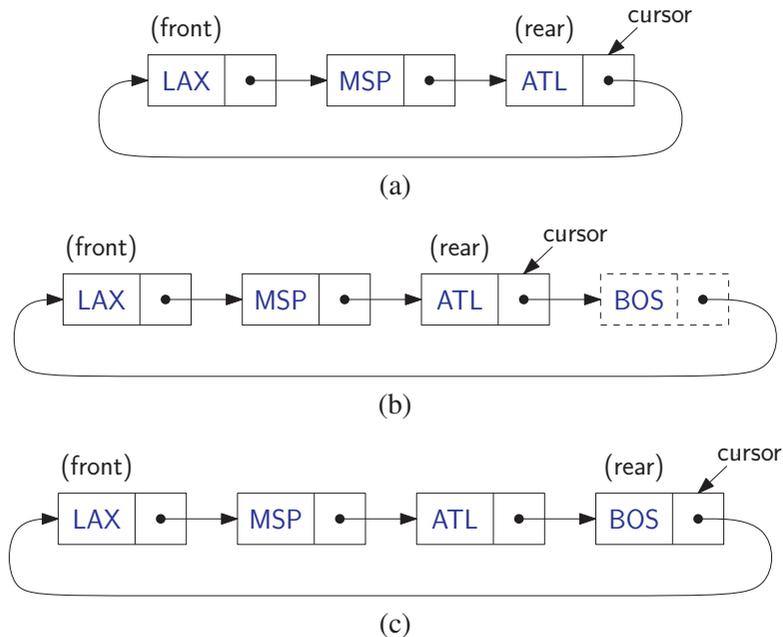
Figure 5.5: Enqueueing "BOS" into a queue represented as a circularly linked list: (a) before the operation; (b) after adding the new node; (c) after advancing the cursor.

In order to implement the queue operation dequeue, we invoke the function remove, thus removing the node just after the cursor, that is, the front of the queue. The process is illustrated in Figure 5.6.

The class structure for the resulting class, called LinkedQueue, is shown in Code Fragment 5.18. To avoid the syntactic messiness inherent in C++ templated classes, we have chosen not to implement a fully generic templated class. Instead, we have opted to define a type for the queue's elements, called Elem. In this example, we define Elem to be of type string. The queue is stored in the circular list
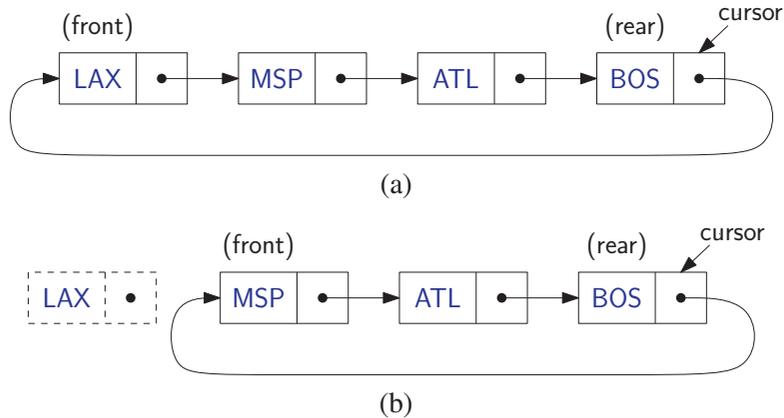
Figure 5.6: Dequeueing an element (in this case "LAX") from the front queue represented as a circularly linked list: (a) before the operation; (b) after removing the node immediately following the cursor.

data structure C. In order to support the size function (which CircleList does not provide), we also maintain the queue size in the member n.

```cpp
typedef string Elem;                          // queue element type
class LinkedQueue {                           // queue as doubly linked list
public:
  LinkedQueue();                              // constructor
  int size() const;                           // number of items in the queue
  bool empty() const;                         // is the queue empty?
  const Elem& front() const throw(QueueEmpty); // the front element
  void enqueue(const Elem& e);                // enqueue element at rear
  void dequeue() throw(QueueEmpty);           // dequeue element at front
private:                                      // member data
  CircleList C;                               // circular list of elements
  int n;                                      // number of elements
};
```

Code Fragment 5.18: The class LinkedQueue, an implementation of a queue based on a circularly linked list.

In Code Fragment 5.19, we present the implementations of the constructor and the basic accessor functions, size, empty, and front. Our constructor creates the initial queue and initializes n to zero. We do not provide an explicit destructor, relying instead on the destructor provided by CircleList. Observe that the function front throws an exception if an attempt is made to access the first element of an empty queue. Otherwise, it returns the element referenced by the front of the circular list, which, by our convention, is also the front element of the queue.

```
LinkedQueue::LinkedQueue()                        // constructor
  : C(), n(0) { }

int LinkedQueue::size() const                     // number of items in the queue
  { return n; }

bool LinkedQueue::empty() const                   // is the queue empty?
  { return n == 0; }
                                                  // get the front element
const Elem& LinkedQueue::front() const throw(QueueEmpty) {
  if (empty())
    throw QueueEmpty("front of empty queue");
  return C.front();                               // list front is queue front
}
```

Code Fragment 5.19: Constructor and accessor functions for the LinkedQueue class.

The definition of the queue operations, enqueue and dequeue are presented in Code Fragment 5.20. Recall that enqueuing involves invoking the add function to insert the new item immediately following the cursor and then advancing the cursor. Before dequeuing, we check whether the queue is empty, and, if so, we throw an exception. Otherwise, dequeuing involves removing the element that immediately follows the cursor. In either case, we update the number of elements in the queue.

```
                                                  // enqueue element at rear
void LinkedQueue::enqueue(const Elem& e) {
  C.add(e);                                       // insert after cursor
  C.advance();                                    // ...and advance
  n++;
}
                                                  // dequeue element at front
void LinkedQueue::dequeue() throw(QueueEmpty) {
  if (empty())
    throw QueueEmpty("dequeue of empty queue");
  C.remove();                                     // remove from list front
  n--;
}
```

Code Fragment 5.20: The enqueue and dequeue functions for LinkedQueue.

Observe that, all the operations of the queue ADT are implemented in $O(1)$ time. Therefore this implementation is quite efficient. Unlike the array-based implementation, by expanding and contracting dynamically, this implementation uses space proportional to the number of elements that are present in the queue at any time.

5.3 Double-Ended Queues

Consider now a queue-like data structure that supports insertion and deletion at both the front and the rear of the queue. Such an extension of a queue is called a ***double-ended queue***, or ***deque***, which is usually pronounced "deck" to avoid confusion with the dequeue function of the regular queue ADT, which is pronounced like the abbreviation "D.Q." An easy way to remember the "deck" pronunciation is to observe that a deque is like a deck of cards in the hands of a crooked card dealer—it is possible to deal off both the top and the bottom.

5.3.1 The Deque Abstract Data Type

The functions of the deque ADT are as follows, where *D* denotes the deque:

insertFront(*e*): Insert a new element *e* at the beginning of the deque.

insertBack(*e*): Insert a new element *e* at the end of the deque.

eraseFront(): Remove the first element of the deque; an error occurs if the deque is empty.

eraseBack(): Remove the last element of the deque; an error occurs if the deque is empty.

Additionally, the deque includes the following support functions:

front(): Return the first element of the deque; an error occurs if the deque is empty.

back(): Return the last element of the deque; an error occurs if the deque is empty.

size(): Return the number of elements of the deque.

empty(): Return true if the deque is empty and false otherwise.

Example 5.5: *The following example shows a series of operations and their effects on an initially empty deque, D, of integers.*

Operation	Output	D
insertFront(3)	–	(3)
insertFront(5)	–	(5,3)
front()	5	(5,3)
eraseFront()	–	(3)
insertBack(7)	–	(3,7)
back()	7	(3,7)
eraseFront()	–	(7)
eraseBack()	–	()

5.3.2 The STL Deque

As with the stack and queue, the Standard Template Library provides an implementation of a deque. The underlying implementation is based on the STL vector class (Sections 1.5.5 and 6.1.4). The pattern of usage is similar to that of the STL stack and STL queue. First, we need to include the definition file "deque." Since it is a member of the std namespace, we need to either preface each usage "std::deque" or provide an appropriate "**using**" statement. The deque class is templated with the base type of the individual elements. For example, the code fragment below declares a deque of strings.

```
#include <deque>
using std::deque;                    // make deque accessible
deque<string> myDeque;               // a deque of strings
```

As with STL stacks and queues, an STL deque dynamically resizes itself as new elements are added.

With minor differences, the STL deque class supports the same operators as our interface. Here is a list of the principal operations.

size(): Return the number of elements in the deque.

empty(): Return true if the deque is empty and false otherwise.

push_front(e): Insert e at the beginning the deque.

push_back(e): Insert e at the end of the deque.

pop_front(): Remove the first element of the deque.

pop_back(): Remove the last element of the deque.

front(): Return a reference to the deque's first element.

back(): Return a reference to the deque's last element.

Similar to STL stacks and queues, the result of applying any of the operations front, back, push_front, or push_back to an empty STL queue is undefined. Thus, no exception is thrown, but the program may abort.

5.3.3 Implementing a Deque with a Doubly Linked List

In this section, we show how to implement the deque ADT using a linked representation. As with the queue, a deque supports efficient access at both ends of the list, so our implementation is based on the use of a doubly linked list. Again, we use the doubly linked list class, called DLinkedList, which was presented earlier in Section 3.3.3. We place the front of the deque at the head of the linked list and the rear of the queue at the tail. An illustration is provided in Figure 5.7.
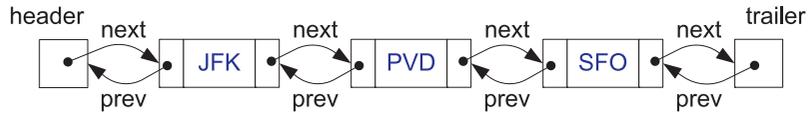
Figure 5.7: A doubly linked list with sentinels, *header* and *trailer*. The front of our deque is stored just after the header ("JFK"), and the back of our deque is stored just before the trailer ("SFO").

The definition of the resulting class, called LinkedDeque, is shown in Code Fragment 5.21. The deque is stored in the data member *D*. In order to support the size function, we also maintain the queue size in the member *n*. As in some of our earlier implementations, we avoid the syntactic messiness inherent in C++ templated classes, and instead use just a type definition to define the deque's base element type.

```
typedef string Elem;                       // deque element type
class LinkedDeque {                         // deque as doubly linked list
public:
  LinkedDeque();                            // constructor
  int size() const;                         // number of items in the deque
  bool empty() const;                       // is the deque empty?
  const Elem& front() const throw(DequeEmpty); // the first element
  const Elem& back() const throw(DequeEmpty);  // the last element
  void insertFront(const Elem& e);          // insert new first element
  void insertBack(const Elem& e);           // insert new last element
  void removeFront() throw(DequeEmpty);     // remove first element
  void removeBack() throw(DequeEmpty);      // remove last element
private:                                    // member data
  DLinkedList D;                            // linked list of elements
  int n;                                    // number of elements
};
```

Code Fragment 5.21: The class structure for class LinkedDeque.

We have not bothered to provide an explicit destructor, because the DLinkedList class provides its own destructor, which is automatically invoked when our Linked-Deque structure is destroyed.

Most of the member functions for the LinkedDeque class are straightforward generalizations of the corresponding functions of the LinkedQueue class, so we have omitted them. In Code Fragment 5.22, we present the implementations of the member functions for performing insertions and removals of elements from the deque. Observe that, in each case, we simply invoke the appropriate operation from the underlying DLinkedList object.

```
                                              // insert new first element
void LinkedDeque::insertFront(const Elem& e) {
  D.addFront(e);
  n++;
}
                                              // insert new last element
void LinkedDeque::insertBack(const Elem& e) {
  D.addBack(e);
  n++;
}
                                              // remove first element
void LinkedDeque::removeFront() throw(DequeEmpty) {
  if (empty())
    throw DequeEmpty("removeFront of empty deque");
  D.removeFront();
  n−−;
}
                                              // remove last element
void LinkedDeque::removeBack() throw(DequeEmpty) {
  if (empty())
    throw DequeEmpty("removeBack of empty deque");
  D.removeBack();
  n−−;
}
```

Code Fragment 5.22: The insertion and removal functions for LinkedDeque.

Table 5.2 shows the running times of functions in a realization of a deque by a doubly linked list. Note that every function of the deque ADT runs in $O(1)$ time.

Operation	Time
size	$O(1)$
empty	$O(1)$
front, back	$O(1)$
insertFront, insertBack	$O(1)$
eraseFront, eraseBack	$O(1)$

Table 5.2: Performance of a deque realized by a doubly linked list. The space usage is $O(n)$, where n is number of elements in the deque.

5.3.4 Adapters and the Adapter Design Pattern

An inspection of code fragments of Sections 5.1.5, 5.2.5, and 5.3.3, reveals a common pattern. In each case, we have taken an existing data structure and *adapted* it

to be used for a special purpose. For example, in Section 5.3.3, we showed how the DLinkedList class of Section 3.3.3 could be adapted to implement a deque. Except for the additional feature of keeping track of the number of elements, we have simply mapped each deque operation (such as insertFront) to the corresponding operation of DLinkedList (such as the addFront).

An *adapter* (also called a *wrapper*) is a data structure, for example, a class in C++, that translates one interface to another. You can think of an adapter as the software analogue to electric power plug adapters, which are often needed when you want to plug your electric appliances into electric wall sockets in different countries.

As an example of adaptation, observe that it is possible to implement the stack ADT by means of a deque data structure. That is, we can translate each stack operation to a functionally equivalent deque operation. Such a mapping is presented in Table 5.3.

Stack Method	*Deque Implementation*
size()	size()
empty()	empty()
top()	front()
push(*o*)	insertFront(*o*)
pop()	eraseFront()

Table 5.3: Implementing a stack with a deque.

Note that, because of the deque's symmetry, performing insertions and removals from the rear of the deque would have been equally efficient.

Likewise, we can develop the correspondences for the queue ADT, as shown in Table 5.4.

Queue Method	*Deque Implementation*
size()	size()
empty()	empty()
front()	front()
enqueue(*e*)	insertBack(*e*)
dequeue()	eraseFront()

Table 5.4: Implementing a queue with a deque.

As a more concrete example of the adapter design pattern, consider the code fragment shown in Code Fragment 5.23. In this code fragment, we present a class DequeStack, which implements the stack ADT. It's implementation is based on translating each stack operation to the corresponding operation on a LinkedDeque, which was introduced in Section 5.3.3.

```
typedef string Elem;                          // element type
class DequeStack {                            // stack as a deque
public:
  DequeStack();                               // constructor
  int size() const;                           // number of elements
  bool empty() const;                         // is the stack empty?
  const Elem& top() const throw(StackEmpty);  // the top element
  void push(const Elem& e);                   // push element onto stack
  void pop() throw(StackEmpty);               // pop the stack
private:
  LinkedDeque D;                              // deque of elements
};
```

Code Fragment 5.23: Implementation of the Stack interface by means of a deque.

The implementations of the various member functions are presented in Code Fragment 5.24. In each case, we translate some stack operation into the corresponding deque operation.

```
DequeStack::DequeStack()                      // constructor
  : D() { }
                                              // number of elements
int DequeStack::size() const
  { return D.size(); }
                                              // is the stack empty?
bool DequeStack::empty() const
  { return D.empty(); }
                                              // the top element
const Elem& DequeStack::top() const throw(StackEmpty) {
  if (empty())
    throw StackEmpty("top of empty stack");
  return D.front();
}
                                              // push element onto stack
void DequeStack::push(const Elem& e)
  { D.insertFront(e); }
                                              // pop the stack
void DequeStack::pop() throw(StackEmpty)
{
  if (empty())
    throw StackEmpty("pop of empty stack");
  D.removeFront();
}
```

Code Fragment 5.24: Implementation of the Stack interface by means of a deque.

5.4 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-5.1 Describe how to implement a capacity-limited stack, which uses the functions of a capacity-limited deque to perform the functions of the stack ADT in ways that do not throw exceptions when we attempt to perform a push on a full stack or a pop on an empty stack.

R-5.2 Describe how to implement a capacity-limited queue, which uses the functions of a capacity-limited deque to perform the functions of the queue ADT in ways that do not throw exceptions when we attempt to perform a enqueue on a full queue or a dequeue on an empty queue.

R-5.3 Suppose an initially empty stack *S* has performed a total of 25 push operations, 12 top operations, and 10 pop operations, 3 of which generated a StackEmpty exception that was caught and ignored. What is the current size of *S*?

R-5.4 If we implemented the stack *S* from the previous problem with an array, as described in this chapter, then what is the current value of the *top* member variable?

R-5.5 Describe the output of the following series of stack operations: push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop().

R-5.6 Give a recursive function for removing all the elements in a stack.

R-5.7 Modify the stack ADT implementation of Section 5.1.5 as a fully generic class (through the use of templates).

R-5.8 Give a precise and complete definition of the concept of matching for grouping symbols in an arithmetic expression.

R-5.9 Describe the output for the following sequence of queue operations:
enqueue(5), enqueue(3), dequeue(), enqueue(2), enqueue(8), dequeue(), dequeue(), enqueue(9), enqueue(1), dequeue(), enqueue(7), enqueue(6), dequeue(), dequeue(), enqueue(4), dequeue(), dequeue().

R-5.10 Describe the output for the following sequence of deque operations:
insertFront(3), insertBack(8), insertBack(9), insertFront(5), removeFront(), eraseBack(), first(), insertBack(7), removeFront(), last(), eraseBack().

R-5.11 Suppose you have a deque D containing the numbers $(1,2,3,4,5,6,7,8)$, in this order. Suppose further that you have an initially empty queue Q. Give a pseudo-code description of a function that uses only D and Q (and no other variables or objects) and results in D storing the elements $(1,2,3,5,4,6,7,8)$, in this order.

R-5.12 Repeat the previous problem using the deque D and an initially empty stack S.

Creativity

C-5.1 Explain how you can implement all the functions of the deque ADT using two stacks.

C-5.2 Suppose you have a stack S containing n elements and a queue Q that is initially empty. Describe how you can use Q to scan S to see if it contains a certain element x, with the additional constraint that your algorithm must return the elements back to S in their original order. You may not use an array or linked list—only S and Q and a constant number of reference variables.

C-5.3 Give a pseudo-code description for an array-based implementation of the deque ADT. What is the running time for each operation?

C-5.4 Suppose Alice has picked three distinct integers and placed them into a stack S in random order. Write a short, straight-line piece of pseudo-code (with no loops or recursion) that uses only one comparison and only one variable x, yet guarantees with probability $2/3$ that at the end of this code the variable x will store the largest of Alice's three integers. Argue why your method is correct.

C-5.5 Describe how to implement the stack ADT using two queues. What is the running time of the push and pop functions in this case?

C-5.6 Suppose we have an $n \times n$ two-dimensional array A that we want to use to store integers, but we don't want to spend the $O(n^2)$ work to initialize it to all 0's, because we already know that we are only going to use up to n of these cells in our algorithm, which itself runs in $O(n)$ time (not counting the time to initialize A). Show how to use an array-based stack S storing (i,j,k) integer triples to allow us to use the array A without initializing it and still implement our algorithm in $O(n)$ time, even though the initial values in the cells of A might be total garbage.

C-5.7 Describe a nonrecursive algorithm for enumerating all permutations of the numbers $\{1,2,\ldots,n\}$.

C-5.8 **Postfix notation** is an unambiguous way of writing an arithmetic expression without parentheses. It is defined so that if "$(exp_1) \circ (exp_2)$" is a

normal fully parenthesized expression whose operation is "∘", then the postfix version of this is "$pexp_1$ $pexp_2$∘", where $pexp_1$ is the postfix version of exp_1 and $pexp_2$ is the postfix version of exp_2. The postfix version of a single number or variable is just that number or variable. So, for example, the postfix version of "$((5+2)*(8-3))/4$" is "5 2 + 8 3 − ∗ 4 /". Describe a nonrecursive way of evaluating an expression in postfix notation.

C-5.9 Suppose you have two nonempty stacks S and T and a deque D. Describe how to use D so that S contains all the elements of T below all of its original elements, with both sets of elements still in their original order.

C-5.10 Alice has three array-based stacks, A, B, and C, such that A has capacity 100, B has capacity 5, and C has capacity 3. Initially, A is full, and B and C are empty. Unfortunately, the person who programmed the class for these stacks made the push and pop functions private. The only function Alice can use is a static function, transfer(S,T), which transfers (by iteratively applying the private pop and push functions) elements from stack S to stack T until either S becomes empty or T becomes full. So, for example, starting from our initial configuration and performing transfer(A,C) results in A now holding 97 elements and C holding 3. Describe a sequence of transfer operations that starts from the initial configuration and results in B holding 4 elements at the end.

C-5.11 Show how to use a stack S and a queue Q to generate all possible subsets of an n-element set T nonrecursively.

Projects

P-5.1 Give an implementation of the deque ADT using an array, so that each of the update functions run in $O(1)$ time.

P-5.2 Design an ADT for a two-color, double-stack ADT that consists of two stacks—one "red" and one "blue"—and has as its operations color-coded versions of the regular stack ADT operations. For example, this ADT should allow for both a red push operation and a blue push operation. Give an efficient implementation of this ADT using a single array whose capacity is set at some value N that is assumed to always be larger than the sizes of the red and blue stacks combined.

P-5.3 Implement the stack ADT in a fully generic manner (through the use of templates) by means of a singly linked list. (Give your implementation "from scratch," without the use of any classes from the Standard Template Library or data structures presented earlier in this book.)

P-5.4 Implement the stack ADT in a fully generic manner using the STL vector class.

P-5.5   Implement the queue ADT in a fully generic manner using a dynamically allocated C++ array.

P-5.6   Implement the queue ADT with a singly linked list.

P-5.7   Implement the deque ADT with an array used in a circular fashion.

P-5.8   Implement the deque ADT with a doubly linked list.

P-5.9   Implement a capacity-limited version of the deque ADT based on an array used in a circular fashion, similar to queue implementation of Section 5.2.4.

P-5.10 Implement the Stack and Queue interfaces with a unique class that is derived from class LinkedDeque (Code Fragment 5.21).

P-5.11 When a share of common stock of some company is sold, the *capital gain* (or, sometimes, loss) is the difference between the share's selling price and the price originally paid to buy it. This rule is easy to understand for a single share, but if we sell multiple shares of stock bought over a long period of time, then we must identify the shares actually being sold. A standard accounting principle for identifying which shares of a stock were sold in such a case is to use a FIFO protocol—the shares sold are the ones that have been held the longest (indeed, this is the default method built into several personal finance software packages). For example, suppose we buy 100 shares at $20 each on day 1, 20 shares at $24 on day 2, 200 shares at $36 on day 3, and then sell 150 shares on day 4 at $30 each. Then applying the FIFO protocol means that of the 150 shares sold, 100 were bought on day 1, 20 were bought on day 2, and 30 were bought on day 3. The capital gain in this case would therefore be $100 \cdot 10 + 20 \cdot 6 + 30 \cdot (-6)$, or $940. Write a program that takes as input a sequence of transactions of the form "buy x share(s) at \$$y$ each" or "sell x share(s) at \$$y$ each," assuming that the transactions occur on consecutive days and the values x and y are integers. Given this input sequence, the output should be the total capital gain (or loss) for the entire sequence, using the FIFO protocol to identify shares.

P-5.12 Implement a program that can input an expression in postfix notation (see Exercise C-5.8) and output its value.

Chapter Notes

We were introduced to the approach of defining data structures first in terms of their ADTs and then in terms of concrete implementations by the classic books by Aho, Hopcroft, and Ullman [4, 5], which incidentally is where we first saw a problem similar to Exercise C-5.6. Exercise C-5.10 is similar to interview questions said to be from a well-known software company. For further study of abstract data types, see Liskov and Guttag [68], Cardelli and Wegner [19], or Demurjian [27].

Chapter

6

List and Iterator ADTs

Contents

6.1   Vectors

Suppose we have a collection S of n elements stored in a certain linear order, so that we can refer to the elements in S as first, second, third, and so on. Such a collection is generically referred to as a **list** or **sequence**. We can uniquely refer to each element e in S using an integer in the range $[0, n-1]$ that is equal to the number of elements of S that precede e in S. The **index** of an element e in S is the number of elements that are before e in S. Hence, the first element in S has index 0 and the last element has index $n-1$. Also, if an element of S has index i, its previous element (if it exists) has index $i-1$, and its next element (if it exists) has index $i+1$. This concept of index is related to that of the **rank** of an element in a list, which is usually defined to be one more than its index; so the first element is at rank 1, the second is at rank 2, and so on.

A sequence that supports access to its elements by their indices is called a **vector**. Since our index definition is more consistent with the way arrays are indexed in C++ and other common programming languages, we refer to the place where an element is stored in a vector as its "index," rather than its "rank."

This concept of index is a simple yet powerful notion, since it can be used to specify where to insert a new element into a list or where to remove an old element.

6.1.1   The Vector Abstract Data Type

A **vector**, also called an **array list**, is an ADT that supports the following fundamental functions (in addition to the standard size() and empty() functions). In all cases, the index parameter i is assumed to be in the range $0 \leq i \leq \text{size}() - 1$.

at(i): Return the element of V with index i; an error condition occurs if i is out of range.

set(i, e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i, e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

We do **not** insist that an array be used to implement a vector, so that the element at index 0 is stored at index 0 in the array, although that is one (very natural) possibility. The index definition offers us a way to refer to the "place" where an element is stored in a sequence without having to worry about the exact implementation of that sequence. The index of an element may change when the sequence is updated, however, as we illustrate in the following example.

Example 6.1: *We show below some operations on an initially empty vector V.*

Operation	*Output*	*V*
insert$(0,7)$	–	(7)
insert$(0,4)$	–	$(4,7)$
at(1)	7	$(4,7)$
insert$(2,2)$	–	$(4,7,2)$
at(3)	"error"	$(4,7,2)$
erase(1)	–	$(4,2)$
insert$(1,5)$	–	$(4,5,2)$
insert$(1,3)$	–	$(4,3,5,2)$
insert$(4,9)$	–	$(4,3,5,2,9)$
at(2)	5	$(4,3,5,2,9)$
set$(3,8)$	–	$(4,3,5,8,9)$

6.1.2 A Simple Array-Based Implementation

An obvious choice for implementing the vector ADT is to use a fixed size array A, where $A[i]$ stores the element at index i. We choose the size N of array A to be sufficiently large, and we maintain the number $n < N$ of elements in the vector in a member variable.

The details of the implementation of the functions of the vector ADT are reasonably simple. To implement the at(i) operation, for example, we just return $A[i]$. The implementations of the functions insert(i,e) and erase(i) are given in Code Fragment 6.1.

Algorithm insert(i,e):
　　for $j = n-1, n-2, \ldots, i$ **do**
　　　$A[j+1] \leftarrow A[j]$ 　　　　{make room for the new element}
　　$A[i] \leftarrow e$
　　$n \leftarrow n+1$
Algorithm erase(i):
　　for $j = i+1, i+2, \ldots, n-1$ **do**
　　　$A[j-1] \leftarrow A[j]$ 　　　　{fill in for the removed element}
　　$n \leftarrow n-1$

Code Fragment 6.1: Methods insert(i,e) and erase(i) in the array implementation of the vector ADT. The member variable n stores the number of elements.

An important (and time-consuming) part of this implementation involves the shifting of elements up or down to keep the occupied cells in the array contiguous. These shifting operations are required to maintain our rule of always storing an

element whose list index i at index i in the array A. (See Figure 6.1.)
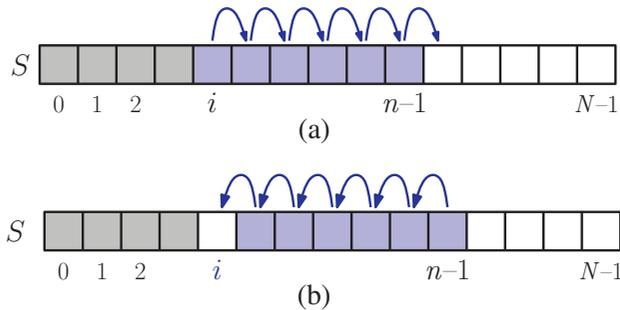
Figure 6.1: Array-based implementation of a vector V that is storing n elements: (a) shifting up for an insertion at index i; (b) shifting down for a removal at index i.

The Performance of a Simple Array-Based Implementation

Table 6.1 shows the worst-case running times of the functions of a vector with n elements realized by means of an array. Methods empty, size, at, and set clearly run in $O(1)$ time, but the insertion and removal functions can take much longer than this. In particular, insert(i,e) runs in time $O(n)$. Indeed, the worst case for this operation occurs when $i = 0$, since all the existing n elements have to be shifted forward. A similar argument applies to function erase(i), which runs in $O(n)$ time, because we have to shift backward $n-1$ elements in the worst case ($i = 0$). In fact, assuming that each possible index is equally likely to be passed as an argument to these operations, their average running time is $O(n)$ because we have to shift $n/2$ elements on average.

Operation	Time
size()	$O(1)$
empty()	$O(1)$
at(i)	$O(1)$
set(i,e)	$O(1)$
insert(i,e)	$O(n)$
erase(i)	$O(n)$

Table 6.1: Performance of a vector with n elements realized by an array. The space usage is $O(N)$, where N is the size of the array.

Looking more closely at insert(i,e) and erase(i), we note that they each run in time $O(n-i+1)$, for only those elements at index i and higher have to be shifted

up or down. Thus, inserting or removing an item at the end of a vector, using the functions insert(n, e) and erase$(n-1)$, take $O(1)$ time each respectively. Moreover, this observation has an interesting consequence for the adaptation of the vector ADT to the deque ADT given in Section 5.3.1. If the vector ADT in this case is implemented by means of an array as described above, then functions insertBack and eraseBack of the deque each run in $O(1)$ time. However, functions insertFront and eraseFront of the deque each run in $O(n)$ time.

Actually, with a little effort, we can produce an array-based implementation of the vector ADT that achieves $O(1)$ time for insertions and removals at index 0, as well as insertions and removals at the end of the vector. Achieving this requires that we give up on our rule that an element at index i is stored in the array at index i, however, as we would have to use a circular array approach like the one we used in Section 5.2 to implement a queue. We leave the details of this implementation for an exercise (R-6.17).

6.1.3 An Extendable Array Implementation

A major weakness of the simple array implementation for the vector ADT given in Section 6.1.2 is that it requires advance specification of a fixed capacity, N, for the total number of elements that may be stored in the vector. If the actual number of elements, n, of the vector is much smaller than N, then this implementation will waste space. Worse, if n increases past N, then this implementation will crash. Fortunately, there is a simple way to fix this major drawback.

Let us provide a means to grow the array A that stores the elements of a vector V. Of course, in C++ (and most other programming languages) we cannot actually grow the array A; its capacity is fixed at some number N, as we have already observed. Instead, when an **overflow** occurs, that is, when $n = N$ and function insert is called, we perform the following steps:

1. Allocate a new array B of capacity N
2. Copy $A[i]$ to $B[i]$, for $i = 0, \ldots, N-1$
3. Deallocate A and reassign A to point to the new array B

This array replacement strategy is known as an **extendable array**, for it can be viewed as extending the end of the underlying array to make room for more elements. (See Figure 6.2.) Intuitively, this strategy is much like that of the hermit crab, which moves into a larger shell when it outgrows its previous one.

We give an implementation of the vector ADT using an extendable array in Code Fragment 6.2. To avoid the complexities of templated classes, we have adopted our earlier practice of using a type definition to specify the base element type, which is an **int** in this case. The class is called ArrayVector. We leave the details of producing a fully generic templated class as an exercise (R-6.7).
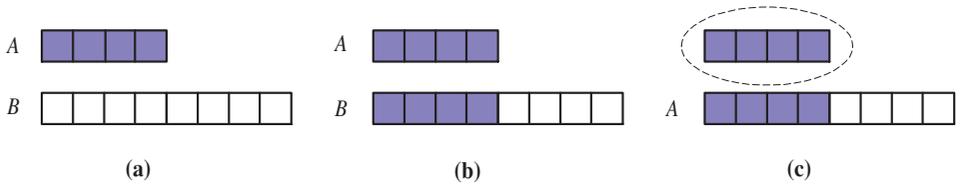
Figure 6.2: The three steps for "growing" an extendable array: (a) create new array *B*; (b) copy elements from *A* to *B*; (c) reassign *A* to refer to the new array and delete the old array.

Our class definition differs slightly from the operations given in our ADT. For example, we provide two means for accessing individual elements of the vector. The first involves overriding the C++ array index operator ("[ ]"), and the second is the at function. The two functions behave the same, except that the at function performs a range test before each access. (Note the similarity with the STL vector class given in Section 6.1.4.) If the index *i* is not in bounds, this function throws an exception. Because both of these access operations return a reference, there is no need to explicitly define a set function. Instead, we can simply use the assignment operator. For example, the ADT function *v*.set(i, 5) could be implemented either as $v[i] = 5$ or, more safely, as v.at(i) = 5.

```
typedef int Elem;                     // base element type
class ArrayVector {
public:
  ArrayVector();                      // constructor
  int size() const;                   // number of elements
  bool empty() const;                 // is vector empty?
  Elem& operator[](int i);            // element at index
  Elem& at(int i) throw(IndexOutOfBounds); // element at index
  void erase(int i);                  // remove element at index
  void insert(int i, const Elem& e);  // insert element at index
  void reserve(int N);                // reserve at least N spots
  // ... (housekeeping functions omitted)
private:
  int capacity;                       // current array size
  int n;                              // number of elements in vector
  Elem* A;                            // array storing the elements
};
```

Code Fragment 6.2: A vector implementation using an extendable array.

The member data for class ArrayVector consists of the array storage *A*, the current number *n* of elements in the vector, and the current storage capacity. The class ArrayVector also provides the ADT functions insert and remove. We discuss their implementations below. We have added a new function, called reserve, that

is not part of the ADT. This function allows the user to explicitly request that the array be expanded to a capacity of a size at least *n*. If the capacity is already larger than this, then the function does nothing.

Even though we have not bothered to show them, the class also provides some of the standard housekeeping functions. These consist of a copy constructor, an assignment operator, and a destructor. Because this class allocates memory, their inclusion is essential for a complete and robust class implementation. We leave them as an exercise (R-6.6). We should also add versions of the indexing operators that return constant references.

In Code Fragment 6.3, we present the class constructor and a number of simple member functions. When the vector is constructed, we do not allocate any storage and simply set *A* to NULL. Note that the first attempt to add an element results in array storage being allocated.

```
ArrayVector::ArrayVector()                      // constructor
  : capacity(0), n(0), A(NULL) { }

int ArrayVector::size() const                   // number of elements
  { return n; }

bool ArrayVector::empty() const                 // is vector empty?
  { return size() == 0; }

Elem& ArrayVector::operator[](int i)            // element at index
  { return A[i]; }
                                                // element at index (safe)
Elem& ArrayVector::at(int i) throw(IndexOutOfBounds) {
  if (i < 0 || i >= n)
    throw IndexOutOfBounds("illegal index in function at()");
  return A[i];
}
```

Code Fragment 6.3: The simple member functions for class ArrayVector.

In Code Fragment 6.4, we present the member function erase. As mentioned above, it removes an element at index *i* by shifting all subsequent elements from index $i+1$ to the last element of the array down by one position.

```
void ArrayVector::erase(int i) {                // remove element at index
  for (int j = i+1; j < n; j++)                 // shift elements down
    A[j − 1] = A[j];
  n−−;                                          // one fewer element
}
```

Code Fragment 6.4: The member function remove for class ArrayVector.

Finally, in Code Fragment 6.5, we present the reserve and insert functions. The reserve function first checks whether the capacity already exceeds *n*, in which case nothing needs to be done. Otherwise, it allocates a new array *B* of the desired sizes, copies the contents of *A* to *B*, deletes *A*, and makes *B* the current array. The insert function first checks whether there is sufficient capacity for one more element. If not, it sets the capacity to the maximum of 1 and twice the current capacity. Then starting at the insertion point, it shifts elements up by one position, and stores the new element in the desired position.

```
void ArrayVector::reserve(int N) {         // reserve at least N spots
  if (capacity >= N) return;               // already big enough
  Elem* B = new Elem[N];                   // allocate bigger array
  for (int j = 0; j < n; j++)              // copy contents to new array
    B[j] = A[j];
  if (A != NULL) delete [] A;              // discard old array
  A = B;                                   // make B the new array
  capacity = N;                            // set new capacity
}
void ArrayVector::insert(int i, const Elem& e) {
  if (n >= capacity)                       // overflow?
    reserve(max(1, 2 * capacity));         // double array size
  for (int j = n − 1; j >= i; j−−)         // shift elements up
    A[j+1] = A[j];
  A[i] = e;                                // put in empty slot
  n++;                                     // one more element
}
```

Code Fragment 6.5: The member functions reserve and insert for class ArrayVector.

In terms of efficiency, this array replacement strategy might, at first, seem rather slow. After all, performing just one array replacement required by an element insertion takes $O(n)$ time, which is not very good. Notice, however, that, after we perform an array replacement, our new array allows us to add *n* new elements to the vector before the array must be replaced again.

This simple observation allows us to show that the running time of a series of operations performed on an initially empty vector is proportional to the total number of elements added. As a shorthand notation, let us refer to the insertion of an element meant to be the last element in a vector as a "push" operation. Using a design pattern called ***amortization***, we show below that performing a sequence of push operations on a vector implemented with an extendable array is quite efficient.

Proposition 6.2: *Let V be a vector implemented by means of an extendable array A, as described above. The total time to perform a series of n push operations in V, starting from V being empty and A having size $N = 1$, is $O(n)$.*

Justification: To perform this analysis, we view the computer as a coin-operated

appliance, which requires the payment of one ***cyber-dollar*** for a constant amount of computing time. When an operation is executed, we should have enough cyber-dollars available in our current "bank account" to pay for that operation's running time. Thus, the total amount of cyber-dollars spent for any computation is proportional to the total time spent on that computation. The beauty of using this analysis method is that we can overcharge some operations in order to save up cyber-dollars to pay for others.

Let us assume that one cyber-dollar is enough to pay for the execution of each push operation in V, excluding the time spent for growing the array. Also, let us assume that growing the array from size k to size $2k$ requires k cyber-dollars for the time spent copying the elements. We shall charge each push operation three cyber-dollars. Thus, we overcharge each push operation that does not cause an overflow by two cyber-dollars. Think of the two cyber-dollars profited in an insertion that does not grow the array as being "stored" at the element inserted.

An overflow occurs when the vector V has 2^i elements, for some $i \geq 0$, and the size of the array used by V is 2^i. Thus, doubling the size of the array requires 2^i cyber-dollars. Fortunately, these cyber-dollars can be found at the elements stored in cells 2^{i-1} through $2^i - 1$. (See Figure 6.3.) Note that the previous overflow occurred when the number of elements became larger than 2^{i-1} for the first time, and thus the cyber-dollars stored in cells 2^{i-1} through $2^i - 1$ were not previously spent. Therefore, we have a valid amortization scheme in which each operation is charged three cyber-dollars and all the computing time is paid for. That is, we can pay for the execution of n push operations using $3n$ cyber-dollars. ■
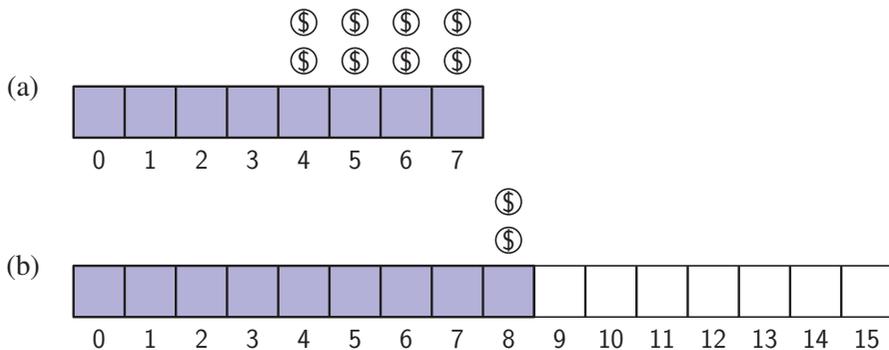
Figure 6.3: A series of push operations on a vector: (a) an 8-cell array is full, with two cyber-dollars "stored" at cells 4 through 7; (b) a push operation causes an overflow and a doubling of capacity. Copying the eight old elements to the new array is paid for by the cyber-dollars already stored in the table; inserting the new element is paid for by one of the cyber-dollars charged to the push operation; and two cyber-dollars profited are stored at cell 8.

6.1.4   STL Vectors

In Section 1.5.5, we introduced the vector class of the C++ Standard Template Library (STL). We mentioned that STL vectors behave much like standard arrays in C++, but they are superior to standard arrays in many respects. In this section we explore this class in greater detail.

The Standard Template Library provides C++ programmers a number of useful built-in classes and algorithms. The classes provided by the STL are organized in various groups. Among the most important of these groups is the set of classes called containers. A ***container*** is a data structure that stores a collection of objects. Many of the data structures that we study later in this book, such as stacks, queues, and lists, are examples of STL containers. The class vector is perhaps the most basic example of an STL container class. We discuss containers further in Section 6.2.1.

The definition of class vector is given in the system include file named "vector." The vector class is part of the std namespace, so it is necessary either to use "std::vector" or to provide an appropriate **using** statement. The vector class is templated with the class of the individual elements. For example, the code fragment below declares a vector containing 100 integers.

```cpp
#include <vector>              // provides definition of vector
using std::vector;            // make vector accessible

vector<int> myVector(100);    // a vector with 100 integers
```

We refer to the type of individual elements as the vector's ***base type***. Each element is initialized to the base type's default value, which for integers is zero.

STL vector objects behave in many respects like standard C++ arrays, but they provide many additional features.

- As with arrays, individual elements of a vector object can be indexed using the usual index operator ("[ ]"). Elements can also be accessed by a member function called at. The advantage of this member function over the index operator is that it performs range checking and generates an error exception if the index is out of bounds.
- Unlike C++ arrays, STL vectors can be dynamically resized, and new elements may be efficiently appended or removed from the end of an array.
- When an STL vector of class objects is destroyed, it automatically invokes the destructor for each of its elements. (With C++ arrays, it is the obligation of the programmer to do this explicitly.)
- STL vectors provide a number of useful functions that operate on entire vectors, not just on individual elements. This includes, for example, the ability to copy all or part of one vector to another, the ability to compare the contents of two arrays, and the ability to insert and erase multiple elements.

Here are the principal member functions of the vector class. Let V be declared to be an STL vector of some base type, and let e denote a single object of this same base type. (For example, V is a vector of integers, and e is an integer.)

vector(n): Construct a vector with space for n elements; if no argument is given, create an empty vector.

size(): Return the number of elements in V.

empty(): Return true if V is empty and false otherwise.

resize(n): Resize V, so that it has space for n elements.

reserve(n): Request that the allocated storage space be large enough to hold n elements.

operator[i]: Return a reference to the ith element of V.

at(i): Same as $V[i]$, but throw an out_of_range exception if i is out of bounds, that is, if $i < 0$ or $i \geq V$.size().

front(): Return a reference to the first element of V.

back(): Return a reference to the last element of V.

push_back(e): Append a copy of the element e to the end of V, thus increasing its size by one.

pop_back(): Remove the last element of V, thus reducing its size by one.

When the base type of an STL vector is class, all copying of elements (for example, in push_back) is performed by invoking the class's copy constructor. Also, when elements are destroyed (for example, by invoking the destroyer or the pop_back member function) the class's destructor is invoked on each deleted element. STL vectors are expandable—when the current array space is exhausted, its storage size is increased.

Although we have not discussed it here, the STL vector also supports functions for inserting elements at arbitrary positions within the vector, and for removing arbitrary elements of the vector. These are discussed in Section 6.1.4.

There are both similarities and differences between our ArrayVector class of Section 6.1.3 and the STL vector class. One difference is that the STL constructor allows for an arbitrary number of initial elements, whereas our ArrayVect constructor always starts with an empty vector. The STL vector functions V.front() and V.back() are equivalent to our functions $V[0]$ and $V[n-1]$, respectively, where n is equal to V.size(). The STL vector functions V.push_back(e) and V.pop_back() are equivalent to our ArrayVect functions V.insert(n, e) and V.remove($n-1$), respectively.

6.2   Lists

Using an index is not the only means of referring to the place where an element appears in a list. If we have a list L implemented with a (singly or doubly) linked list, then it could possibly be more natural and efficient to use a ***node*** instead of an index as a means of identifying where to access and update a list. In this section, define the list ADT, which abstracts the concrete linked list data structure (presented in Sections 3.2 and 3.3) using a related position ADT that abstracts the notion of "place" in a list.

6.2.1   Node-Based Operations and Iterators

Let L be a (singly or doubly) linked list. We would like to define functions for L that take nodes of the list as parameters and provide nodes as return types. Such functions could provide significant speedups over index-based functions, because finding the index of an element in a linked list requires searching through the list incrementally from its beginning or end, counting elements as we go.

For instance, we might want to define a hypothetical function remove(v) that removes the element of L stored at node v of the list. Using a node as a parameter allows us to remove an element in $O(1)$ time by simply going directly to the place where that node is stored and then "linking out" this node through an update of the *next* and *prev* links of its neighbors. Similarly, in $O(1)$ time, we could insert a new element e into L with an operation such as insert(v, e), which specifies the node v before which the node of the new element should be inserted. In this case, we simply "link in" the new node.

Defining functions of a list ADT by adding such node-based operations raises the issue of how much information we should be exposing about the implementation of our list. Certainly, it is desirable for us to be able to use either a singly or doubly linked list without revealing this detail to a user. Likewise, we do not wish to allow a user to modify the internal structure of a list without our knowledge. Such modifications would be possible, however, if we provided a pointer to a node in our list in a form that allows the user to access internal data in that node (such as the *next* or *prev* field).

To abstract and unify the different ways of storing elements in the various implementations of a list, we introduce a data type that abstracts the notion of the relative position or place of an element within a list. Such an object might naturally be called a ***position***. Because we want this object not only to access individual elements of a list, but also to move around in order to enumerate all the elements of a list, we adopt the convention used in the C++ Standard Template Library, and call it an ***iterator***.

Containers and Positions

In order to safely expand the set of operations for lists, we abstract a notion of "position" that allows us to enjoy the efficiency of doubly or singly linked list implementations without violating object-oriented design principles. In this framework, we think of a list as an instance of a more general class of objects, called a container. A ***container*** is a data structure that stores any collection of elements. We assume that the elements of a container can be arranged in a linear order. A ***position*** is defined to be an abstract data type that is associated with a particular container and which supports the following function.

> element(): Return a reference to the element stored at this position.

C++'s ability to overload operators provides us with an elegant alternative manner in which to express the element operation. In particular, we overload the dereferencing operator ("*"), so that, given a position variable p, the associated element can be accessed by $*p$, rather than p.element(). This can be used both for accessing and modifying the element's value.

A position is always defined in a ***relative*** manner, that is, in terms of its neighbors. Unless it is the first or last of the container, a position q is always "after" some position p and "before" some position r (see Figure 6.4). A position q, which is associated with some element e in a container, does not change, even if the index of e changes in the container, unless we explicitly remove e. If the associated node is removed, we say that q is ***invalidated***. Moreover, the position q does not change even if we replace or swap the element e stored at q with another element.
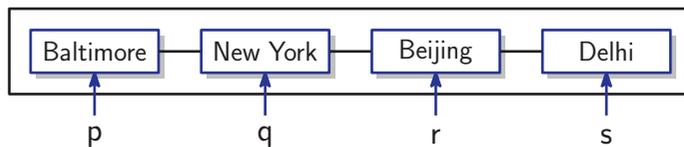
Figure 6.4: A list container. The positions in the current order are p, q, r, and s.

Iterators

Although a position is a useful object, it would be more useful still to be able to navigate through the container, for example, by advancing to the next position in the container. Such an object is called an ***iterator***. An iterator is an extension of a position. It supports the ability to access a node's element, but it also provides the ability to navigate forwards (and possibly backwards) through the container.

There are a number of ways in which to define an ADT for an iterator object. For example, given an iterator object p, we could define an operation p.next(), which returns an iterator that refers to the node just after p in the container. Because of C++'s ability to overload operators, there is a more elegant way to do

this by overloading the increment operator ("++"). In particular, the operation $++p$ advances p to the next position of the container. By repeatedly applying this operation, we can step through all the elements of the container. For some implementations of containers, such as a doubly linked list, navigation may be possible both forwards and backwards. If so, we can also overload the decrement operator ("−−") to move the iterator to the previous position in the container.

In addition to navigating through the container, we need some way of initializing an iterator to the first node of a container and determining whether it has gone beyond the end of the container. To do this, we assume that each container provides two special iterator values, *begin* and *end*. The beginning iterator refers to the first position of the container. We think of the ending iterator as referring to an imaginary position that lies *just after* the last node of the container. Given a container object L, the operation $L.\text{begin}()$ returns an instance of the beginning iterator for L, and the operation $L.\text{end}()$ returns an instance of the ending iterator. (See Figure 6.5.)
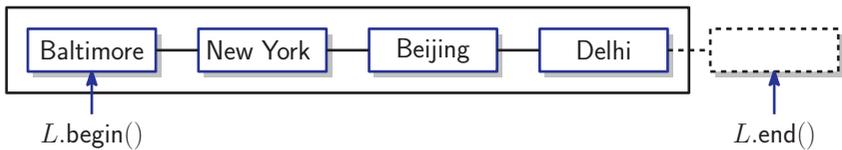
Figure 6.5: The special iterators $L.\text{begin}()$ and $L.\text{end}()$ for a list L.

In order to enumerate all the elements of a given container L, we define an iterator p whose value is initialized to $L.\text{begin}()$. The associated element is accessed using $*p$. We can enumerate all of the elements of the container by advancing p to the next node using the operation $++p$. We repeat this until p is equal to $L.\text{end}()$, which means that we have fallen off the end of the list.

6.2.2 The List Abstract Data Type

Using the concept of an iterator to encapsulate the idea of "node" in a list, we can define another type of sequence ADT, called simply the *list* ADT. In addition to the above functions, we include the generic functions size and empty with the usual meanings. This ADT supports the following functions for a list L and an iterator p for this list.

> begin(): Return an iterator referring to the first element of L; same as end() if L is empty.
>
> end(): Return an iterator referring to an imaginary element just after the last element of L.
>
> insertFront(e): Insert a new element e into L as the first element.

insertBack(*e*): Insert a new element *e* into *L* as the last element.

insert(*p*, *e*): Insert a new element *e* into *L* before position *p* in *L*.

eraseFront(): Remove the first element of *L*.

eraseBack(): Remove the last element of *L*.

erase(*p*): Remove from *L* the element at position *p*; invalidates *p* as a position.

The functions insertFront(*e*) and insertBack(*e*) are provided as a convenience, since they are equivalent to insert(*L*.begin(), *e*) and insert(*L*.end(), *e*), respectively. Similarly, eraseFront and eraseBack can be performed by the more general function erase.

An error condition occurs if an invalid position is passed as an argument to one of the list operations. Reasons for a position *p* to be invalid include:

- *p* was never initialized or was set to a position in a different list
- *p* was previously removed from the list
- *p* results from an illegal operation, such as attempting to perform ++*p*, where *p* = *L*.end(), that is, attempting to access a position beyond the end position

We do not check for these errors in our implementation. Instead, it is the responsibility of the programmer to be sure that only legal positions are used.

Example 6.3: *We show a series of operations for an initially empty list L below. We use variables p and q to denote different positions, and we show the object currently stored at such a position in parentheses in the Output column.*

Operation	*Output*	*L*
insertFront(8)	−	(8)
p = begin()	*p* : (8)	(8)
insertBack(5)	−	(8, 5)
q = *p*; ++*q*	*q* : (5)	(8, 5)
p == begin()	*true*	(8, 5)
insert(*q*, 3)	−	(8, 3, 5)
q = 7	−	(8, 3, 7)
insertFront(9)	−	(9, 8, 3, 7)
eraseBack()	−	(9, 8, 3)
erase(*p*)	−	(9, 3)
eraseFront()	−	(3)

The list ADT, with its built-in notion of position, is useful in a number of settings. For example, a program that models several people playing a game of cards could model each person's hand as a list. Since most people like to keep cards of the same suit together, inserting and removing cards from a person's hand could

be implemented using the functions of the list ADT, with the positions being determined by a natural ordering of the suits. Likewise, a simple text editor embeds the notion of positional insertion and removal, since such editors typically perform all updates relative to a *cursor*, which represents the current position in the list of characters of text being edited.

6.2.3 Doubly Linked List Implementation

There are a number of different ways to implement our list ADT in C++. Probably the most natural and efficient way is to use a doubly linked list, similar to the one we introduced in Section 3.3. Recall that our doubly linked list structure is based on two *sentinel nodes*, called the *header* and *trailer*. These are created when the list is first constructed. The other elements of the list are inserted between these sentinels.

Following our usual practice, in order to keep the code simple, we sacrifice generality by forgoing the use of class templates. Instead, we provide a type definition Elem, which is the base element type of the list. We leave the details of producing a fully generic templated class as an exercise (R-6.11).

Before defining the class, which we call NodeList, we define two important structures. The first represents a node of the list and the other represents an iterator for the list. Both of these objects are defined as nested classes within NodeList. Since users of the class access nodes exclusively through iterators, the node is declared a private member of NodeList, and the iterator is a public member.

The node object is called Node and is presented in Code Fragment 6.6. This is a simple C++ structure, which has only (public) data members, consisting of the node's element, a link to the previous node of the list, and a link to the next node of the list. Since it is declared to be private to NodeList, its members are accessible only within NodeList.

```
struct Node {                          // a node of the list
  Elem elem;                           // element value
  Node* prev;                          // previous in list
  Node* next;                          // next in list
};
```

Code Fragment 6.6: The declaration of a node of a doubly linked list.

Our iterator object is called Iterator. To users of class NodeList, it can be accessed by the qualified type name NodeList::Iterator. Its definition, which is presented in Code Fragment 6.7, is placed in the public part of NodeList. An element associated with an iterator can be accessed by overloading the dereferencing operator ("*"). In order to make it possible to compare iterator objects, we overload the

equality and inequality operators ("==" and "!="). We provide the ability to move forward or backward in the list by providing the increment and decrement operators ("++" and "−−"). We declare NodeList to be a friend, so that it may access the private members of Iterator. The private data member consists of a pointer *v* to the associated node of the list. We also provide a private constructor, which initializes the node pointer. (The constructor is private so that only NodeList is allowed to create new iterators.)

```
class Iterator {                            // an iterator for the list
public:
  Elem& operator*();                        // reference to the element
  bool operator==(const Iterator& p) const; // compare positions
  bool operator!=(const Iterator& p) const;
  Iterator& operator++();                   // move to next position
  Iterator& operator−−();                   // move to previous position
  friend class NodeList;                    // give NodeList access
private:
  Node* v;                                  // pointer to the node
  Iterator(Node* u);                        // create from node
};
```

Code Fragment 6.7: Class Iterator, realizing an iterator for a doubly linked list.

In Code Fragment 6.8 we present the implementations of the member functions for the Iterator class. These all follow directly from the definitions given earlier.

```
NodeList::Iterator::Iterator(Node* u)       // constructor from Node*
  { v = u; }

Elem& NodeList::Iterator::operator*()       // reference to the element
  { return v−>elem; }
                                            // compare positions
bool NodeList::Iterator::operator==(const Iterator& p) const
  { return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const
  { return v != p.v; }
                                            // move to next position
NodeList::Iterator& NodeList::Iterator::operator++()
  { v = v−>next; return *this; }
                                            // move to previous position
NodeList::Iterator& NodeList::Iterator::operator−−()
  { v = v−>prev; return *this; }
```

Code Fragment 6.8: Implementations of the Iterator member functions.

To keep the code simple, we have not implemented any error checking. We assume that the functions of Code Fragment 6.8 are defined outside the class body. Because of this, when referring to the nested class Iterator, we need to apply the scope resolution operator, as in NodeList::Iterator, so the compiler knows that we are referring to the iterator type associated with NodeList. Observe that the increment and decrement operators not only update the position, but they also return a reference to the updated position. This makes it possible to use the result of the increment operation, as in "$q = ++p$."

Having defined the supporting structures Node and Iterator, let us now present the declaration of class NodeList, which is given in Code Fragment 6.9. The class declaration begins by inserting the Node and Iterator definitions from Code Fragments 6.6 and 6.7. This is followed by the public members, that consist of a simple default constructor and the members of the list ADT. We have omitted the standard housekeeping functions from our class definition. These include the class destructor, a copy constructor, and an assignment operator. The definition of the destructor is important, since this class allocates memory, so it is necessary to delete this memory when an object of this type is destroyed. We leave the implementation of these housekeeping functions as an exercise (R-6.12).

```
typedef int Elem;                        // list base element type
class NodeList {                         // node-based list
private:
  // insert Node declaration here...
public:
  // insert Iterator declaration here...
public:
  NodeList();                            // default constructor
  int size() const;                      // list size
  bool empty() const;                    // is the list empty?
  Iterator begin() const;                // beginning position
  Iterator end() const;                  // (just beyond) last position
  void insertFront(const Elem& e);       // insert at front
  void insertBack(const Elem& e);        // insert at rear
  void insert(const Iterator& p, const Elem& e); // insert e before p
  void eraseFront();                     // remove first
  void eraseBack();                      // remove last
  void erase(const Iterator& p);         // remove p
  // housekeeping functions omitted...
private:                                 // data members
  int      n;                            // number of items
  Node*  header;                         // head-of-list sentinel
  Node*  trailer;                        // tail-of-list sentinel
};
```

Code Fragment 6.9: Class NodeList realizing the C++-based list ADT.

The private data members include pointers to the header and trailer sentinel nodes. In order to implement the function size efficiently, we also provide a variable *n*, which stores the number of elements in the list.

Next, let us see how the various functions of class NodeList are implemented. In Code Fragment 6.10, we begin by presenting a number of simple functions, including the constructor, the size and empty functions, and the begin and end functions. The constructor creates an initially empty list by setting *n* to zero, then allocating the header and trailer nodes and linking them together. The function begin returns the position of the first node of the list, which is the node just following the header sentinel, and the function end returns the position of the trailer. As desired, this is the position following the last element of the list. In both cases, we are invoking the private constructor declared within class Iterator. We are allowed to do so because NodeList is a friend of Iterator.

```
NodeList::NodeList() {                        // constructor
  n = 0;                                      // initially empty
  header = new Node;                          // create sentinels
  trailer = new Node;
  header->next = trailer;                     // have them point to each other
  trailer->prev = header;
}

int NodeList::size() const                    // list size
  { return n; }

bool NodeList::empty() const                  // is the list empty?
  { return (n == 0); }

NodeList::Iterator NodeList::begin() const    // begin position is first item
  { return Iterator(header->next); }

NodeList::Iterator NodeList::end() const      // end position is just beyond last
  { return Iterator(trailer); }
```

Code Fragment 6.10: Implementations of a number of simple member functions of class NodeList.

Let us next see how to implement insertion. There are three different public insertion functions, insert, insertFront, and insertBack, which are shown in Code Fragment 6.11. The function insert(p, e) performs insertion into the doubly linked list using the same approach that we was explained in Section 3.3. In particular, let *w* be a pointer to *p*'s node, let *u* be a pointer to *p*'s predecessor. We create a new node *v*, and link it before *w* and after *u*. Finally, we increment *n* to indicate that there is one additional element in the list.

```
                                              // insert e before p
void NodeList::insert(const NodeList::Iterator& p, const Elem& e) {
  Node* w = p.v;                              // p's node
  Node* u = w->prev;                          // p's predecessor
  Node* v = new Node;                         // new node to insert
  v->elem = e;
  v->next = w; w->prev = v;                   // link in v before w
  v->prev = u; u->next = v;                   // link in v after u
  n++;
}

void NodeList::insertFront(const Elem& e)  // insert at front
  { insert(begin(), e); }

void NodeList::insertBack(const Elem& e)   // insert at rear
  { insert(end(), e); }
```

Code Fragment 6.11: Implementations of the insertion functions of class NodeList.

The function insertFront invokes insert on the beginning of the list, and the function insertBack invokes insert on the list's trailer.

Finally, in Code Fragment 6.12 we present the implementation of the erase function, which removes a node from the list. Again, our approach follows directly from the method described in Section 3.3 for removal of a node from a doubly linked list. Let *v* be a pointer to the node to be deleted, and let *w* be its successor and *u* be its predecessor. We unlink *v* by linking *u* and *w* to each other. Once *v* has been unlinked from the list, we need to return its allocated storage to the system in order to avoid any memory leaks. Finally, we decrement the number of elements in the list.

```
void NodeList::erase(const Iterator& p) {  // remove p
  Node* v = p.v;                            // node to remove
  Node* w = v->next;                        // successor
  Node* u = v->prev;                        // predecessor
  u->next = w; w->prev = u;                 // unlink p
  delete v;                                 // delete this node
  n--;                                      // one fewer element
}

void NodeList::eraseFront()                // remove first
  { erase(begin()); }

void NodeList::eraseBack()                 // remove last
  { erase(--end()); }
```

Code Fragment 6.12: Implementations of the function erase of class NodeList.

There are number of other enhancements that could be added to our implementation of NodeList, such as better error checking, a more sophisticated suite of constructors, and a post-increment operator for the iterator class.

You might wonder why we chose to define the iterator function end to return an imaginary position that lies just beyond the end of the list, rather than the last node of the list. This choice offers a number of advantages. First, it is well defined, even if the list is empty. Second, the function insert(p, e) can be used to insert a new element at any position of the list. In particular, it is possible to insert an element at the end of the list by invoking insert(end(), e). If we had instead defined end to return the last position of the list, this would not be possible, and the only way to insert an element at the end of the list would be through the insertBack function.

Observe that our implementation is quite efficient with respect to both time and space. All of the operations of the list ADT run in time $O(1)$. (The only exceptions to this are the omitted housekeeping functions, the destructor, copy constructor, and assignment operator. They require $O(n)$ time, where n is the number of elements in the list.) The space used by the data structure is proportional to the number of elements in the list.

6.2.4   STL Lists

The C++ Standard Template Library provides an implementation of a list, which is called list. Like the STL vector, the STL list is an example of an STL container. As in our implementation of class NodeList, the STL list is implemented as a doubly linked list.

In order to define an object to be of type list, it is necessary to first include the appropriate system definition file, which is simply called "list." As with the STL vector, the list class is a member of the std namespace, it is necessary either to preface references to it with the namespace resolution operator, as in "std::list", or to provide an appropriate **using** statement. The list class is templated with the ***base type*** of the individual elements. For example, the code fragment below declares a list of floats. By default, the initial list is empty.

```
#include <list>
using std::list;                        // make list accessible
list<float> myList;                     // an empty list of floats
```

Below is a list of the principal member functions of the list class. Let L be declared to be an STL list of some base type, and let x denote a single object of this same base type. (For example, L is a list of integers, and e is an integer.)

list(n): Construct a list with n elements; if no argument list is given, an empty list is created.

size(): Return the number of elements in L.

empty(): Return true if *L* is empty and false otherwise.

front(): Return a reference to the first element of *L*.

back(): Return a reference to the last element of *L*.

push_front(*e*): Insert a copy of *e* at the beginning of *L*.

push_back(*e*): Insert a copy of *e* at the end of *L*.

pop_front(): Remove the fist element of *L*.

pop_back(): Remove the last element of *L*.

The functions push_front and push_back are the STL equivalents of our functions insertFront and insertBack, respectively, of our list ADT. Similarly, the functions pop_front and pop_back are equivalent to the respective functions eraseFront and eraseBack.

Note that, when the base type of an STL vector is class object, all copying of elements (for example, in push_back) is performed by invoking the base class's copy constructor. Whenever elements are destroyed (for example, by invoking the destroyer or the pop_back member function) the class's destructor is invoked on each deleted element.

6.2.5 STL Containers and Iterators

In order to develop a fuller understanding of STL vectors and lists, it is necessary to understand the concepts of STL ***containers*** and ***iterators***. Recall that a container is a data structure that stores a collection of elements. The STL provides a variety of different container classes, many of which are discussed later.

STL Container	*Description*
vector	Vector
deque	Double ended queue
list	List
stack	Last-in, first-out stack
queue	First-in, first-out queue
priority_queue	Priority queue
set (and multiset)	Set (and multiset)
map (and multimap)	Map (and multi-key map)

Different containers organize their elements in different ways, and hence support different methods for accessing individual elements. STL iterators provide a relatively uniform method for accessing and enumerating the elements stored in containers.

Before introducing how iterators work for STL containers, let us begin with a simple function that sums the elements of an STL vector, denoted by *V*, shown in

Code Fragment 6.13. The elements are accessed in the standard manner through the indexing operator.

```
int vectorSum1(const vector<int>& V) {
    int sum = 0;
    for (int i = 0; i < V.size(); i++)
        sum += V[i];
    return sum;
}
```

Code Fragment 6.13: A simple C++ function that sums the entries of an STL vector.

This particular method of iterating through the elements of a vector should be quite familiar by now. Unfortunately, this method would not be applicable to other types of containers, because it relies on the fact that the elements of a vector can be accessed efficiently through indexing. This is not true for all containers, such as lists. What we desire is a uniform mechanism for accessing elements.

STL Iterators

Every STL container class defines a special associated class called an *iterator*. As mentioned earlier, an iterator is an object that specifies a position within a container and which is endowed with the ability to navigate to other positions. If p is an iterator that refers to some position within a container, then $*p$ yields a reference to the associated element.

Advancing to the next element of the container is done by incrementing the iterator. For example, either $++p$ or $p++$ advances p to point to the next element of the container. The former returns the updated value of the iterator, and the latter returns its original value. (In our implementation of an iterator for class NodeList in Code Fragment 6.7, we defined only the preincrement operator, but the postincrement operator would be an easy extension. See Exercise R-6.13.)

Each STL container class provides two member functions, begin and end, each of which returns an iterator for this container. The first returns an iterator that points to the first element of the container, and the second returns an iterator that can be thought of as pointing to an imaginary element *just beyond* the last element of the container. An example for the case of lists was shown in Figure 6.5, and an example of how this works for the STL vector is shown in Figure 6.6.
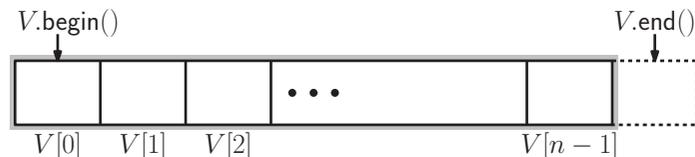
Figure 6.6: The special iterators $V.\text{begin}()$ and $V.\text{end}()$ for an STL vector V.

Using Iterators

Let us see how we can use iterators to enumerate the elements of an STL container *C*. Suppose, for example, that *C* is of type vector**<int>**, that is, it is an STL list of integers. The associated iterator type is denoted "vector**<int>**::iterator." In general, if *C* is an STL container of some type cont and the base type is of type base, then the iterator type would be denoted "cont<base>::iterator."

For example, Code Fragment 6.14 demonstrates how to sum the elements of an STL vector *V* using an iterator. We begin by providing a type definition to the iterator type, called Iterator. We then create a loop, which is controlled by an iterator *p*. We start with *V*.begin(), and we terminate when *p* reaches *V*.end(). Although this approach is less direct than the approach based on indexing individual elements, it has the advantage that it can be applied to *any* STL container class, not just vectors.

```
int vectorSum2(vector<int> V) {
  typedef vector<int>::iterator Iterator;          // iterator type
  int sum = 0;
  for (Iterator p = V.begin(); p != V.end(); ++p)
    sum += *p;
  return sum;
}
```

Code Fragment 6.14: Using an iterator to sum the elements of an STL vector.

Different containers provide iterators with different capabilities. Most STL containers (including lists, sets, and maps) provide the ability to move not only forwards, but backwards as well. For such containers the decrement operators $--p$ and $p--$ are also defined for their iterators. This is called a *bidirectional iterator*.

A few STL containers (including vectors and deques) support the additional feature of allowing the addition and subtraction of an integer. For example, for such an iterator, *p*, the value $p + 3$ references the element three positions after *p* in the container. This is called a *random-access iterator*.

As with pointers, care is needed in the use of iterators. For example, it is up to the programmer to be sure that an iterator points to a valid element of the container before attempting to dereference it. Attempting to dereference an invalid iterator can result in your program aborting. As mentioned earlier, iterators can be *invalid* for various reasons. For example, an iterator becomes invalid if the position that it refers to is deleted.

Const Iterators

Observe that in Code Fragment 6.14, we passed the vector *V* into the function *by value* (recall Section 1.4). This can be quite inefficient, because the system constructs a complete copy of the actual argument. Since our function does not

modify *V*, the best approach would be to declare the argument to be a constant reference instead, that is, "**const** vector**<int>**&."

A problem arises, however, if we declare an iterator for such a vector. Many STL implementations generate an error message if we attempt to use a regular iterator with a constant vector reference, since such an iterator may lead to an attempt to modify the vector's contents.

The solution is a special read-only iterator, called a ***const iterator***. When using a const iterator, it is possible to read the values of the container by dereferencing the iterator, but it is not possible to modify the container's values. For example, if *p* is a const iterator, it is possible to read the value of *∗p*, but you cannot assign it a value. The const iterator type for our vector type is denoted "vector**<int>**::const_iterator." We make use of the **typedef** command to rename this lengthy definition to the more concise ConstIterator. The final code fragment is presented in Code Fragment 6.15.

```
int vectorSum3(const vector<int>& V) {
  typedef vector<int>::const_iterator ConstIterator;  // iterator type
  int sum = 0;
  for (ConstIterator p = V.begin(); p != V.end(); ++p)
    sum += *p;
  return sum;
}
```

Code Fragment 6.15: Using a constant iterator to sum the elements of a vector.

STL Iterator-Based Container Functions

STL iterators offer a flexible and uniform way to access the elements of STL containers. Many of the member functions and predefined algorithms that work with STL containers use iterators as their arguments. Here are a few of the member functions of the STL vector class that use iterators as arguments. Let *V* be an STL vector of some given base type, and let *e* be an object of this base type. Let *p* and *q* be iterators over this base type, both drawn from the same container.

vector(*p*, *q*): Construct a vector by iterating between *p* and *q*, copying each of these elements into the new vector.

assign(*p*, *q*): Delete the contents of *V*, and assigns its new contents by iterating between *p* and *q* and copying each of these elements into *V*.

insert(*p*, *e*): Insert a copy of *e* just prior to the position given by iterator *p* and shifts the subsequent elements one position to the right.

erase(*p*): Remove and destroy the element of *V* at the position given by *p* and shifts the subsequent elements one po-

sition to the left.

erase(p, q): Iterate between p and q, removing and destroying all these elements and shifting subsequent elements to the left to fill the gap.

clear(): Delete all these elements of V.

When presenting a range of iterator values (as we have done above with the constructor $V(p, q)$, assign(p, q), and erase(p, q)), the iterator range is understood to start with p and end **just prior** to q. Borrowing from interval notation in mathematics, this iterator range is often expressed as $[p, q)$, implying that p is included in the range, but q is not. This convention holds whenever dealing with iterator ranges.

Note that the vector member functions insert and erase move elements around in the vector. They should be used with care, because they can be quite slow. For example, inserting or erasing an element at the beginning of a vector causes all the later elements of the vector to be shifted one position.

The above functions are also defined for the STL list and the STL deque (but, of course, the constructors are named differently). Since the list is defined as a doubly linked list, there is no need to shift elements when performing insert or erase. These three STL containers (vector, list, and deque) are called **sequence containers**, because they explicitly store elements in sequential order. The STL containers set, multiset, map, and multimap support all of the above functions except assign. They are called **associated containers**, because elements are typically accessed by providing an associated key value. We discuss them in Chapter 9.

It is worthwhile noting that, in the constructor and assignment functions, the iterators p and q do not need to be drawn from the same type of container as V, as long as the container they are drawn from has the same base type. For example, suppose that L is an STL list container of integers. We can create a copy of L in the form of an STL vector V as follows:

```
list<int> L;                    // an STL list of integers
// ...
vector<int> V(L.begin(), L.end());   // initialize V to be a copy of L
```

The iterator-based form of the constructor is quite handy, since it provides an easy way to initialize the contents of an STL container from a standard C++ array. Here, we make use of a low-level feature of C++, which is inherited from its predecessor, the C programming language. Recall from Section 1.1.3 that a C++ array A is represented as a pointer to its first element $A[0]$. In addition, $A + 1$ points to $A[1]$, $A + 2$ points to $A[2]$, and generally $A + i$ points to $A[i]$.

Addressing the elements of an array in this manner is called **pointer arithmetic**. It is generally frowned upon as poor programming practice, but in this instance it

provides an elegant way to initialize a vector from an C++ array, as follows.

```
int A[] = {2, 5, −3, 8, 6};        // a C++ array of 5 integers
vector<int> V(A, A+5);             // V = (2, 5, -3, 8, 6)
```

Even though the pointers A and $A+5$ are not STL iterators, through the magic of pointer arithmetic, they essentially behave as though they were. This same trick can be used to initialize any of the STL sequence containers.

STL Vectors and Algorithms

In addition to the above member functions for STL vectors, the STL also provides a number of algorithms that operate on containers in general, and vectors in particular. To access these functions, use the include statement "#include <algorithm>." Let p and q be iterators over some base type, and let e denote an object of this base type. As above, these operations apply to the iterator range $[p,q)$, which starts at p and ends just prior to q.

sort(p,q): Sort the elements in the range from p to q in ascending order. It is assumed that less-than operator ("<") is defined for the base type.

random_shuffle(p,q): Rearrange the elements in the range from p to q in random order.

reverse(p,q): Reverse the elements in the range from p to q.

find(p,q,e): Return an iterator to the first element in the range from p to q that is equal to e; if e is not found, q is returned.

min_element(p,q): Return an iterator to the minimum element in the range from p to q.

max_element(p,q): Return an iterator to the maximum element in the range from p to q.

for_each(p,q,f): Apply the function f the elements in the range from p to q.

For example, to sort an entire vector V, we would use sort$(V.\text{begin}(),V.\text{end}())$. To sort just the first 10 elements, we could use sort$(V.\text{begin}(),V.\text{begin}()+10)$.

All of the above functions are supported for the STL deque. All of the above functions, except sort and random_shuffle are supported for the STL list.

An Illustrative Example

In Code Fragment 6.16, we provide a short example program of the functionality of the STL vector class. The program begins with the necessary "#include"

statements. We include <vector>, for the definitions of vectors and iterators, and <algorithm>, for the definitions of sort and random_shuffle.

We first initialize a standard C++ array containing six integers, and we then use the iterator-based constructor to create a six-element vector containing these values. We show how to use the member functions size, pop_back, push_back, front, and back. Observe that popping decreases the array size and pushing increases the array size. We then show how to use iterator arithmetic to sort a portion of the vector, in this case, the first four elements. The call to the erase member function removes two of the last four elements of the vector. After the removal, the remaining two elements at the end have been shifted forward to fill the empty positions.

Next, we demonstrate how to generate a vector of characters. We apply the function random_shuffle to permute the elements of the vector randomly. Finally, we show how to use the member function insert, to insert a character at the beginning of the vector. Observe how the other elements are shifted to the right.

```cpp
#include <cstdlib>                              // provides EXIT_SUCCESS
#include <iostream>                             // I/O definitions
#include <vector>                               // provides vector
#include <algorithm>                            // for sort, random_shuffle

using namespace std;                            // make std:: accessible

int main () {
  int a[] = {17, 12, 33, 15, 62, 45};
  vector<int> v(a, a + 6);                      // v: 17 12 33 15 62 45
  cout << v.size() << endl;                     // outputs: 6
  v.pop_back();                                 // v: 17 12 33 15 62
  cout << v.size() << endl;                     // outputs: 5
  v.push_back(19);                              // v: 17 12 33 15 62 19
  cout << v.front() << " " << v.back() << endl; // outputs: 17 19
  sort(v.begin(), v.begin() + 4);               // v: (12 15 17 33) 62 19
  v.erase(v.end() − 4, v.end() − 2);            // v: 12 15 62 19
  cout << v.size() << endl;                     // outputs: 4

  char b[] = {'b', 'r', 'a', 'v', 'o'};
  vector<char> w(b, b + 5);                     // w: b r a v o
  random_shuffle(w.begin(), w.end());           // w: o v r a b
  w.insert(w.begin(), 's');                     // w: s o v r a b

  for (vector<char>::iterator p = w.begin(); p != w.end(); ++p)
    cout << *p << " ";                          // outputs: s o v r a b
  cout << endl;
  return EXIT_SUCCESS;
}
```

Code Fragment 6.16: An example of the use of the STL vector and iterators.

6.3 Sequences

In this section, we define an abstract data type that generalizes the vector and list ADTs. This ADT therefore provides access to its elements using both indices and positions, and is a versatile data structure for a wide variety of applications.

6.3.1 The Sequence Abstract Data Type

A *sequence* is an ADT that supports all the functions of the list ADT (discussed in Section 6.2), but it also provides functions for accessing elements by their index, as we did in the vector ADT (discussed in Section 6.1). The interface consists of the operations of the list ADT, plus the following two "bridging" functions, which provide connections between indices and positions.

atIndex(i): Return the position of the element at index i.

indexOf(p): Return the index of the element at position p.

6.3.2 Implementing a Sequence with a Doubly Linked List

One possible implementation of a sequence, of course, is with a doubly linked list. By doing so, all of the functions of the list ADT can be easily implemented to run in $O(1)$ time each. The functions atIndex and indexOf from the vector ADT can also be implemented with a doubly linked list, though in a less efficient manner.

In particular, if we want the functions from the list ADT to run efficiently (using position objects to indicate where accesses and updates should occur), then we can no longer explicitly store the indices of elements in the sequence. Hence, to perform the operation atIndex(i), we must perform link "hopping" from one of the ends of the list until we locate the node storing the element at index i. As a slight optimization, we could start hopping from the closest end of the sequence, which would achieve a running time of

$$O(\min(i+1, n-i)).$$

This is still $O(n)$ in the worst case (when i is near the middle index), but it would be more efficient in applications where many calls to atIndex(i) are expected to involve index values that are significantly closer to either end of the list. In our implementation, we just use the simple approach of walking from the front of the list, and we leave the two-sided solution as an exercise (R-6.14).

In Code Fragment 6.17, we present a definition of a class NodeSequence, which implements the sequence ADT. Observe that, because a sequence extends the definition of a list, we have inherited our class by extending the NodeList class

that was introduced in Section 6.2. We simply add definitions of our bridging functions. Through the magic of inheritance, users of our class NodeSequence have access to all the members of the NodeList class, including its nested class, NodeList::Iterator.

```
class NodeSequence : public NodeList {
public:
  Iterator atIndex(int i) const;            // get position from index
  int indexOf(const Iterator& p) const;     // get index from position
};
```

Code Fragment 6.17: The definition of class NodeSequence, which implements the sequence ADT using a doubly linked list.

Next, in Code Fragment 6.18, we show the implementations of the atIndex and indexOf member functions. The function atIndex(i) hops i positions to the right, starting at the beginning, and returns the resulting position. The function indexOf hops through the list until finding the position that matches the given position p. Observe that the conditional "q != p" uses the overloaded comparison operator for positions defined in Code Fragment 6.8.

```
                                            // get position from index
NodeSequence::Iterator NodeSequence::atIndex(int i) const {
  Iterator p = begin();
  for (int j = 0; j < i; j++) ++p;
  return p;
}
                                            // get index from position
int NodeSequence::indexOf(const Iterator& p) const {
  Iterator q = begin();
  int j = 0;
  while (q != p) {                          // until finding p
    ++q; ++j;                               // advance and count hops
  }
  return j;
}
```

Code Fragment 6.18: Definition of the functions atIndex and indexOf of class Node-Sequence.

Both of these functions are quite fragile, and are likely to abort if their arguments are not in bounds. A more careful implementation of atIndex would first check that the argument i lies in the range from 0 to $n-1$, where n is the size of the sequence. The function indexOf should check that it does not run past the end of the sequence. In either case, an appropriate exception should be thrown.

The worst-case running times of both of the functions atIndex and indexOf are $O(n)$, where n is the size of the list. Although this is not very efficient, we may take consolation in the fact that all the other operations of the list ADT run in time $O(1)$. A natural alternative approach would be to implement the sequence ADT using an array. Although we could now provide very efficient implementations of atIndex and indexOf, the insertion and removal operations of the list ADT would now require $O(n)$ time. Thus, neither solution is perfect under all circumstances. We explore this alternative in the next section.

6.3.3 Implementing a Sequence with an Array

Suppose we want to implement a sequence S by storing each element e of S in a cell $A[i]$ of an array A. We can define a position object p to hold an index i and a reference to array A, as member variables. We can then implement function element(p) by simply returning a reference to $A[i]$. A major drawback with this approach, however, is that the cells in A have no way to reference their corresponding positions. For example, after performing an insertFront operation, the elements have been shifted to new positions, but we have no way of informing the existing positions for S that the associated positions of their elements have changed. (Remember that positions in a sequence are defined relative to their neighboring positions, not their ranks.) Hence, if we are going to implement a general sequence with an array, we need to find a different approach.

Consider an alternate solution in which, instead of storing the elements of S in array A, we store a pointer to a new kind of position object in each cell of A. Each new position object p stores a pair consisting of the index i and the element e associated with p. We can easily scan through the array to update the i value associated with each position whose rank changes as the result of an insertion or deletion. An example is shown in Figure 6.7, where a new element containing BWI is inserted at index 1 of an existing sequence. After the insertion, the elements PVD and SFO are shifted to the right, so we increment the index value associated with their index-element pairs.
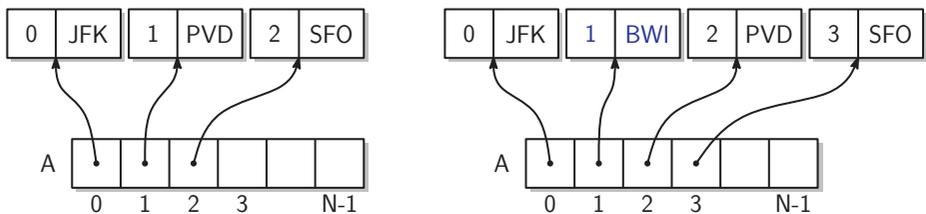
Figure 6.7: An array-based implementation of the sequence ADT.

In this array-based implementation of a sequence, the functions insertFront, insert, and erase take $O(n)$ time because we have to shift position objects to make room for the new position or to fill in the hole created by the removal of the old position (just as in the insert and remove functions based on rank). All the other position-based functions take $O(1)$ time.

Note that we can use an array in a circular fashion, as we did for implementing a queue (see Section 5.2.4). With a little work, we can then perform functions insert-Front in $O(1)$ time. Note that functions insert and erase still take $O(n)$ time. Now, the worst case occurs when the element to be inserted or removed has rank $\lfloor n/2 \rfloor$.

Table 6.2 compares the running times of the implementations of the general sequence ADT, by means of a circular array and a doubly linked list.

Operations	*Circular Array*	*List*
size, empty	$O(1)$	$O(1)$
atIndex, indexOf	$O(1)$	$O(n)$
begin, end	$O(1)$	$O(1)$
$*p, ++p, --p$	$O(1)$	$O(1)$
insertFront, insertBack	$O(1)$	$O(1)$
insert, erase	$O(n)$	$O(1)$

Table 6.2: Comparison of the running times of the functions of a sequence implemented with either an array (used in a circular fashion) or a doubly linked list. We denote with n the number of elements in the sequence at the time the operation is performed. The space usage is $O(n)$ for the doubly linked list implementation, and $O(N)$ for the array implementation, where N is the size of the array.

Summarizing this table, we see that the array-based implementation is superior to the linked-list implementation on the rank-based access operations, atIndex and indexOf. It is equal in performance to the linked-list implementation on all the other access operations. Regarding update operations, the linked-list implementation beats the array-based implementation in the position-based update operations, insert and erase. For update operations insertFront and insertBack, the two implementations have comparable performance.

Considering space usage, note that an array requires $O(N)$ space, where N is the size of the array (unless we utilize an extendable array), while a doubly linked list uses $O(n)$ space, where n is the number of elements in the sequence. Since n is less than or equal to N, this implies that the asymptotic space usage of a linked-list implementation is superior to that of a fixed-size array, although there is a small constant factor overhead that is larger for linked lists, since arrays do not need links to maintain the ordering of their cells.

6.4 Case Study: Bubble-Sort on a Sequence

In this section, we illustrate the use of the sequence ADT and its implementation trade-offs with sample C++ functions using the well-known *bubble-sort* algorithm.

6.4.1 The Bubble-Sort Algorithm

Consider a sequence of n elements such that any two elements in the sequence can be compared according to an order relation (for example, companies compared by revenue, states compared by population, or words compared lexicographically). The *sorting* problem is to reorder the sequence so that the elements are in non-decreasing order. The *bubble-sort* algorithm (see Figure 6.8) solves this problem by performing a series of *passes* over the sequence. In each pass, the elements are scanned by increasing rank, from rank 0 to the end of the sequence. At each position in a pass, an element is compared with its neighbor, and if these two consecutive elements are found to be in the wrong relative order (that is, the preceding element is larger than the succeeding one), then the two elements are swapped. The sequence is sorted by completing n such passes.

pass	swaps	sequence
		$(5,7,2,6,9,3)$
1st	$7 \leftrightarrow 2\ \ 7 \leftrightarrow 6\ \ 9 \leftrightarrow 3$	$(5,2,6,7,3,9)$
2nd	$5 \leftrightarrow 2\ \ 7 \leftrightarrow 3$	$(2,5,6,3,7,9)$
3rd	$6 \leftrightarrow 3$	$(2,5,3,6,7,9)$
4th	$5 \leftrightarrow 3$	$(2,3,5,6,7,9)$

Figure 6.8: The bubble-sort algorithm on a sequence of integers. For each pass, the swaps performed and the sequence after the pass are shown.

The bubble-sort algorithm has the following properties:

- In the first pass, once the largest element is reached, it keeps on being swapped until it gets to the last position of the sequence.
- In the second pass, once the second largest element is reached, it keeps on being swapped until it gets to the second-to-last position of the sequence.
- In general, at the end of the ith pass, the right-most i elements of the sequence (that is, those at indices from $n-1$ down to $n-i$) are in final position.

The last property implies that it is correct to limit the number of passes made by a bubble-sort on an n-element sequence to n. Moreover, it allows the ith pass to be limited to the first $n-i+1$ elements of the sequence.

6.4.2   A Sequence-Based Analysis of Bubble-Sort

Assume that the implementation of the sequence is such that the accesses to elements and the swaps of elements performed by bubble-sort take $O(1)$ time each. That is, the running time of the ith pass is $O(n - i + 1)$. We have that the overall running time of bubble-sort is

$$O\left(\sum_{i=1}^{n}(n - i + 1)\right).$$

We can rewrite the sum inside the big-Oh notation as

$$n + (n - 1) + \cdots + 2 + 1 = \sum_{i=1}^{n} i.$$

By Proposition 4.3, we have

$$\sum_{i=1}^{n} i = \frac{n(n + 1)}{2}.$$

Thus, bubble-sort runs in $O(n^2)$ time, provided that accesses and swaps can each be implemented in $O(1)$ time. As we see in future chapters, this performance for sorting is quite inefficient. We discuss the bubble-sort algorithm here. Our aim is to demonstrate, not as an example of a good sorting algorithm.

Code Fragments 6.19 and 6.20 present two implementations of bubble-sort on a sequence of integers. The parameter S is of type Sequence, but we do not specify whether it is a node-based or array-based implementation. The two bubble-sort implementations differ in the preferred choice of functions to access and modify the sequence. The first is based on accessing elements by their index. We use the function atIndex to access the two elements of interest.

Since function bubbleSort1 accesses elements only through the index-based interface functions atIndex, this implementation is suitable only for the array-based implementation of the sequence, for which atIndex takes $O(1)$ time. Given such an array-based sequence, this bubble-sort implementation runs in $O(n^2)$ time.

On the other hand, if we had used our node-based implementation of the sequence, each atIndex call would take $O(n)$ time in the worst case. Since this function is called with each iteration of the inner loop, the entire function would run in $O(n^3)$ worst-case time, which is quite slow if n is large.

In contrast to bubbleSort1, our second function, bubbleSort2, accesses the elements entirely through the use of iterators. The iterators *prec* and *succ* play the roles that indices $j - 1$ and j play, respectively, in bubbleSort1. Observe that when we first enter the inner loop of bubbleSort1, the value of $j - 1$ is 0, that is, it refers to the first element of the sequence. This is why we initialize *prec* to the beginning

```
void bubbleSort1(Sequence& S) {                      // bubble-sort by indices
  int n = S.size();
  for (int i = 0; i < n; i++) {                       // i-th pass
    for (int j = 1; j < n−i; j++) {
      Sequence::Iterator prec = S.atIndex(j−1);       // predecessor
      Sequence::Iterator succ = S.atIndex(j);         // successor
      if (*prec > *succ) {                            // swap if out of order
        int tmp = *prec; *prec = *succ; *succ = tmp;
      }
    }
  }
}
```

Code Fragment 6.19: A C++ implementations of bubble-sort based on indices.

of the sequence before entering the inner loop. Whenever we reenter the inner loop, we initialize *succ* to *prec* and then immediately increment it. Thus, *succ* refers to the position immediately after *prec*. Before resuming the loop, we increment *prec*.

```
void bubbleSort2(Sequence& S) {                      // bubble-sort by positions
  int n = S.size();
  for (int i = 0; i < n; i++) {                       // i-th pass
    Sequence::Iterator prec = S.begin();              // predecessor
    for (int j = 1; j < n−i; j++) {
      Sequence::Iterator succ = prec;
      ++succ;                                         // successor
      if (*prec > *succ) {                            // swap if out of order
        int tmp = *prec; *prec = *succ; *succ = tmp;
      }
      ++prec;                                         // advance predecessor
    }
  }
}
```

Code Fragment 6.20: Two C++ implementations of bubble-sort.

Since the iterator increment operator takes $O(1)$ time in either the array-based or node-based implementation of a sequence, this second implementation of bubble-sort would run in $O(n^2)$ worst-case time, irrespective of the manner in which the sequence was implemented.

The two bubble-sort implementations given above show the importance of providing efficient implementations of ADTs. Nevertheless, in spite of its implementation simplicity, computing researchers generally feel that the bubble-sort algorithm is not a good sorting method, because, even if implemented in the best possible way, it still takes quadratic time. Indeed, there are much more efficient sorting algorithms that run in $O(n \log n)$ time. We explore these in Chapters 8 and 11.

6.5   Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-6.1 Give a C++ code fragment for reversing an array.

R-6.2 Give a C++ code fragment for randomly permuting an array.

R-6.3 Give a C++ code fragment for circularly rotating an array by distance d.

R-6.4 Draw a representation of an initially empty vector A after performing the following sequence of operations: insert(0,4), insert(0,3), insert(0,2), insert(2,1), insert(1,5), insert(1,6), insert(3,7), insert(0,8).

R-6.5 Give an adapter class to support the Stack interface using the functions of the vector ADT.

R-6.6 Provide the missing housekeeping functions (copy constructor, assignment operator, and destructor) for the class ArrayVector of Code Fragment 6.2.

R-6.7 Provide a fully generic version of the class ArrayVector of Code Fragment 6.2 using a templated class.

R-6.8 Give a templated C++ function sum(v) that returns the sum of elements in an STL vector v. Use an STL iterator to enumerate the elements of v. Assume that the element type of v is any numeric type that supports the $+$ operator.

R-6.9 Rewrite the justification of Proposition 6.2 under the assumption that the cost of growing the array from size k to size $2k$ is $3k$ cyber-dollars. How much should each push operation be charged to make the amortization work?

R-6.10 Draw pictures illustrating each of the major steps in the algorithms for functions insert(p,e), insertFront(e), and insertBack(e) of Code Fragment 6.5.

R-6.11 Provide a fully generic version of the class NodeList of Code Fragment 6.9 using a templated class.

R-6.12 Provide the missing housekeeping functions (copy constructor, assignment operator, and destructor) for the class NodeList, which was presented in Code Fragment 6.9.

R-6.13 In our implementation of an iterator for class NodeList in Code Fragment 6.7, we defined only the preincrement operator. Provide a definition for a postincrement operator.

R-6.14 In our implementation of the atRank(i) function in Code Fragment 6.18 for class NodeSequence, we walked from the front of the list. Present a more efficient implementation, which walks from whichever end of the list is closer to index i.

R-6.15 Provide the details of an array implementation of the list ADT.

R-6.16 Suppose that we have made kn total accesses to the elements in a list L of n elements, for some integer $k \geq 1$. What are the minimum and maximum number of elements that have been accessed fewer than k times?

R-6.17 Give pseudo-code describing how to implement all the operations in the sequence ADT using an array used in a circular fashion. What is the running time for each of these functions?

R-6.18 Using the Sequence interface functions, describe a recursive function for determining if a sequence S of n integer objects contains a given integer k. Your function should not contain any loops. How much space does your function use in addition to the space used for S?

R-6.19 Briefly describe how to perform a new sequence function makeFirst(p) that moves an element of a sequence S at position p to be the first element in S while keeping the relative ordering of the remaining elements in S unchanged. Your function should run in $O(1)$ time if S is implemented with a doubly linked list.

R-6.20 Describe how to implement an iterator for the class ArrayVector of Code Fragment 6.2, based on an integer index. Include pseudo-code fragments describing the dereferencing operator ("*"), equality test ("=="), and increment and decrement ("++" and "−−").

Creativity

C-6.1 Describe what changes need to be made to the extendable array implementation given in Code Fragment 6.2 in order to avoid unexpected termination due to an error. Specify the new types of exceptions you would add, and when and where they should be thrown.

C-6.2 Give complete C++ code for a new class, ShrinkingVector, that extends the ArrayVector class shown in Code Fragment 6.2 and adds a function, shrinkToFit, which replaces the underlying array with an array whose capacity is exactly equal to the number of elements currently in the vector.

C-6.3 Describe what changes need to be made to the extendable array implementation given in Code Fragment 6.2 in order to shrink the size N of the array by half any time the number of elements in the vector goes below $N/4$.

C-6.4 Show that, using an extendable array that grows and shrinks as in the previous exercise, the following series of $2n$ operations takes $O(n)$ time: (i) n push operations on a vector with initial capacity $N = 1$; (ii) n pop (removal of the last element) operations.

C-6.5 Describe a function for performing a *card shuffle* of an array of $2n$ elements, by converting it into two lists. A card shuffle is a permutation where a list L is cut into two lists, L_1 and L_2, where L_1 is the first half of L and L_2 is the second half of L, and then these two lists are merged into one by taking the first element in L_1, then the first element in L_2, followed by the second element in L_1, the second element in L_2, and so on.

C-6.6 Show how to improve the implementation of function insert(i, e) in Code Fragment 6.5 so that, in case of an overflow, the elements are copied into their final place in the new array.

C-6.7 Consider an implementation of the vector ADT using an extendable array, but instead of copying the elements into an array of double the size (that is, from N to $2N$) when its capacity is reached, we copy the elements into an array with $\lceil N/4 \rceil$ additional cells, going from capacity N to $N + \lceil N/4 \rceil$. Show that performing a sequence of n push operations (that is, insertions at the end) still runs in $O(n)$ time in this case.

C-6.8 The NodeList implementation given in Code Fragments 6.9 through 6.12 does not do any checking to determine whether a given position p is actually a member of this particular list. For example, if p is a position in list S and we call T.insert(p, e) on a different list T, then we actually will add the element to S just before p. Describe how to change the NodeList implementation in an efficient manner to disallow such misuses.

C-6.9 Describe an implementation of the functions insertBack and insertFront realized by using combinations of only the functions empty and insert.

C-6.10 Consider the following fragment of C++ code, assuming that the constructor Sequence creates an empty sequence of integer objects. Recall that division between integers performs truncation (for example, $7/2 = 3$).

```
Sequence<int> seq;
for (int i = 0; i < n; i++)
    seq.insertAtRank(i/2, i);
```

a. Assume that the **for** loop is executed 10 times, that is, $n = 10$, and show the sequence after each iteration of the loop.
b. Draw a schematic illustration of the sequence at the end of the **for** loop, for a generic number n of iterations.

C-6.11 Suppose we want to extend the Sequence abstract data type with functions indexOfElement(e) and positionOfElement(e), which respectively return the index and the position of the (first occurrence of) element e in the

sequence. Show how to implement these functions by expressing them in terms of other functions of the Sequence interface.

C-6.12 Describe the structure and pseudo-code for an array-based implementation of the vector ADT that achieves $O(1)$ time for insertions and removals at index 0, as well as insertions and removals at the end of the vector. Your implementation should also provide for a constant time elemAtRank function.

C-6.13 Describe an efficient way of putting a vector representing a deck of n cards into random order. You may use a function, randomInteger(n), which returns a random number between 0 and $n - 1$, inclusive. Your method should guarantee that every possible ordering is equally likely. What is the running time of your function?

C-6.14 Design a circular node list ADT that abstracts a circularly linked list in the same way that the node list ADT abstracts a doubly linked list.

C-6.15 An array is *sparse* if most of its entries are NULL. A list L can be used to implement such an array, A, efficiently. In particular, for each nonnull cell $A[i]$, we can store an entry (i, e) in L, where e is the element stored at $A[i]$. This approach allows us to represent A using $O(m)$ storage, where m is the number of nonnull entries in A. Describe and analyze efficient ways of performing the functions of the vector ADT on such a representation.

C-6.16 Show that only $n - 1$ passes are needed in the execution of bubble-sort on a sequence with n elements.

C-6.17 Give a pseudo-code description of an implementation of the bubble-sort algorithm that uses only two stacks and, at most, five additional variables to sort a collection of objects stored initially in one of the stacks. You may operate on the stacks using only functions of the stack ADT. The final output should be one of the stacks containing all the elements so that a sequence of pop operations would list the elements in order.

C-6.18 A useful operation in databases is the *natural join*. If we view a database as a list of ordered pairs of objects, then the natural join of databases A and B is the list of all ordered triples (x, y, z) such that the pair (x, y) is in A and the pair (y, z) is in B. Describe and analyze an efficient algorithm for computing the natural join of a list A of n pairs and a list B of m pairs.

C-6.19 When Bob wants to send Alice a message M on the Internet, he breaks M into n *data packets*, numbers the packets consecutively, and injects them into the network. When the packets arrive at Alice's computer, they may be out of order, so Alice must assemble the sequence of n packets in order before she can be sure she has the entire message. Describe an efficient scheme for Alice to do this. What is the running time of this algorithm?

C-6.20 Given a list L of n positive integers, each represented with $k = \lceil \log n \rceil + 1$ bits, describe an $O(n)$-time function for finding a k-bit integer not in L.

C-6.21 Argue why any solution to the previous problem must run in $\Omega(n)$ time.

C-6.22 Given a list L of n arbitrary integers, design an $O(n)$-time function for finding an integer that cannot be formed as the sum of two integers in L.

Projects

P-6.1 Implement the vector ADT by means of an extendable array used in a circular fashion, so that insertions and deletions at the beginning and end of the vector run in constant time.

P-6.2 Implement the vector ADT using a doubly linked list. Show experimentally that this implementation is worse than the array-based approach.

P-6.3 Write a simple text editor, which stores a string of characters using the list ADT, together with a cursor object that highlights the position of some character in the string (or possibly the position before the first character). Your editor should support the following operations and redisplay the current text (that is, the list) after performing any one of them.

- left: Move cursor left one character (or nothing if at the beginning)
- right: Move cursor right one character (or do nothing if at the end)
- delete: Delete the character to the right of the cursor (or do nothing if at the end)
- insert c: Insert the character c just after the cursor

P-6.4 Implement the sequence ADT by means of an extendable array used in a circular fashion, so that insertions and deletions at the beginning and end of the sequence run in constant time.

P-6.5 Implement the sequence ADT by means of a singly linked list.

P-6.6 Write a complete adapter class that implements the sequence ADT using an STL vector object.

Chapter Notes

Sequences and iterators are pervasive concepts in the C++ Standard Template Library (STL) [81], and they play fundamental roles in JDSL, the data structures library in Java. For further information on STL vector and list classes, see books by Stroustrup [91], Lippmann and Lajoie [67], and Musser and Saini [81]. The list ADT was proposed by several authors, including Aho, Hopcroft, and Ullman [5], who introduce the "position" abstraction, and Wood [104], who defines a list ADT similar to ours. Implementations of sequences via arrays and linked lists are discussed in Knuth's seminal book, *Fundamental Algorithms* [56]. Knuth's companion volume, *Sorting and Searching* [57], describes the bubble-sort function and the history of this and other sorting algorithms.

Chapter

7

Trees
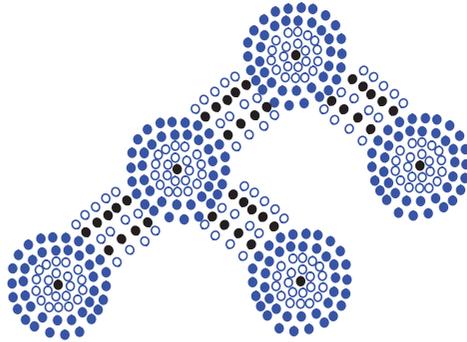
Contents

7.1   General Trees

Productivity experts say that breakthroughs come by thinking "nonlinearly." In this chapter, we discuss one of the most important nonlinear data structures in computing—***trees***. Tree structures are indeed a breakthrough in data organization, for they allow us to implement a host of algorithms much faster than when using linear data structures, such as lists, vectors, and sequences. Trees also provide a natural organization for data, and consequently have become ubiquitous structures in file systems, graphical user interfaces, databases, Web sites, and other computer systems.

It is not always clear what productivity experts mean by "nonlinear" thinking, but when we say that trees are "nonlinear," we are referring to an organizational relationship that is richer than the simple "before" and "after" relationships between objects in sequences. The relationships in a tree are ***hierarchical***, with some objects being "above" and some "below" others. Actually, the main terminology for tree data structures comes from family trees, with the terms "parent," "child," "ancestor," and "descendant" being the most common words used to describe relationships. We show an example of a family tree in Figure 7.1.
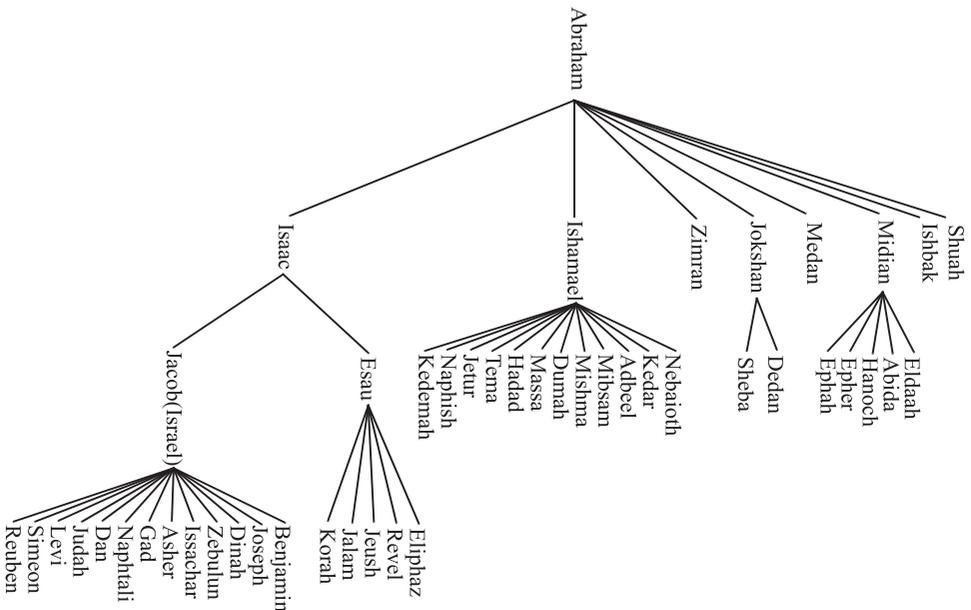
Figure 7.1: A family tree showing some descendants of Abraham, as recorded in Genesis, chapters 25–36.

7.1.1 Tree Definitions and Properties

A *tree* is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a *parent* element and zero or more *children* elements. A tree is usually visualized by placing elements inside ovals or rectangles, and by drawing the connections between parents and children with straight lines. (See Figure 7.2.) We typically call the top element the *root* of the tree, but it is drawn as the highest element, with the other elements being connected below (just the opposite of a botanical tree).
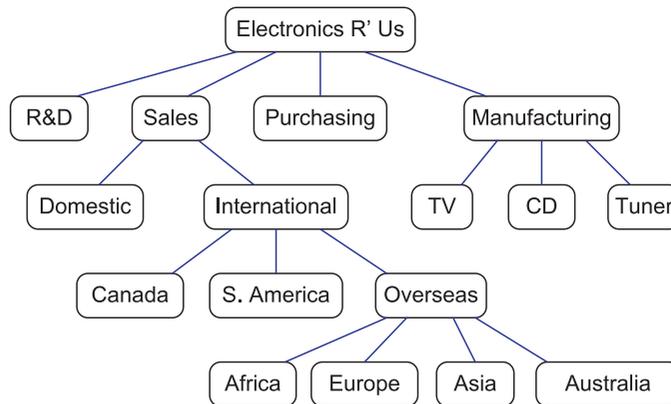
Figure 7.2: A tree with 17 nodes representing the organizational structure of a fictitious corporation. *Electronics R'Us* is stored at the root. The children of the root store *R&D*, *Sales*, *Purchasing*, and *Manufacturing*. The internal nodes store *Sales*, *International*, *Overseas*, *Electronics R'Us*, and *Manufacturing*.

Formal Tree Definition

Formally, we define *tree* T to be a set of *nodes* storing elements in a *parent-child* relationship with the following properties:

- If T is nonempty, it has a special node, called the *root* of T, that has no parent.
- Each node v of T different from the root has a unique *parent* node w; every node with parent w is a *child* of w.

Note that according to our definition, a tree can be empty, meaning that it doesn't have any nodes. This convention also allows us to define a tree recursively, such that a tree T is either empty or consists of a node r, called the root of T, and a (possibly empty) set of trees whose roots are the children of r.

Other Node Relationships

Two nodes that are children of the same parent are **siblings**. A node v is **external** if v has no children. A node v is **internal** if it has one or more children. External nodes are also known as **leaves**.

Example 7.1: *In most operating systems, files are organized hierarchically into nested directories (also called folders), which are presented to the user in the form of a tree. (See Figure 7.3.) More specifically, the internal nodes of the tree are associated with directories and the external nodes are associated with regular files. In the UNIX and Linux operating systems, the root of the tree is appropriately called the "root directory," and is represented by the symbol "/."*
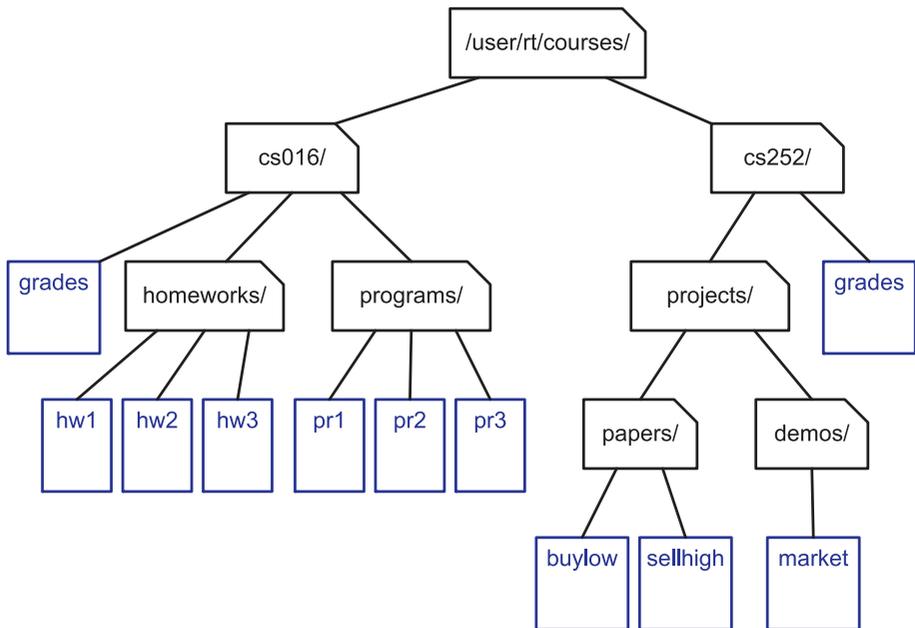
Figure 7.3: Tree representing a portion of a file system.

A node u is an **ancestor** of a node v if $u = v$ or u is an ancestor of the parent of v. Conversely, we say that a node v is a **descendent** of a node u if u is an ancestor of v. For example, in Figure 7.3, cs252/ is an ancestor of papers/, and pr3 is a descendent of cs016/. The **subtree** of T **rooted** at a node v is the tree consisting of all the descendents of v in T (including v itself). In Figure 7.3, the subtree rooted at cs016/ consists of the nodes cs016/, grades, homeworks/, programs/, hw1, hw2, hw3, pr1, pr2, and pr3.

Edges and Paths in Trees

An *edge* of tree T is a pair of nodes (u, v) such that u is the parent of v, or vice versa. A *path* of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge. For example, the tree in Figure 7.3 contains the path (cs252/, projects/, demos/, market).

Example 7.2: *When using single inheritance, the inheritance relation between classes in a C++ program forms a tree. The base class is the root of the tree.*

Ordered Trees

A tree is *ordered* if there is a linear ordering defined for the children of each node; that is, we can identify children of a node as being the first, second, third, and so on. Such an ordering is determined by how the tree is to be used, and is usually indicated by drawing the tree with siblings arranged from left to right, corresponding to their linear relationship. Ordered trees typically indicate the linear order relationship existing between siblings by listing them in a sequence or iterator in the correct order.

Example 7.3: *A structured document, such as a book, is hierarchically organized as a tree whose internal nodes are chapters, sections, and subsections, and whose external nodes are paragraphs, tables, figures, the bibliography, and so on. (See Figure 7.4.) The root of the tree corresponds to the book itself. We could, in fact, consider expanding the tree further to show paragraphs consisting of sentences, sentences consisting of words, and words consisting of characters. In any case, such a tree is an example of an ordered tree, because there is a well-defined ordering among the children of each node.*
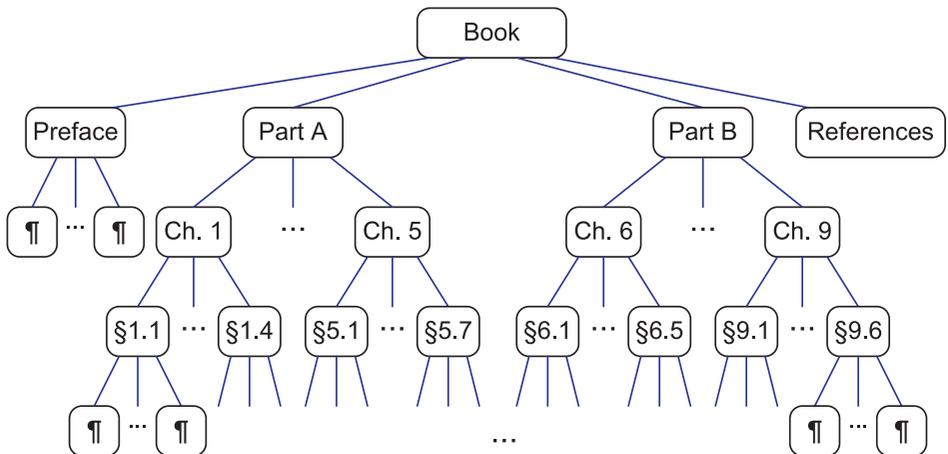
Figure 7.4: An ordered tree associated with a book.

7.1.2 Tree Functions

The tree ADT stores elements at the nodes of the tree. Because nodes are internal aspects of our implementation, we do not allow access to them directly. Instead, each node of the tree is associated with a ***position*** object, which provides public access to nodes. For this reason, when discussing the public interfaces of functions of our ADT, we use the notation p (rather than v) to clarify that the argument to the function is a position and not a node. But, given the tight connection between these two objects, we often blur the distinction between them, and use the terms "position" and "node" interchangeably for trees.

As we did with positions for lists in Chapter 6, we exploit C++'s ability to overload the dereferencing operator ("$*$") to access the element associated with a position. Given a position variable p, the associated element is accessed by $*p$. This can be used both for reading and modifying the element's value.

It is useful to store collections of positions. For example, the children of a node in a tree can be presented to the user as such a list. We define ***position list***, to be a list whose elements are tree positions.

The real power of a tree position arises from its ability to access the neighboring elements of the tree. Given a position p of tree T, we define the following:

p.parent(): Return the parent of p; an error occurs if p is the root.

p.children(): Return a position list containing the children of node p.

p.isRoot(): Return true if p is the root and false otherwise.

p.isExternal(): Return true if p is external and false otherwise.

If a tree T is ordered, then the list provided by p.children() provides access to the children of p in order. If p is an external node, then p.children() returns an empty list. If we wanted, we could also provide a function p.isInternal(), which would simply return the complement of p.isExternal().

The tree itself provides the following functions. The first two, size and empty, are just the standard functions that we defined for the other container types we already saw. The function root yields the position of the root and positions produces a list containing all the tree's nodes.

size(): Return the number of nodes in the tree.

empty(): Return true if the tree is empty and false otherwise.

root(): Return a position for the tree's root; an error occurs if the tree is empty.

positions(): Return a position list of all the nodes of the tree.

We have not defined any specialized update functions for a tree here. Instead, we prefer to describe different tree update functions in conjunction with specific applications of trees in subsequent chapters. In fact, we can imagine several kinds of tree update operations beyond those given in this book.

7.1.3   A C++ Tree Interface

Let us present an informal C++ interface for the tree ADT. We begin by presenting an informal C++ interface for the class Position, which represents a position in a tree. This is given in Code Fragment 7.1.

```
template <typename E>              // base element type
class Position<E> {                // a node position
public:
  E& operator*();                  // get element
  Position parent() const;         // get parent
  PositionList children() const;   // get node's children
  bool isRoot() const;             // root node?
  bool isExternal() const;         // external node?
};
```

Code Fragment 7.1: An informal interface for a position in a tree (not a complete C++ class).

We have provided a version of the derferencing operator ("∗") that returns a standard (readable and writable) reference. (For simplicity, we did not provide a version that returns a constant reference, but this would be an easy addition.)

Next, in Code Fragment 7.2, we present our informal C++ interface for a tree. To keep the interface as simple as possible, we ignore error processing; hence, we do not declare any exceptions to be thrown.

```
template <typename E>              // base element type
class Tree<E> {
public:                            // public types
  class Position;                  // a node position
  class PositionList;              // a list of positions
public:                            // public functions
  int size() const;                // number of nodes
  bool empty() const;              // is tree empty?
  Position root() const;           // get the root
  PositionList positions() const;  // get positions of all nodes
};
```

Code Fragment 7.2: An informal interface for the tree ADT (not a complete class).

Although we have not formally defined an interface for the class PositionList, we may assume that it satisfies the standard list ADT as given in Chapter 6. In our code examples, we assume that PositionList is implemented as an STL list of objects of type Position, or more concretely, "std::list<Position>." In particular, we assume that PositionList provides an iterator type, which we simply call Iterator in our later examples.

7.1.4   A Linked Structure for General Trees

A natural way to realize a tree T is to use a **linked structure**, where we represent each node of T by a position object p (see Figure 7.5(a)) with the following fields: a reference to the node's element, a link to the node's parent, and some kind of collection (for example, a list or array) to store links to the node's children. If p is the root of T, then the *parent* field of p is NULL. Also, we store a reference to the root of T and the number of nodes of T in internal variables. This structure is schematically illustrated in Figure 7.5(b).

(a)   (b)
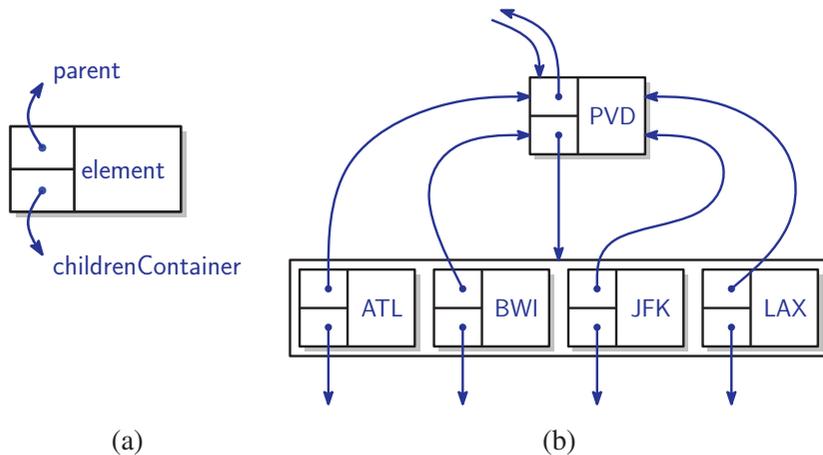
Figure 7.5: The linked structure for a general tree: (a) the node structure; (b) the portion of the data structure associated with a node and its children.

Table 7.1 summarizes the performance of the linked-structure implementation of a tree. The analysis is left as an exercise (C-7.27), but we note that, by using a container to store the children of each node p, we can implement the children(p) function by using the iterator for the container to enumerate its elements.

Operation	Time
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

Table 7.1: Running times of the functions of an n-node linked tree structure. Let c_p denote the number of children of a node p. The space usage is $O(n)$.

7.2 Tree Traversal Algorithms

In this section, we present algorithms for performing traversal computations on a tree by accessing it through the tree ADT functions.

7.2.1 Depth and Height

Let p be a node of a tree T. The **depth** of p is the number of ancestors of p, excluding p itself. For example, in the tree of Figure 7.2, the node storing *International* has depth 2. Note that this definition implies that the depth of the root of T is 0. The depth of p's node can also be recursively defined as follows:

- If p is the root, then the depth of p is 0
- Otherwise, the depth of p is one plus the depth of the parent of p

Based on the above definition, the recursive algorithm depth(T, p) shown in Code Fragment 7.3, computes the depth of a node referenced by position p of T by calling itself recursively on the parent of p, and adding 1 to the value returned.

Algorithm depth(T, p):
  if p.isRoot() **then**
    return 0
  else
    return $1 + $ depth$(T, p$.parent())

Code Fragment 7.3: An algorithm to compute the depth of a node p in a tree T.

A simple C++ implementation of algorithm depth is shown in Code Fragment 7.4.

```
int depth(const Tree& T, const Position& p) {
  if (p.isRoot())
    return 0;                         // root has depth 0
  else
    return 1 + depth(T, p.parent());  // 1 + (depth of parent)
}
```

Code Fragment 7.4: A C++ implementation of the algorithm of Code Fragment 7.3.

The running time of algorithm depth(T, p) is $O(d_p)$, where d_p denotes the depth of the node p in the tree T, because the algorithm performs a constant-time recursive step for each ancestor of p. Thus, in the worst case, the depth algorithm runs in $O(n)$ time, where n is the total number of nodes in the tree T, since some nodes may have this depth in T. Although such a running time is a function of the input size, it is more accurate to characterize the running time in terms of the parameter d_p, since it is often much smaller than n.

The ***height*** of a node p in a tree T is also defined recursively.

- If p is external, then the height of p is 0
- Otherwise, the height of p is one plus the maximum height of a child of p

The ***height*** of a tree T is the height of the root of T. For example, the tree of Figure 7.2 has height 4. In addition, height can also be viewed as follows.

Proposition 7.4: *The height of a tree is equal to the maximum depth of its external nodes.*

We leave the justification of this fact to an exercise (R-7.7). Based on this proposition, we present an algorithm, height1, for computing the height of a tree T. It is shown in Code Fragment 7.5. It enumerates all the nodes in the tree and invokes function depth (Code Fragment 7.3) to compute the depth of each external node.

Algorithm height1(T):
  $h = 0$
  for each $p \in T$.positions() **do**
    if p.isExternal() **then**
        $h = \max(h, \mathrm{depth}(T, p))$
  return h

Code Fragment 7.5: Algorithm height1(T) for computing the height of a tree T based on computing the maximum depth of the external nodes.

The C++ implementation of this algorithm is shown in Code Fragment 7.6. We assume that Iterator is the iterator class for PositionList. Given such an iterator q, we can access the associated position as $*q$.

```
int height1(const Tree& T) {
  int h = 0;
  PositionList nodes = T.positions();         // list of all nodes
  for (Iterator q = nodes.begin(); q != nodes.end(); ++q) {
    if (q->isExternal())
      h = max(h, depth(T, *q));               // get max depth among leaves
  }
  return h;
}
```

Code Fragment 7.6: A C++ implementation of the function height1.

Unfortunately, algorithm height1 is not very efficient. Since height1 calls algorithm depth(p) on each external node p of T, the running time of height1 is given by $O(n + \sum_p (1 + d_p))$, where n is the number of nodes of T, d_p is the depth of node p, and E is the set of external nodes of T. In the worst case, the sum $\sum_p (1 + d_p)$ is proportional to n^2. (See Exercise C-7.8.) Thus, algorithm height1 runs in $O(n^2)$ time.

Algorithm height2, shown in Code Fragment 7.7 and implemented in C++ in Code Fragment 7.8, computes the height of tree T in a more efficient manner by using the recursive definition of height.

Algorithm height2(T, p):

  if p.isExternal() **then**

    return 0

  else

    $h = 0$

    for each $q \in p$.children() **do**

      $h = \max(h, \text{height2}(T, q))$

    return $1 + h$

Code Fragment 7.7: A more efficient algorithm for computing the height of the subtree of tree T rooted at a node p.

```cpp
int height2(const Tree& T, const Position& p) {
    if (p.isExternal()) return 0;            // leaf has height 0
    int h = 0;
    PositionList ch = p.children();          // list of children
    for (Iterator q = ch.begin(); q != ch.end(); ++q)
        h = max(h, height2(T, *q));
    return 1 + h;                            // 1 + max height of children
}
```

Code Fragment 7.8: Method height2 written in C++.

Algorithm height2 is more efficient than height1 (from Code Fragment 7.5). The algorithm is recursive, and, if it is initially called on the root of T, it will eventually be called on each node of T. Thus, we can determine the running time of this method by summing, over all the nodes, the amount of time spent at each node (on the nonrecursive part). Processing each node in children(p) takes $O(c_p)$ time, where c_p denotes the number of children of node p. Also, the **while** loop has c_p iterations and each iteration of the loop takes $O(1)$ time plus the time for the recursive call on a child of p. Thus, algorithm height2 spends $O(1 + c_p)$ time at each node p, and its running time is $O(\sum_p (1 + c_p))$. In order to complete the analysis, we make use of the following property.

Proposition 7.5: *Let T be a tree with n nodes, and let c_p denote the number of children of a node p of T. Then $\sum_p c_p = n - 1$.*

Justification: Each node of T, with the exception of the root, is a child of another node, and thus contributes one unit to the above sum. ∎

By Proposition 7.5, the running time of algorithm height2, when called on the root of T, is $O(n)$, where n is the number of nodes of T.

7.2.2　Preorder Traversal

A *traversal* of a tree T is a systematic way of accessing, or "visiting," all the nodes of T. In this section, we present a basic traversal scheme for trees, called preorder traversal. In the next section, we study another basic traversal scheme, called postorder traversal.

In a *preorder* traversal of a tree T, the root of T is visited first and then the subtrees rooted at its children are traversed recursively. If the tree is ordered, then the subtrees are traversed according to the order of the children. The specific action associated with the "visit" of a node depends on the application of this traversal, and could involve anything from incrementing a counter to performing some complex computation for this node. The pseudo-code for the preorder traversal of the subtree rooted at a node referenced by position p is shown in Code Fragment 7.9. We initially invoke this routine with the call preorder(T, T.root()).

Algorithm preorder(T, p):

　perform the "visit" action for node p
　for each child q of p **do**
　　recursively traverse the subtree rooted at q by calling preorder(T, q)

Code Fragment 7.9: Algorithm preorder for performing the preorder traversal of the subtree of a tree T rooted at a node p.

The preorder traversal algorithm is useful for producing a linear ordering of the nodes of a tree where parents must always come before their children in the ordering. Such orderings have several different applications. We explore a simple instance of such an application in the next example.
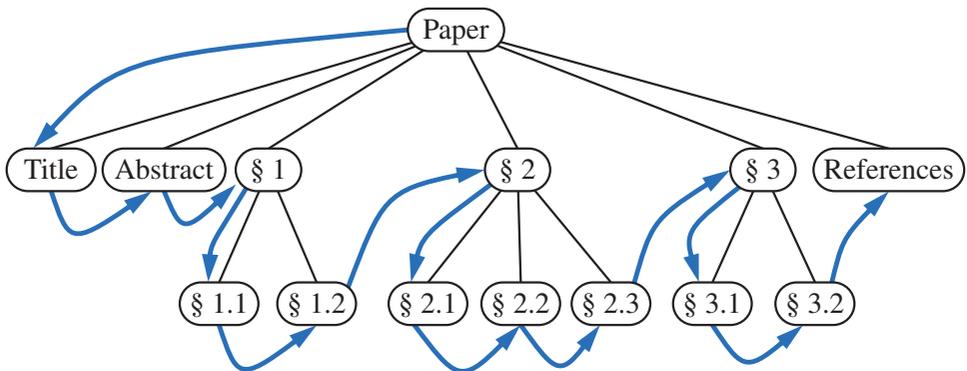
Figure 7.6: Preorder traversal of an ordered tree, where the children of each node are ordered from left to right.

Example 7.6: *The preorder traversal of the tree associated with a document, as in Example 7.3, examines an entire document sequentially, from beginning to end. If the external nodes are removed before the traversal, then the traversal examines the table of contents of the document. (See Figure 7.6.)*

The preorder traversal is also an efficient way to access all the nodes of a tree. To justify this, let us consider the running time of the preorder traversal of a tree T with n nodes under the assumption that visiting a node takes $O(1)$ time. The analysis of the preorder traversal algorithm is actually similar to that of algorithm height2 (Code Fragment 7.8), given in Section 7.2.1. At each node p, the nonrecursive part of the preorder traversal algorithm requires time $O(1 + c_p)$, where c_p is the number of children of p. Thus, by Proposition 7.5, the overall running time of the preorder traversal of T is $O(n)$.

Algorithm preorderPrint(T, p), implemented in C++ in Code Fragment 7.10, performs a preorder printing of the subtree of a node p of T, that is, it performs the preorder traversal of the subtree rooted at p and prints the element stored at a node when the node is visited. Recall that, for an ordered tree T, function T.children(p) returns an iterator that accesses the children of p in order. We assume that Iterator is this iterator type. Given an iterator q, the associated position is given by $*q$.

```cpp
void preorderPrint(const Tree& T, const Position& p) {
  cout << *p;                           // print element
  PositionList ch = p.children();       // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    cout << " ";
    preorderPrint(T, *q);
  }
}
```

Code Fragment 7.10: Method preorderPrint(T, p) that performs a preorder printing of the elements in the subtree associated with position p of T.

There is an interesting variation of the preorderPrint function that outputs a different representation of an entire tree. The ***parenthetic string representation*** $P(T)$ of tree T is recursively defined as follows. If T consists of a single node referenced by a position p, then

$$P(T) = *p.$$

Otherwise,

$$P(T) = *p + \texttt{"("} + P(T_1) + P(T_2) + \cdots + P(T_k) + \texttt{")"},$$

where p is the root position of T and $T_1, T_2, \ldots, T_k$ are the subtrees rooted at the children of p, which are given in order if T is an ordered tree.

Note that the definition of $P(T)$ is recursive. Also, we are using "+" here to denote string concatenation. (Recall the string type from Section 1.1.3.) The parenthetic representation of the tree of Figure 7.2 is shown in Figure 7.7.

> *Electronics R'Us* (
>   *R&D*
>   *Sales* (
>   *Domestic*
>   *International* (
>   *Canada*
>   *S. America*
>   *Overseas* (*Africa*   *Europe*   *Asia*   *Australia*)))
>   *Purchasing*
>   *Manufacturing* (*TV*   *CD*   *Tuner*))

Figure 7.7: Parenthetic representation of the tree of Figure 7.2. Indentation, line breaks, and spaces have been added for clarity.

Note that, technically speaking, there are some computations that occur between and after the recursive calls at a node's children in the above algorithm. We still consider this algorithm to be a preorder traversal, however, since the primary action of printing a node's contents occurs prior to the recursive calls.

The C++ function parenPrint, shown in Code Fragment 7.11, is a variation of function preorderPrint (Code Fragment 7.10). It implements the definition given above to output a parenthetic string representation of a tree T. It first prints the element associated with each node. For each internal node, we first print "(", followed by the parenthetical representation of each of its children, followed by ")".

```cpp
void parenPrint(const Tree& T, const Position& p) {
  cout << *p;                              // print node's element
  if (!p.isExternal()) {
    PositionList ch = p.children();        // list of children
    cout << "( ";                          // open
    for (Iterator q = ch.begin(); q != ch.end(); ++q) {
      if (q != ch.begin()) cout << " ";    // print separator
      parenPrint(T, *q);                   // visit the next child
    }
    cout << " )";                          // close
  }
}
```

Code Fragment 7.11: A C++ implementation of algorithm parenPrint.

We explore a modification of Code Fragment 7.11 in Exercise R-7.10, to display a tree in a fashion more closely matching that given in Figure 7.7.

7.2.3 Postorder Traversal

Another important tree traversal algorithm is the ***postorder traversal***. This algorithm can be viewed as the opposite of the preorder traversal, because it recursively traverses the subtrees rooted at the children of the root first, and then visits the root. It is similar to the preorder traversal, however, in that we use it to solve a particular problem by specializing an action associated with the "visit" of a node p. Still, as with the preorder traversal, if the tree is ordered, we make recursive calls for the children of a node p according to their specified order. Pseudo-code for the postorder traversal is given in Code Fragment 7.12.

Algorithm postorder(T, p):
    for each child q of p **do**
      recursively traverse the subtree rooted at q by calling postorder(T, q)
    perform the "visit" action for node p

Code Fragment 7.12: Algorithm postorder for performing the postorder traversal of the subtree of a tree T rooted at a node p.

The name of the postorder traversal comes from the fact that this traversal method visits a node p after it has visited all the other nodes in the subtree rooted at p. (See Figure 7.8.)
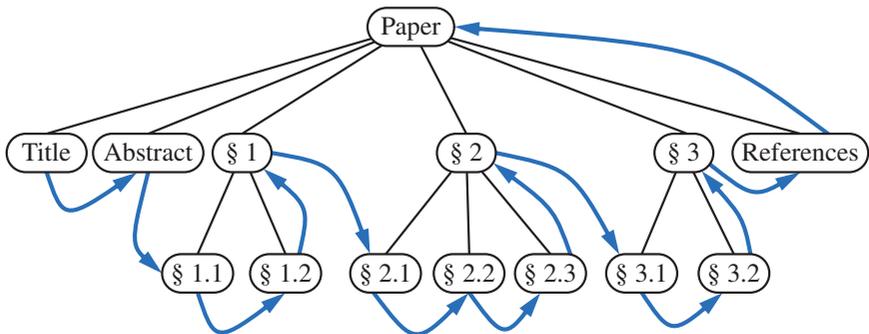
Figure 7.8: Postorder traversal of the ordered tree of Figure 7.6.

The analysis of the running time of a postorder traversal is analogous to that of a preorder traversal. (See Section 7.2.2.) The total time spent in the nonrecursive portions of the algorithm is proportional to the time spent visiting the children of each node in the tree. Thus, a postorder traversal of a tree T with n nodes takes $O(n)$ time, assuming that visiting each node takes $O(1)$ time. That is, the postorder traversal runs in linear time.

In Code Fragment 7.13, we present a C++ function postorderPrint which per-

forms a postorder traversal of a tree T. This function prints the element stored at a node when it is visited.

```cpp
void postorderPrint(const Tree& T, const Position& p) {
  PositionList ch = p.children();              // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    postorderPrint(T, *q);
    cout << " ";
  }
  cout << *p;                                  // print element
}
```

Code Fragment 7.13: The function postorderPrint(T, p), which prints the elements of the subtree of position p of T.

The postorder traversal method is useful for solving problems where we wish to compute some property for each node p in a tree, but computing that property for p requires that we have already computed that same property for p's children. Such an application is illustrated in the following example.

Example 7.7: *Consider a file-system tree T, where external nodes represent files and internal nodes represent directories (Example 7.1). Suppose we want to compute the disk space used by a directory, which is recursively given by the sum of the following (see Figure 7.9):*

- *The size of the directory itself*
- *The sizes of the files in the directory*
- *The space used by the children directories*
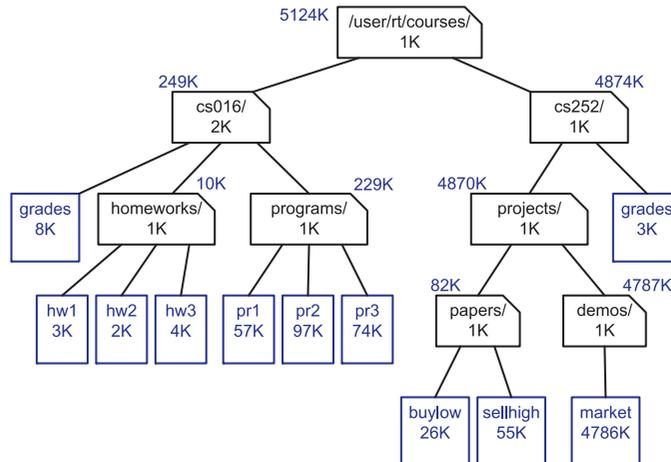
Figure 7.9: The tree of Figure 7.3 representing a file system, showing the name and size of the associated file/directory inside each node, and the disk space used by the associated directory above each internal node.

This computation can be done with a postorder traversal of tree T. After the subtrees of an internal node p have been traversed, we compute the space used by p by adding the sizes of the directory p itself and of the files contained in p, to the space used by each internal child of p, which was computed by the recursive postorder traversals of the children of p.

Motivated by Example 7.7, algorithm diskSpace, which is presented in Code Fragment 7.14, performs a postorder traversal of a file-system tree T, printing the name and disk space used by the directory associated with each internal node of T. When called on the root of tree T, diskSpace runs in time $O(n)$, where n is the number of nodes of the tree, provided the auxiliary functions name(p) and size(p) take $O(1)$ time.

```
int diskSpace(const Tree& T, const Position& p) {
  int s = size(p);                         // start with size of p
  if (!p.isExternal()) {                   // if p is internal
    PositionList ch = p.children();        // list of p's children
    for (Iterator q = ch.begin(); q != ch.end(); ++q)
      s += diskSpace(T, *q);               // sum the space of subtrees
    cout << name(p) << ": " << s << endl;  // print summary
  }
  return s;
}
```

Code Fragment 7.14: The function diskSpace, which prints the name and disk space used by the directory associated with p, for each internal node p of a file-system tree T. This function calls the auxiliary functions name and size, which should be defined to return the name and size of the file/directory associated with a node.

Other Kinds of Traversals

Preorder traversal is useful when we want to perform an action for a node and then recursively perform that action for its children, and postorder traversal is useful when we want to first perform an action on the descendents of a node and then perform that action on the node.

Although the preorder and postorder traversals are common ways of visiting the nodes of a tree, we can also imagine other traversals. For example, we could traverse a tree so that we visit all the nodes at depth d before we visit the nodes at depth $d+1$. Such a traversal, called a **breadth-first traversal**, could be implemented using a queue, whereas the preorder and postorder traversals use a stack. (This stack is implicit in our use of recursion to describe these functions, but we could make this use explicit, as well, to avoid recursion.) In addition, binary trees, which we discuss next, support an additional traversal method known as the inorder traversal.

7.3 Binary Trees

A *binary tree* is an ordered tree in which every node has at most two children.

1. Every node has at most two children.
2. Each child node is labeled as being either a *left child* or a *right child*.
3. A left child precedes a right child in the ordering of children of a node.

The subtree rooted at a left or right child of an internal node is called the node's *left subtree* or *right subtree*, respectively. A binary tree is *proper* if each node has either zero or two children. Some people also refer to such trees as being *full* binary trees. Thus, in a proper binary tree, every internal node has exactly two children. A binary tree that is not proper is *improper*.

Example 7.8: *An important class of binary trees arises in contexts where we wish to represent a number of different outcomes that can result from answering a series of yes-or-no questions. Each internal node is associated with a question. Starting at the root, we go to the left or right child of the current node, depending on whether the answer to the question is "Yes" or "No." With each decision, we follow an edge from a parent to a child, eventually tracing a path in the tree from the root to an external node. Such binary trees are known as* **decision trees**, *because each external node p in such a tree represents a decision of what to do if the questions associated with p's ancestors are answered in a way that leads to p. A decision tree is a proper binary tree. Figure 7.10 illustrates a decision tree that provides recommendations to a prospective investor.*
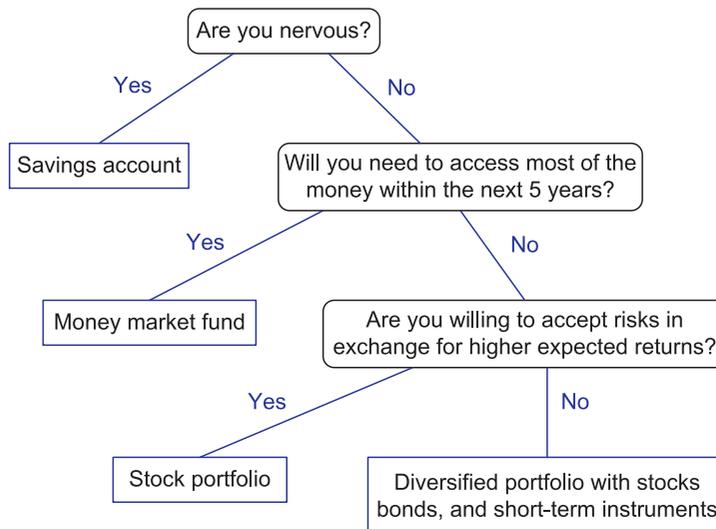
Figure 7.10: A decision tree providing investment advice.

Example 7.9: *An arithmetic expression can be represented by a tree whose external nodes are associated with variables or constants, and whose internal nodes are associated with one of the operators* $+$, $-$, $\times$, *and* $/$. *(See Figure 7.11.) Each node in such a tree has a value associated with it.*

- *If a node is external, then its value is that of its variable or constant.*
- *If a node is internal, then its value is defined by applying its operation to the values of its children.*

Such an arithmetic-expression tree is a proper binary tree, since each of the operators $+$, $-$, $\times$, and $/$ take exactly two operands. Of course, if we were to allow for unary operators, like negation ($-$), as in "$-x$," then we could have an improper binary tree.
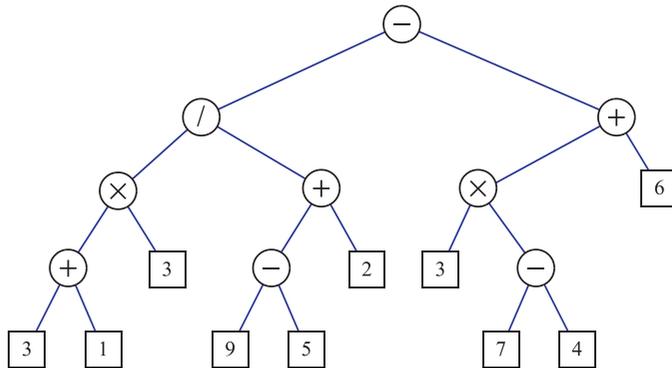
Figure 7.11: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$. The value associated with the internal node labeled "$/$" is 2.

A Recursive Binary Tree Definition

Incidentally, we can also define a binary tree in a recursive way such that a binary tree is either empty or consists of:

- A node r, called the **root** of T and storing an element
- A binary tree, called the **left subtree** of T
- A binary tree, called the **right subtree** of T

We discuss some of the specialized topics for binary trees below.

7.3.1   The Binary Tree ADT

In this section, we introduce an abstract data type for a binary tree. As with our earlier tree ADT, each node of the tree stores an element and is associated with a

position object, which provides public access to nodes. By overloading the dereferencing operator, the element associated with a position p can be accessed by $*p$. In addition, a position p supports the following operations.

p.left(): Return the left child of p; an error condition occurs if p is an external node.

p.right(): Return the right child of p; an error condition occurs if p is an external node.

p.parent(): Return the parent of p; an error occurs if p is the root.

p.isRoot(): Return true if p is the root and false otherwise.

p.isExternal(): Return true if p is external and false otherwise.

The tree itself provides the same operations as the standard tree ADT. Recall that a position list is a list of tree positions.

size(): Return the number of nodes in the tree.

empty(): Return true if the tree is empty and false otherwise.

root(): Return a position for the tree's root; an error occurs if the tree is empty.

positions(): Return a position list of all the nodes of the tree.

As in Section 7.1.2 for the tree ADT, we do not define specialized update functions for binary trees, but we consider them later.

7.3.2 A C++ Binary Tree Interface

Let us present an informal C++ interface for the binary tree ADT. We begin in Code Fragment 7.15 by presenting an informal C++ interface for the class Position, which represents a position in a tree. It differs from the tree interface of Section 7.1.3 by replacing the tree member function children with the two functions left and right.

```
template <typename E>                          // base element type
class Position<E> {                            // a node position
public:
  E& operator*();                              // get element
  Position left() const;                       // get left child
  Position right() const;                      // get right child
  Position parent() const;                     // get parent
  bool isRoot() const;                         // root of tree?
  bool isExternal() const;                     // an external node?
};
```

Code Fragment 7.15: An informal interface for the binary tree ADT (not a complete C++ class).

Next, in Code Fragment 7.16, we present an informal C++ interface for a binary tree. To keep the interface as simple as possible, we have ignored error processing, and hence we do not declare any exceptions to be thrown.

```
template <typename E>                      // base element type
class BinaryTree<E> {                       // binary tree
public:                                     // public types
  class Position;                           // a node position
  class PositionList;                       // a list of positions
public:                                     // member functions
  int size() const;                         // number of nodes
  bool empty() const;                       // is tree empty?
  Position root() const;                    // get the root
  PositionList positions() const;           // list of nodes
};
```

Code Fragment 7.16: An informal interface for the binary tree ADT (not a complete C++ class).

Although we have not formally defined an interface for the class PositionList, we may assume that it satisfies the standard list ADT as given in Chapter 6. In our code examples, we assume that PositionList is implemented as an STL list of objects of type Position.

7.3.3 Properties of Binary Trees

Binary trees have several interesting properties dealing with relationships between their heights and number of nodes. We denote the set of all nodes of a tree T, at the same depth d, as the **level** d of T. In a binary tree, level 0 has one node (the root), level 1 has, at most, two nodes (the children of the root), level 2 has, at most, four nodes, and so on. (See Figure 7.12.) In general, level d has, at most, 2^d nodes.

We can see that the maximum number of nodes on the levels of a binary tree grows exponentially as we go down the tree. From this simple observation, we can derive the following properties relating the height of a binary T to its number of nodes. A detailed justification of these properties is left as an exercise (R-7.16).

Proposition 7.10: *Let T be a nonempty binary tree, and let n, n_E, n_I and h denote the number of nodes, number of external nodes, number of internal nodes, and height of T, respectively. Then T has the following properties:*

1. $h+1 \le n \le 2^{h+1}-1$
2. $1 \le n_E \le 2^h$
3. $h \le n_I \le 2^h - 1$
4. $\log(n+1)-1 \le h \le n-1$

Also, if T is proper, then it has the following properties:

1. $2h+1 \le n \le 2^{h+1}-1$
2. $h+1 \le n_E \le 2^h$
3. $h \le n_I \le 2^h-1$
4. $\log(n+1)-1 \le h \le (n-1)/2$
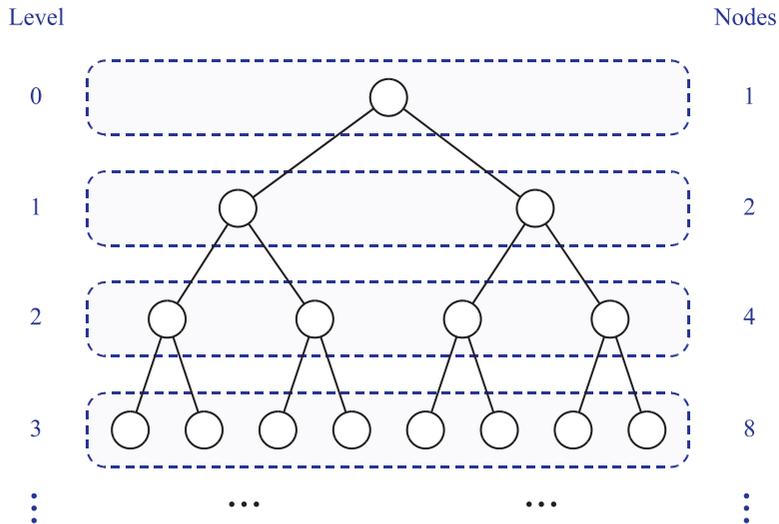
Level   Nodes

Figure 7.12: Maximum number of nodes in the levels of a binary tree.

In addition to the binary tree properties above, we also have the following relationship between the number of internal nodes and external nodes in a proper binary tree.

Proposition 7.11: *In a nonempty proper binary tree T, the number of external nodes is one more than the number of internal nodes.*

Justification: We can see this using an argument based on induction. If the tree consists of a single root node, then clearly we have one external node and no internal nodes, so the proposition holds.

If, on the other hand, we have two or more, then the root has two subtrees. Since the subtrees are smaller than the original tree, we may assume that they satisfy the proposition. Thus, each subtree has one more external node than internal nodes. Between the two of them, there are two more external nodes than internal nodes. But, the root of the tree is an internal node. When we consider the root and both subtrees together, the difference between the number of external and internal nodes is $2-1=1$, which is just what we want.   ∎

Note that the above relationship does not hold, in general, for improper binary trees and nonbinary trees, although there are other interesting relationships that can hold as we explore in an exercise (C-7.9).

7.3.4 A Linked Structure for Binary Trees

In this section, we present an implementation of a binary tree T as a ***linked structure***, called LinkedBinaryTree. We represent each node v of T by a node object storing the associated element and pointers to its parent and two children. (See Figure 7.13.) For simplicity, we assume the tree is ***proper***, meaning that each node has either zero or two children.
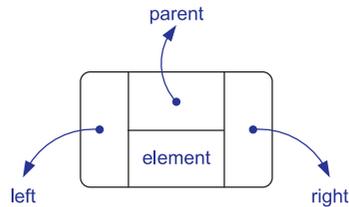
Figure 7.13: A node in a linked data structure for representing a binary tree.

In Figure 7.14, we show a linked structure representation of a binary tree. The structure stores the tree's size, that is, the number of nodes in the tree, and a pointer to the root of the tree. The rest of the structure consists of the nodes linked together appropriately. If v is the root of T, then the pointer to the parent node is NULL, and if v is an external node, then the pointers to the children of v are NULL.
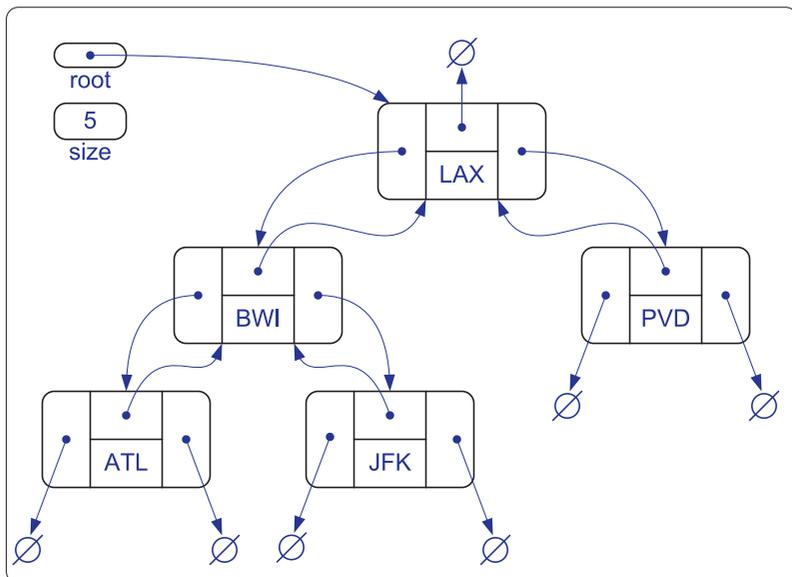
Figure 7.14: An example of a linked data structure for representing a binary tree.

We begin by defining the basic constituents that make up the LinkedBinaryTree class. The most basic entity is the structure Node, shown in Code Fragment 7.17, that represents a node of the tree.

```
struct Node {                                    // a node of the tree
   Elem    elt;                                  // element value
   Node*  par;                                   // parent
   Node*  left;                                  // left child
   Node*  right;                                 // right child
   Node() : elt(), par(NULL), left(NULL), right(NULL) { } // constructor
};
```

Code Fragment 7.17: Structure Node implementing a node of a binary tree. It is nested in the protected section of class BinaryTree.

Although all its members are public, class Node is declared within the protected section of the LinkedBinaryTree class. Thus, it is not publicly accessible. Each node has a member variable *elt*, which contains the associated element, and pointers *par*, *left*, and *right*, which point to the associated relatives.

Next, we define the public class Position in Code Fragment 7.18. Its data member consists of a pointer *v* to a node of the tree. Access to the node's element is provided by overloading the dereferencing operator ("*"). We declare Linked-BinaryTree to be a friend, providing it access to the private data.

```
class Position {                                 // position in the tree
private:
   Node* v;                                      // pointer to the node
public:
   Position(Node* _v = NULL) : v(_v) { }         // constructor
   Elem& operator*()                             // get element
      { return v−>elt; }
   Position left() const                         // get left child
      { return Position(v−>left); }
   Position right() const                        // get right child
      { return Position(v−>right); }
   Position parent() const                       // get parent
      { return Position(v−>par); }
   bool isRoot() const                           // root of the tree?
      { return v−>par == NULL; }
   bool isExternal() const                       // an external node?
      { return v−>left == NULL && v−>right == NULL; }
   friend class LinkedBinaryTree;                // give tree access
};
typedef std::list<Position> PositionList;        // list of positions
```

Code Fragment 7.18: Class Position implementing a position in a binary tree. It is nested in the public section of class LinkedBinaryTree.

Most of the functions of class Position simply involve accessing the appropriate members of the Node structure. We have also included a declaration of the class PositionList, as an STL list of positions. This is used to represent collections of nodes. To keep the code simple, we have omitted error checking, and, rather than using templates, we simply provide a type definition for the base element type, called Elem. (See Exercise P-7.2.)

We present the major part of the class LinkedBinaryTree in Code Fragment 7.19. The class declaration begins by inserting the above declarations of Node and Position. This is followed by a declaration of the public members, local utility functions, and the private member data. We have omitted housekeeping functions, such as a destructor, assignment operator, and copy constructor.

```cpp
typedef int Elem;                               // base element type
class LinkedBinaryTree {
protected:
  // insert Node declaration here...
public:
  // insert Position declaration here...
public:
  LinkedBinaryTree();                           // constructor
  int size() const;                             // number of nodes
  bool empty() const;                           // is tree empty?
  Position root() const;                        // get the root
  PositionList positions() const;               // list of nodes
  void addRoot();                               // add root to empty tree
  void expandExternal(const Position& p);       // expand external node
  Position removeAboveExternal(const Position& p); // remove p and parent
  // housekeeping functions omitted...
protected:                                      // local utilities
  void preorder(Node* v, PositionList& pl) const; // preorder utility
private:
  Node* _root;                                  // pointer to the root
  int n;                                        // number of nodes
};
```

Code Fragment 7.19: Implementation of a LinkedBinaryTree class.

The private data for class LinkedBinaryTree consists of a pointer _root to the root node and a variable n, containing the number of nodes in the tree. (We added the underscore to the name *root* to avoid a name conflict with the member function root.) In addition to the functions of the ADT, we have introduced a few update functions, addRoot, expandExternal, and removeAboveExternal, which provide the means to build and modify trees. They are discussed below. We define a utility function preorder, which is used in the implementation of the function positions.

In Code Fragment 7.20, we present the definitions of the constructor and sim-

pler member functions of class LinkedBinaryTree. The function addRoot assumes that the tree is empty, and it creates a single root node. (It should not be invoked if the tree is nonempty, since otherwise a memory leak results.)

```
LinkedBinaryTree::LinkedBinaryTree()                    // constructor
  : _root(NULL), n(0) { }
int LinkedBinaryTree::size() const                      // number of nodes
  { return n; }
bool LinkedBinaryTree::empty() const                    // is tree empty?
  { return size() == 0; }
LinkedBinaryTree::Position LinkedBinaryTree::root() const // get the root
  { return Position(_root); }
void LinkedBinaryTree::addRoot()                        // add root to empty tree
  { _root = new Node; n = 1; }
```

Code Fragment 7.20: Simple member functions for class LinkedBinaryTree.

Binary Tree Update Functions

In addition to the BinaryTree interface functions and addRoot, the class LinkedBinaryTree also includes the following update functions given a position p. The first is used for adding nodes to the tree and the second is used for removing nodes.

expandExternal(p): Transform p from an external node into an internal node by creating two new external nodes and making them the left and right children of p, respectively; an error condition occurs if p is an internal node.

removeAboveExternal(p): Remove the external node p together with its parent q, replacing q with the sibling of p (see Figure 7.15, where p's node is w and q's node is v); an error condition occurs if p is an internal node or p is the root.
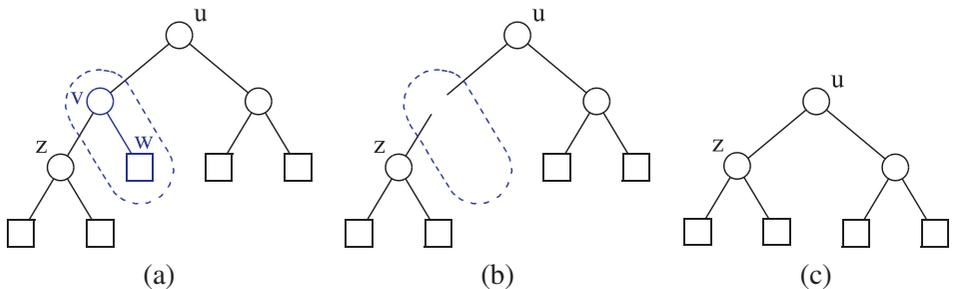
Figure 7.15: Operation removeAboveExternal(p), which removes the external node w to which p refers and its parent node v.

The function expandExternal(p) is shown in Code Fragment 7.21. Letting v be p's associated node, it creates two new nodes. One becomes v's left child and the other becomes v's right child. The constructor for Node initializes the node's pointers to NULL, so we need only update the new node's parent links.

```
                                                    // expand external node
  void LinkedBinaryTree::expandExternal(const Position& p) {
    Node* v = p.v;                                  // p's node
    v->left = new Node;                             // add a new left child
    v->left->par = v;                              // v is its parent
    v->right = new Node;                           // and a new right child
    v->right->par = v;                             // v is its parent
    n += 2;                                         // two more nodes
  }
```

Code Fragment 7.21: The function expandExternal(p) of class LinkedBinaryTree.

The function removeAboveExternal(p) is shown in Code Fragment 7.22. Let w be p's associated node and let v be its parent. We assume that w is external and is not the root. There are two cases. If w is a child of the root, removing w and its parent (the root) causes w's sibling to become the tree's new root. If not, we replace w's parent with w's sibling. This involves finding w's grandparent and determining whether v is the grandparent's left or right child. Depending on which, we set the link for the appropriate child of the grandparent. After unlinking w and v, we delete these nodes. Finally, we update the number of nodes in the tree.

```
  LinkedBinaryTree::Position                        // remove p and parent
  LinkedBinaryTree::removeAboveExternal(const Position& p) {
    Node* w = p.v; Node* v = w->par;               // get p's node and parent
    Node* sib = (w == v->left ? v->right : v->left);
    if (v == _root) {                              // child of root?
      _root = sib;                                 // ...make sibling root
      sib->par = NULL;
    }
    else {
      Node* gpar = v->par;                         // w's grandparent
      if (v == gpar->left) gpar->left = sib;       // replace parent by sib
      else gpar->right = sib;
      sib->par = gpar;
    }
    delete w; delete v;                            // delete removed nodes
    n -= 2;                                         // two fewer nodes
    return Position(sib);
  }
```

Code Fragment 7.22: An implementation of the function removeAboveExternal(p).

The function positions is shown in Code Fragment 7.23. It invokes the utility function preorder, which traverses the tree and stores the node positions in an STL vector.

```
                                                        // list of all nodes
LinkedBinaryTree::PositionList LinkedBinaryTree::positions() const {
  PositionList pl;
  preorder(_root, pl);                                  // preorder traversal
  return PositionList(pl);                              // return resulting list
}
                                                        // preorder traversal
void LinkedBinaryTree::preorder(Node* v, PositionList& pl) const {
  pl.push_back(Position(v));                            // add this node
  if (v−>left != NULL)                                  // traverse left subtree
    preorder(v−>left, pl);
  if (v−>right != NULL)                                 // traverse right subtree
    preorder(v−>right, pl);
}
```

Code Fragment 7.23: An implementation of the function positions.

We have omitted the housekeeping functions (the destructor, copy constructor, and assignment operator). We leave these as exercises (Exercise C-7.22), but they also involve performing a traversal of the tree.

Performance of the LinkedBinaryTree Implementation

Let us now analyze the running times of the functions of class LinkedBinaryTree, which uses a linked structure representation.

- Each of the position functions left, right, parent, isRoot, and isExternal takes $O(1)$ time.
- By accessing the member variable n, which stores the number of nodes of T, functions size and empty each run in $O(1)$ time.
- The accessor function root runs in $O(1)$ time.
- The update functions expandExternal and removeAboveExternal visit only a constant number of nodes, so they both run in $O(1)$ time.
- Function positions is implemented by performing a preorder traversal, which takes $O(n)$ time. (We discuss three different binary-tree traversals in Section 7.3.6. Any of these suffice.) The nodes visited by the traversal are each added in $O(1)$ time to an STL list. Thus, function positions takes $O(n)$ time.

Table 7.2 summarizes the performance of this implementation of a binary tree. There is an object of class Node (Code Fragment 7.17) for each node of tree T. Thus, the overall space requirement is $O(n)$.

	Operation	Time
	left, right, parent, isExternal, isRoot	$O(1)$
	size, empty	$O(1)$
	root	$O(1)$
	expandExternal, removeAboveExternal	$O(1)$
	positions	$O(n)$

Table 7.2: Running times for the functions of an n-node binary tree implemented with a linked structure. The space usage is $O(n)$.

7.3.5 A Vector-Based Structure for Binary Trees

A simple structure for representing a binary tree T is based on a way of numbering the nodes of T. For every node v of T, let $f(v)$ be the integer defined as follows:

- If v is the root of T, then $f(v) = 1$
- If v is the left child of node u, then $f(v) = 2f(u)$
- If v is the right child of node u, then $f(v) = 2f(u) + 1$

The numbering function f is known as a *level numbering* of the nodes in a binary tree T, because it numbers the nodes on each level of T in increasing order from left to right, although it may skip some numbers. (See Figure 7.16.)

The level numbering function f suggests a representation of a binary tree T by means of a vector S, such that node v of T is associated with the element of S at rank $f(v)$. (See Figure 7.17.) Typically, we realize the vector S by means of an extendable array. (See Section 6.1.3.) Such an implementation is simple and efficient, for we can use it to easily perform the functions root, parent, left, right, sibling, isExternal, and isRoot by using simple arithmetic operations on the numbers $f(v)$ associated with each node v involved in the operation. That is, each position object v is simply a "wrapper" for the index $f(v)$ into the vector S. We leave the details of such implementations as a simple exercise (R-7.26).

Let n be the number of nodes of T, and let f_M be the maximum value of $f(v)$ over all the nodes of T. The vector S has size $N = f_M + 1$, since the element of S at index 0 is not associated with any node of T. Also, S will have, in general, a number of empty elements that do not refer to existing nodes of T. For a tree of height h, $N = O(2^h)$. In the worst case, this can be as high as $2^n - 1$. The justification is left as an exercise (R-7.24). In Section 8.3, we discuss a class of binary trees called "heaps," for which $N = n + 1$. Thus, in spite of the worst-case space usage, there are applications for which the array-list representation of a binary tree is space efficient. Still, for general binary trees, the exponential worst-case space requirement of this representation is prohibitive.

Table 7.3 summarizes the running times of the functions of a binary tree imple-
mented with a vector. We do not include any tree update functions here. The vector
implementation of a binary tree is a fast and easy way of realizing the binary-tree
ADT, but it can be very space inefficient if the height of the tree is large.

(a)

(b)
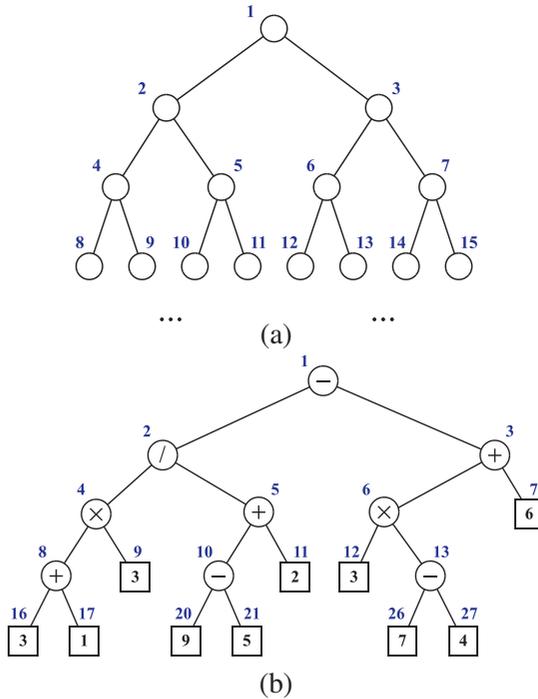
Figure 7.16: Binary tree level numbering: (a) general scheme; (b) an example.
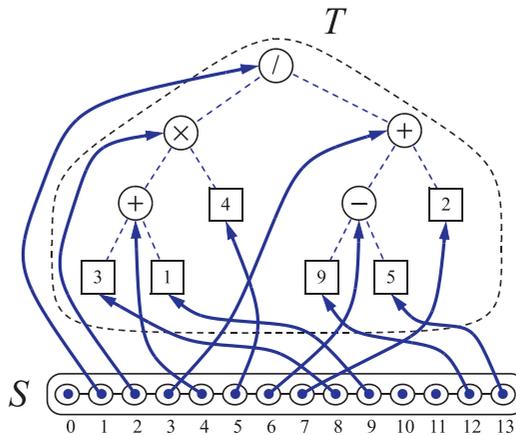
Figure 7.17: Representation of a binary tree *T* by means of a vector *S*.

Operation	Time
left, right, parent, isExternal, isRoot	$O(1)$
size, empty	$O(1)$
root	$O(1)$
expandExternal, removeAboveExternal	$O(1)$
positions	$O(n)$

Table 7.3: Running times for a binary tree T implemented with a vector S. We denote the number of nodes of T with n, and N denotes the size of S. The space usage is $O(N)$, which is $O(2^n)$ in the worst case.

7.3.6 Traversals of a Binary Tree

As with general trees, binary-tree computations often involve traversals.

Preorder Traversal of a Binary Tree

Since any binary tree can also be viewed as a general tree, the preorder traversal for general trees (Code Fragment 7.9) can be applied to any binary tree. We can simplify the algorithm in the case of a binary-tree traversal, however, as we show in Code Fragment 7.24. (Also see Code Fragment 7.23.)

Algorithm binaryPreorder(T, p):
   perform the "visit" action for node p
   if p is an internal node **then**
      binaryPreorder(T, p.left())   {recursively traverse left subtree}
      binaryPreorder(T, p.right())   {recursively traverse right subtree}

Code Fragment 7.24: Algorithm binaryPreorder, which performs the preorder traversal of the subtree of a binary tree T rooted at node p.

For example, a preorder traversal of the binary tree shown in Figure 7.14 visits the nodes in the order $\langle \text{LAX}, \text{BWI}, \text{ATL}, \text{JFK}, \text{PVD} \rangle$. As is the case for general trees, there are many applications of the preorder traversal for binary trees.

Postorder Traversal of a Binary Tree

Analogously, the postorder traversal for general trees (Code Fragment 7.12) can be specialized for binary trees as shown in Code Fragment 7.25.

A postorder traversal of the binary tree shown in Figure 7.14 visits the nodes in the order $\langle \text{ATL}, \text{JFK}, \text{BWI}, \text{PVD}, \text{LAX} \rangle$.

Algorithm binaryPostorder(T, p):
    if p is an internal node **then**
        binaryPostorder(T, p.left())   {recursively traverse left subtree}
        binaryPostorder(T, p.right())   {recursively traverse right subtree}
    perform the "visit" action for the node p

Code Fragment 7.25: Algorithm binaryPostorder for performing the postorder traversal of the subtree of a binary tree T rooted at node p.

Evaluating an Arithmetic Expression

The postorder traversal of a binary tree can be used to solve the expression evaluation problem. In this problem, we are given an arithmetic-expression tree, that is, a binary tree where each external node has a value associated with it and each internal node has an arithmetic operation associated with it (see Example 7.9), and we want to compute the value of the arithmetic expression represented by the tree.

    Algorithm evaluateExpression, given in Code Fragment 7.26, evaluates the expression associated with the subtree rooted at a node p of an arithmetic-expression tree T by performing a postorder traversal of T starting at p. In this case, the "visit" action consists of performing a single arithmetic operation.

Algorithm evaluateExpression(T, p):
    if p is an internal node **then**
        $x \leftarrow$ evaluateExpression(T, p.left())
        $y \leftarrow$ evaluateExpression(T, p.right())
        Let $\circ$ be the operator associated with p
        return $x \circ y$
    else
        return the value stored at p

Code Fragment 7.26: Algorithm evaluateExpression for evaluating the expression represented by the subtree of an arithmetic-expression tree T rooted at node p.

    The expression-tree evaluation application of the postorder traversal provides an $O(n)$-time algorithm for evaluating an arithmetic expression represented by a binary tree with n nodes. Indeed, like the general postorder traversal, the postorder traversal for binary trees can be applied to other "bottom-up" evaluation problems (such as the size computation given in Example 7.7) as well. The specialization of the postorder traversal for binary trees simplifies that for general trees, however, because we use the left and right functions to avoid a loop that iterates through the children of an internal node.

    Interestingly, the specialization of the general preorder and postorder traversal

methods to binary trees suggests a third traversal in a binary tree that is different from both the preorder and postorder traversals. We explore this third kind of traversal for binary trees in the next subsection.

Inorder Traversal of a Binary Tree

An additional traversal method for a binary tree is the ***inorder*** traversal. In this traversal, we visit a node between the recursive traversals of its left and right subtrees. The inorder traversal of the subtree rooted at a node p in a binary tree T is given in Code Fragment 7.27.

Algorithm inorder(T, p):
    if p is an internal node **then**
        inorder(T, p.left())   {recursively traverse left subtree}
    perform the "visit" action for node p
    if p is an internal node **then**
        inorder(T, p.right())   {recursively traverse right subtree}

Code Fragment 7.27: Algorithm inorder for performing the inorder traversal of the subtree of a binary tree T rooted at a node p.

For example, an inorder traversal of the binary tree shown in Figure 7.14 visits the nodes in the order $\langle \text{ATL}, \text{BWI}, \text{JFK}, \text{LAX}, \text{PVD} \rangle$. The inorder traversal of a binary tree T can be informally viewed as visiting the nodes of T "from left to right." Indeed, for every node p, the inorder traversal visits p after all the nodes in the left subtree of p and before all the nodes in the right subtree of p. (See Figure 7.18.)
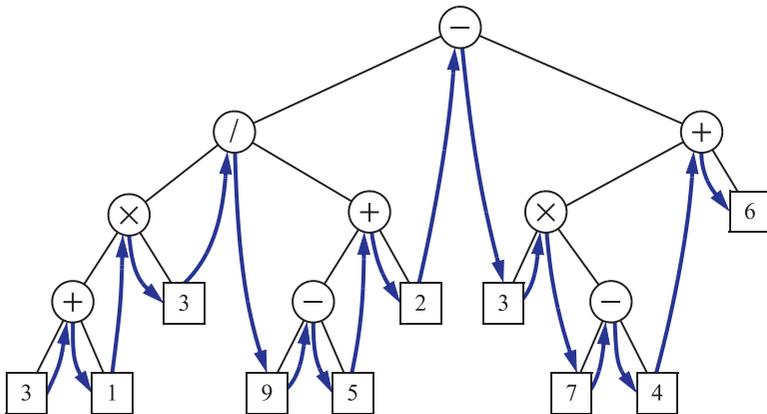
Figure 7.18: Inorder traversal of a binary tree.

Binary Search Trees

Let *S* be a set whose elements have an order relation. For example, *S* could be a set of integers. A ***binary search*** tree for *S* is a proper binary tree *T* such that:

- Each internal node *p* of *T* stores an element of *S*, denoted with $x(p)$

- For each internal node *p* of *T*, the elements stored in the left subtree of *p* are less than or equal to $x(p)$ and the elements stored in the right subtree of *p* are greater than or equal to $x(p)$

- The external nodes of *T* do not store any element

An inorder traversal of the internal nodes of a binary search tree *T* visits the elements in nondecreasing order. (See Figure 7.19.)

We can use a binary search tree *T* to locate an element with a certain value *x* by traversing down the tree *T*. At each internal node we compare the value of the current node to our search element *x*. If the answer to the question is "smaller," then the search continues in the left subtree. If the answer is "equal," then the search terminates successfully. If the answer is "greater," then the search continues in the right subtree. Finally, if we reach an external node (which is empty), then the search terminates unsuccessfully. (See Figure 7.19.)

Note that the time for searching in a binary search tree *T* is proportional to the height of *T*. Recall from Proposition 7.10 that the height of a tree with *n* nodes can be as small as $O(\log n)$ or as large as $\Omega(n)$. Thus, binary search trees are most efficient when they have small height. We illustrate an example search in a binary search tree in Figure 7.19. We study binary search trees in more detail in Section 10.1.
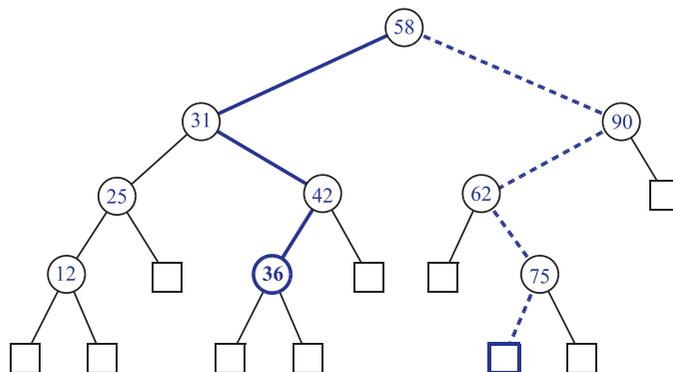
Figure 7.19: A binary search tree storing integers. The blue solid path is traversed when searching (successfully) for 36. The blue dashed path is traversed when searching (unsuccessfully) for 70.

Using Inorder Traversal for Tree Drawing

The inorder traversal can also be applied to the problem of computing a drawing of a binary tree. We can draw a binary tree T with an algorithm that assigns x- and y-coordinates to a node p of T using the following two rules (see Figure 7.20).

- $x(p)$ is the number of nodes visited before p in the inorder traversal of T.
- $y(p)$ is the depth of p in T.

In this application, we take the convention common in computer graphics that x-coordinates increase left to right and y-coordinates increase top to bottom. So the origin is in the upper left corner of the computer screen.
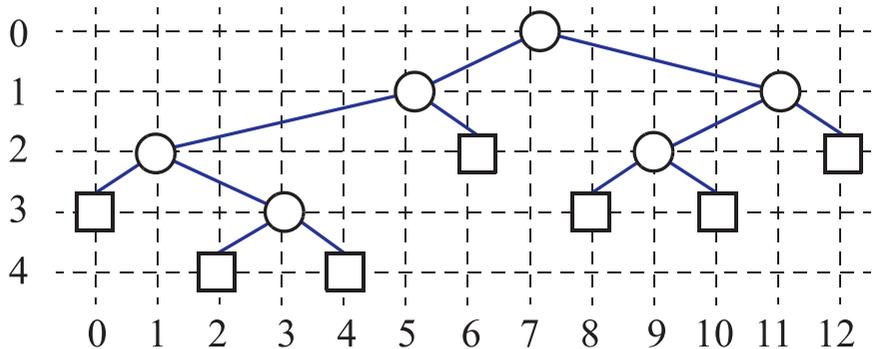
Figure 7.20: The inorder drawing algorithm for a binary tree.

The Euler Tour Traversal of a Binary Tree

The tree-traversal algorithms we have discussed so far are all forms of iterators. Each traversal visits the nodes of a tree in a certain order, and is guaranteed to visit each node exactly once. We can unify the tree-traversal algorithms given above into a single framework, however, by relaxing the requirement that each node be visited exactly once. The resulting traversal method is called the ***Euler tour traversal***, which we study next. The advantage of this traversal is that it allows for more general kinds of algorithms to be expressed easily.

The Euler tour traversal of a binary tree T can be informally defined as a "walk" around T, where we start by going from the root toward its left child, viewing the edges of T as being "walls" that we always keep to our left. (See Figure 7.21.) Each node p of T is encountered three times by the Euler tour:

- "On the left" (before the Euler tour of p's left subtree)
- "From below" (between the Euler tours of p's two subtrees)
- "On the right" (after the Euler tour of p's right subtree)

If p is external, then these three "visits" actually all happen at the same time.
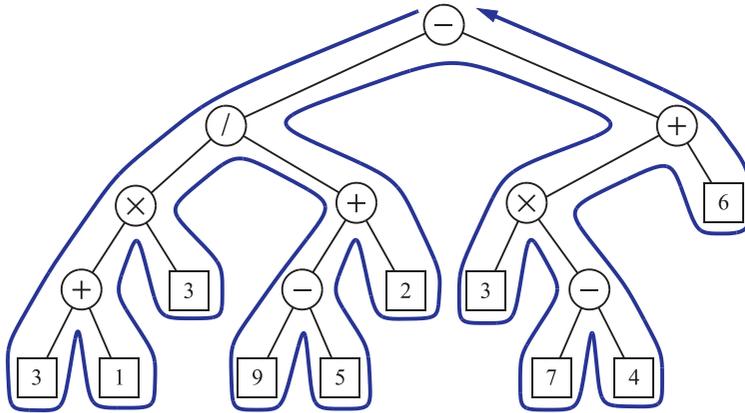
Figure 7.21: Euler tour traversal of a binary tree.

We give pseudo-code for the Euler tour of the subtree rooted at a node p in Code Fragment 7.28.

Algorithm eulerTour(T, p):

    perform the action for visiting node p on the left

    if p is an internal node **then**

      recursively tour the left subtree of p by calling eulerTour(T, p.left())

    perform the action for visiting node p from below

    if p is an internal node **then**

      recursively tour the right subtree of p by calling eulerTour(T, p.right())

    perform the action for visiting node p on the right

Code Fragment 7.28: Algorithm eulerTour for computing the Euler tour traversal of the subtree of a binary tree T rooted at a node p.

The preorder traversal of a binary tree is equivalent to an Euler tour traversal in which each node has an associated "visit" action occur only when it is encountered on the left. Likewise, the inorder and postorder traversals of a binary tree are equivalent to an Euler tour, where each node has an associated "visit" action occur only when it is encountered from below or on the right, respectively.

The Euler tour traversal extends the preorder, inorder, and postorder traversals, but it can also perform other kinds of traversals. For example, suppose we wish to compute the number of descendants of each node p in an n node binary tree T. We start an Euler tour by initializing a counter to 0, and then increment the counter each time we visit a node on the left. To determine the number of descendants of a node p, we compute the difference between the values of the counter when p is visited on the left and when it is visited on the right, and add 1. This simple rule gives us the number of descendants of p, because each node in the subtree rooted

at p is counted between p's visit on the left and p's visit on the right. Therefore, we have an $O(n)$-time method for computing the number of descendants of each node in T.

The running time of the Euler tour traversal is easy to analyze, assuming that visiting a node takes $O(1)$ time. Namely, in each traversal, we spend a constant amount of time at each node of the tree during the traversal, so the overall running time is $O(n)$ for an n node tree.

Another application of the Euler tour traversal is to print a fully parenthesized arithmetic expression from its expression tree (Example 7.9). Algorithm printExpression, shown in Code Fragment 7.29, accomplishes this task by performing the following actions in an Euler tour:

- "On the left" action: if the node is internal, print "("
- "From below" action: print the value or operator stored at the node
- "On the right" action: if the node is internal, print ")"

Algorithm printExpression(T, p):
  if p.isExternal() **then**
    print the value stored at p
  else
    print "("
    printExpression(T, p.left())
    print the operator stored at p
    printExpression(T, p.right())
    print ")"

Code Fragment 7.29: An algorithm for printing the arithmetic expression associated with the subtree of an arithmetic-expression tree T rooted at p.

7.3.7 The Template Function Pattern

The tree traversal functions described above are actually examples of an interesting object-oriented software design pattern, the ***template function pattern***. This is not to be confused with templated classes or functions in C++, but the principal is similar. The template function pattern describes a generic computation mechanism that can be specialized for a particular application by redefining certain steps.

Euler Tour with the Template Function Pattern

Following the template function pattern, we can design an algorithm, template-EulerTour, that implements a generic Euler tour traversal of a binary tree. When

called on a node p, function templateEulerTour calls several other auxiliary func-
tions at different phases of the traversal. First of all, it creates a three-element
structure r to store the result of the computation calling auxiliary function initRe-
sult. Next, if p is an external node, templateEulerTour calls auxiliary function
visitExternal, else (p is an internal node) templateEulerTour executes the follow-
ing steps:

- Calls auxiliary function visitLeft, which performs the computations associ-
  ated with encountering the node on the left
- Recursively calls itself on the left child
- Calls auxiliary function visitBelow, which performs the computations asso-
  ciated with encountering the node from below
- Recursively calls itself on the right subtree
- Calls auxiliary function visitRight, which performs the computations associ-
  ated with encountering the node on the right

Finally, templateEulerTour returns the result of the computation by calling aux-
iliary function returnResult. Function templateEulerTour can be viewed as a ***tem-
plate*** or "skeleton" of an Euler tour. (See Code Fragment 7.30.)

Algorithm templateEulerTour(T, p):
    $r \leftarrow$ initResult()
    if p.isExternal() **then**
        $r.finalResult \leftarrow$ visitExternal(T, p, r)
    else
        visitLeft(T, p, r)
        $r.leftResult \leftarrow$ templateEulerTour(T, p.left())
        visitBelow(T, p, r)
        $r.rightResult \leftarrow$ templateEulerTour(T, p.right())
        visitRight(T, p, r)
    return returnResult(r)

Code Fragment 7.30: Function templateEulerTour for computing a generic Euler
tour traversal of the subtree of a binary tree T rooted at a node p, following the
template function pattern. This function calls the functions initResult, visitExter-
nal, visitLeft, visitBelow, visitRight, and returnResult.

In an object-oriented context, we can then write a class EulerTour that:

- Contains function templateEulerTour
- Contains all the auxiliary functions called by templateEulerTour as empty
  place holders (that is, with no instructions or returning NULL)
- Contains a function execute that calls templateEulerTour(T, T.root())

Class EulerTour itself does not perform any useful computation. However, we can extend it with the inheritance mechanism and override the empty functions to do useful tasks.

Template Function Examples

As a first example, we can evaluate the expression associated with an arithmetic-expression tree (see Example 7.9) by writing a new class EvaluateExpression that:

- Extends class EulerTour
- Overrides function initResult by returning an array of three numbers
- Overrides function visitExternal by returning the value stored at the node
- Overrides function visitRight by combining *r.leftResult* and *r.rightResult* with the operator stored at the node, and setting *r.finalResult* equal to the result of the operation
- Overrides function returnResult by returning *r.finalResult*

This approach should be compared with the direct implementation of the algorithm shown in Code Fragment 7.26.

As a second example, we can print the expression associated with an arithmetic-expression tree (see Example 7.9) using a new class PrintExpression that:

- Extends class EulerTour
- Overrides function visitExternal by printing the value of the variable or constant associated with the node
- Overrides function visitLeft by printing "("
- Overrides function visitBelow by printing the operator associated with the node
- Overrides function visitRight by printing ")"

This approach should be compared with the direct implementation of the algorithm shown in Code Fragment 7.29.

C++ Implementation

A complete C++ implementation of the generic EulerTour class and of its specializations EvaluateExpressionTour and PrintExpressionTour are shown in Code Fragments 7.31 through 7.34. These are based on a linked binary tree implementation.

We begin by defining a local structure Result with fields *leftResult*, *rightResult*, and *finalResult*, which store the intermediate results of the tour. In order to avoid typing lengthy qualified type names, we give two type definitions, BinaryTree and Position, for the tree and a position in the tree, respectively. The only data member is a pointer to the binary tree. We provide a simple function, called initialize, that sets this pointer to an existing binary tree. The remaining functions are protected,

since they are not invoked directly, but rather by the derived classes, which produce
the desired specialized behavior.

```
template <typename E, typename R>     // element and result types
class EulerTour {                       // a template for Euler tour
protected:
  struct Result {                       // stores tour results
    R leftResult;                       // result from left subtree
    R rightResult;                      // result from right subtree
    R finalResult;                      // combined result
  };
  typedef BinaryTree<E> BinaryTree;     // the tree
  typedef typename BinaryTree::Position Position; // a position in the tree
protected:                              // data member
  const BinaryTree* tree;               // pointer to the tree
public:
  void initialize(const BinaryTree& T)  // initialize
    { tree = &T; }
protected:                              // local utilities
  int eulerTour(const Position& p) const;  // perform the Euler tour
                                        // functions given by subclasses
  virtual void visitExternal(const Position& p, Result& r) const {}
  virtual void visitLeft(const Position& p, Result& r) const {}
  virtual void visitBelow(const Position& p, Result& r) const {}
  virtual void visitRight(const Position& p, Result& r) const {}
  Result initResult() const { return Result(); }
  int result(const Result& r) const { return r.finalResult; }
};
```

Code Fragment 7.31: Class EulerTour defining a generic Euler tour of a binary tree.
This class realizes the template function pattern and must be specialized in order to
generate an interesting computation.

Next, in Code Fragment 7.32, we present the principal traversal function, called
eulerTour. This recursive function performs an Euler traversal on the tree and in-
vokes the appropriate functions as it goes. If run on the generic Euler tree, noth-
ing interesting would result, because these functions (as defined in Code Frag-
ment 7.31) do nothing. It is up to the derived functions to provide more interesting
definitions for these generic functions.

In Code Fragment 7.33, we present our first example of a derived class us-
ing the template pattern, called EvaluateExpressionTour. It evaluates an integer
arithmetic-expression tree. We assume that each external node of an expression
tree provides a function called value, which returns the value associated with this
node. We assume that each internal node of an expression tree provides a function
called operation, which performs the operation associated with this node to the two
operands arising from its left and right subtrees, and returns the result.

```
template <typename E, typename R>       // do the tour
int EulerTour<E, R>::eulerTour(const Position& p) const {
  Result r = initResult();
  if (p.isExternal()) {                         // external node
    visitExternal(p, r);
  }
  else {                                        // internal node
    visitLeft(p, r);
    r.leftResult = eulerTour(p.left());         // recurse on left
    visitBelow(p, r);
    r.rightResult = eulerTour(p.right());       // recurse on right
    visitRight(p, r);
  }
  return result(r);
}
```

Code Fragment 7.32: The principal member function eulerTour, which recursively traverses the tree and accumulates the results.

Using these two functions, we can evaluate the expression recursively as we traverse the tree. The main entry point is the function execute, which initializes the tree, invokes the recursive Euler tour starting at the root, and prints the final result. For example, given the expression tree of Figure 7.21, this procedure would output the string "The value is: -13".

```
template <typename E, typename R>
class EvaluateExpressionTour : public EulerTour<E, R> {
protected:                                     // shortcut type names
  typedef typename EulerTour<E, R>::BinaryTree BinaryTree;
  typedef typename EulerTour<E, R>::Position Position;
  typedef typename EulerTour<E, R>::Result Result;
public:
  void execute(const BinaryTree& T) {      // execute the tour
    initialize(T);
    std::cout << "The value is: " << eulerTour(T.root()) << "\n";
  }
protected:                                     // leaf: return value
  virtual void visitExternal(const Position& p, Result& r) const
    { r.finalResult = (*p).value(); }
                                               // internal: do operation
  virtual void visitRight(const Position& p, Result& r) const
    { r.finalResult = (*p).operation(r.leftResult, r.rightResult); }
};
```

Code Fragment 7.33: Implementation of class EvaluateExpressionTour which specializes EulerTour to evaluate the expression associated with an arithmetic-expression tree.

Finally, in Code Fragment 7.34, we present a second example of a derived class, called PrintExpressionTour. In contrast to the previous function, which evaluates the value of an expression tree, this one prints the expression. We assume that each node of an expression tree provides a function called print. For each external node, this function prints the value associated with this node. For each internal node, this function prints the operator, for example, printing "+" for addition or "*" for multiplication.

```
template <typename E, typename R>
class PrintExpressionTour : public EulerTour<E, R> {
protected: // ...same type name shortcuts as in EvaluateExpressionTour
public:
  void execute(const BinaryTree& T) {      // execute the tour
    initialize(T);
    cout << "Expression: "; eulerTour(T.root()); cout << endl;
  }
protected:                                 // leaf: print value
  virtual void visitExternal(const Position& p, Result& r) const
    { (*p).print(); }
                                           // left: open new expression
  virtual void visitLeft(const Position& p, Result& r) const
    { cout << "("; }
                                           // below: print operator
  virtual void visitBelow(const Position& p, Result& r) const
    { (*p).print(); }
                                           // right: close expression
  virtual void visitRight(const Position& p, Result& r) const
    { cout << ")"; }
};
```

Code Fragment 7.34: A class that prints an arithmetic-expression tree.

When entering a subtree, the function visitLeft has been overridden to print "(" and on exiting a subtree, the function visitRight has been overridden to print ")." The main entry point is the function execute, which initializes the tree, and invokes the recursive Euler tour starting at the root. When combined, these functions print the entire expression (albeit with lots of redundant parentheses). For example, given the expression tree of Figure 7.21, this procedure would output the following string.

$$((((3 + 1) * 3) / ((9 - 5) + 2)) - ((3 * (7 - 4)) + 6))$$

7.3.8 Representing General Trees with Binary Trees

An alternative representation of a general tree T is obtained by transforming T into a binary tree T'. (See Figure 7.22.) We assume that either T is ordered or that it has been arbitrarily ordered. The transformation is as follows:

- For each node u of T, there is an internal node u' of T' associated with u
- If u is an external node of T and does not have a sibling immediately following it, then the children of u' in T' are external nodes
- If u is an internal node of T and v is the first child of u in T, then v' is the left child of u' in T
- If node v has a sibling w immediately following it, then w' is the right child of v' in T'

Note that the external nodes of T' are not associated with nodes of T, and serve only as place holders (hence, may even be null).

(a)   (b)
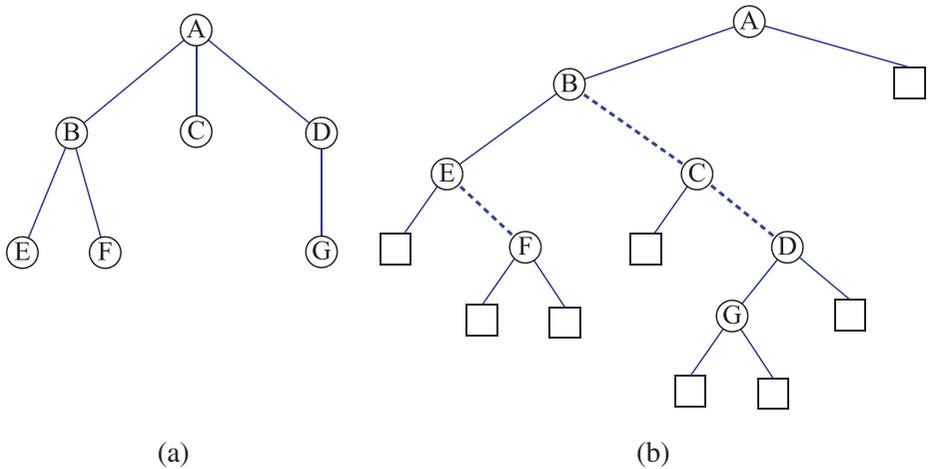
Figure 7.22: Representation of a tree by means of a binary tree: (a) tree T; (b) binary tree T' associated with T. The dashed edges connect nodes of T' associated with sibling nodes of T.

It is easy to maintain the correspondence between T and T', and to express operations in T in terms of corresponding operations in T'. Intuitively, we can think of the correspondence in terms of a conversion of T into T' that takes each set of siblings $\{v_1, v_2, \ldots, v_k\}$ in T with parent v and replaces it with a chain of right children rooted at v_1, which then becomes the left child of v.

7.4   Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-7.1 Describe an algorithm for counting the number of left external nodes in a binary tree, using the Binary tree ADT.

R-7.2 The following questions refer to the tree of Figure 7.3.

    a. Which node is the root?
    b. What are the internal nodes?
    c. How many descendents does node cs016/ have?
    d. How many ancestors does node cs016/ have?
    e. What are the siblings of node homeworks/?
    f. Which nodes are in the subtree rooted at node projects/?
    g. What is the depth of node papers/?
    h. What is the height of the tree?

R-7.3 Find the value of the arithmetic expression associated with each subtree of the binary tree of Figure 7.11.

R-7.4 Let T be an n-node improper binary tree (that is, each internal node has one or two children). Describe how to represent T by means of a *proper* binary tree T' with $O(n)$ nodes.

R-7.5 What are the minimum and maximum number of internal and external nodes in an improper binary tree with n nodes?

R-7.6 Show a tree achieving the worst-case running time for algorithm depth.

R-7.7 Give a justification of Proposition 7.4.

R-7.8 What is the running time of algorithm height2(T,v) (Code Fragment 7.7) when called on a node v distinct from the root of T?

R-7.9 Let T be the tree of Figure 7.3.

    a. Give the output of preorderPrint$(T,T.\text{root}())$ (Code Fragment 7.10).
    b. Give the output of parenPrint$(T,T.\text{root}())$(Code Fragment cod:paren:Print).

R-7.10 Describe a modification to the parenPrint function given in Code Fragment 7.11, so that it uses the size function for string objects to output the parenthetic representation of a tree with line breaks and spaces added to display the tree in a text window that is 80 characters wide.

R-7.11 Draw an arithmetic-expression tree that has four external nodes, storing the numbers 1, 5, 6, and 7 (with each number stored in a distinct external node, but not necessarily in this order), and has three internal nodes, each storing an operator from the set $\{+, -, \times, /\}$, so that the value of the root is 21. The operators may return and act on fractions, and an operator may be used more than once.

R-7.12 Let T be an ordered tree with more than one node. Is it possible that the preorder traversal of T visits the nodes in the same order as the postorder traversal of T? If so, give an example; otherwise, argue why this cannot occur. Likewise, is it possible that the preorder traversal of T visits the nodes in the reverse order of the postorder traversal of T? If so, give an example; otherwise, argue why this cannot occur.

R-7.13 Answer the previous question for the case when T is a proper binary tree with more than one node.

R-7.14 Let T be a tree with n nodes. What is the running time of the function parenPrint(T, T.root())? (See Code Fragment 7.11.)

R-7.15 Draw a (single) binary tree T, such that:

- Each internal node of T stores a single character
- A *preorder* traversal of T yields EXAMFUN
- An *inorder* traversal of T yields MAFXUEN

R-7.16 Answer the following questions so as to justify Proposition 7.10.

a. What is the minimum number of external nodes for a binary tree with height h? Justify your answer.
b. What is the maximum number of external nodes for a binary tree with height h? Justify your answer.
c. Let T be a binary tree with height h and n nodes. Show that

$$\log(n+1) - 1 \le h \le (n-1)/2.$$

d. For which values of n and h can the above lower and upper bounds on h be attained with equality?

R-7.17 Describe a generalization of the Euler tour traversal of trees such that each internal node has three children. Describe how you could use this traversal to compute the height of each node in such a tree.

R-7.18 Modify the C++ function preorderPrint, given in Code Fragment 7.10, so that it will print the strings associated with the nodes of a tree one per line, and indented proportionally to the depth of the node.

R-7.19 Let T be the tree of Figure 7.3. Draw, as best as you can, the output of the algorithm postorderPrint(T, T.root()) (Code Fragment 7.13).

R-7.20 Let T be the tree of Figure 7.9. Compute, in terms of the values given in this figure, the output of algorithm diskSpace$(T, T$.root$())$. (See Code Fragment 7.14.)

R-7.21 Let T be the binary tree of Figure 7.11.

a. Give the output of preorderPrint$(T, T$.root$())$ (Code Fragment 7.10).

b. Give the output of the function printExpression$(T, T$.root$())$ (Code Fragment 7.29).

R-7.22 Describe, in pseudo-code, an algorithm for computing the number of descendents of each node of a binary tree. The algorithm should be based on the Euler tour traversal.

R-7.23 Let T be a (possibly improper) binary tree with n nodes, and let D be the sum of the depths of all the external nodes of T. Show that if T has the minimum number of external nodes possible, then D is $O(n)$ and if T has the maximum number of external nodes possible, then D is $O(n \log n)$.

R-7.24 Let T be a binary tree with n nodes, and let f be the level numbering of the nodes of T as given in Section 7.3.5.

a. Show that, for every node v of T, $f(v) \leq 2^n - 1$.

b. Show an example of a binary tree with seven nodes that attains the above upper bound on $f(v)$ for some node v.

R-7.25 Draw the binary tree representation of the following arithmetic expression: "$(((5+2)*(2-1))/((2+9)+((7-2)-1))*8)$."

R-7.26 Let T be a binary tree with n nodes that is realized with a vector, S, and let f be the level numbering of the nodes in T as given in Section 7.3.5. Give pseudo-code descriptions of each of the functions root, parent, leftChild, rightChild, isExternal, and isRoot.

R-7.27 Show how to use the Euler tour traversal to compute the level number, defined in Section 7.3.5, of each node in a binary tree T.

Creativity

C-7.1 Show that there are more than 2^n different potentially improper binary trees with n internal nodes, where two trees are considered different if they can be drawn as different looking trees.

C-7.2 Describe an efficient algorithm for converting a fully balanced string of parentheses into an equivalent tree. The tree associated with such a string is defined recursively. The outer-most pair of balanced parentheses is associated with the root and each substring inside this pair, defined by the substring between two balanced parentheses, is associated with a subtree of this root.

C-7.3 For each node v in a tree T, let $pre(v)$ be the rank of v in a preorder traversal of T, let $post(v)$ be the rank of v in a postorder traversal of T, let $depth(v)$ be the depth of v, and let $desc(v)$ be the number of descendents of v, not counting v itself. Derive a formula defining $post(v)$ in terms of $desc(v)$, $depth(v)$, and $pre(v)$, for each node v in T.

C-7.4 Let T be a tree whose nodes store strings. Give an algorithm that computes and prints, for every internal node v of T, the string stored at v and the height of the subtree rooted at v.

C-7.5 Design algorithms for the following operations for a binary tree T.

- preorderNext(v): return the node visited after node v in a preorder traversal of T.
- inorderNext(v): return the node visited after node v in an inorder traversal of T.
- postorderNext(v): return the node visited after node v in a postorder traversal of T.

What are the worst-case running times of your algorithms?

C-7.6 Give an $O(n)$-time algorithm for computing the depth of all the nodes of a tree T, where n is the number of nodes of T.

C-7.7 The ***indented parenthetic representation*** of a tree T is a variation of the parenthetic representation of T (see Figure 7.7) that uses indentation and line breaks as illustrated in Figure 7.23. Give an algorithm that prints this representation of a tree.
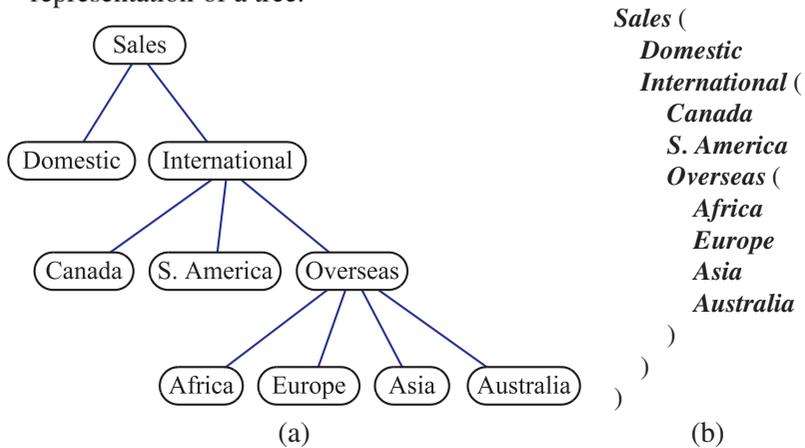
Figure 7.23: (a) Tree T; (b) indented parenthetic representation of T.

C-7.8 Let T be a (possibly improper) binary tree with n nodes, and let D be the sum of the depths of all the external nodes of T. Describe a configuration for T such that D is $\Omega(n^2)$. Such a tree would be the worst case for the asymptotic running time of Algorithm height1 (Code Fragment 7.6).

C-7.9 For a tree T, let n_I denote the number of its internal nodes, and let n_E denote the number of its external nodes. Show that if every internal node in T has exactly 3 children, then $n_E = 2n_I + 1$.

C-7.10 The update operations expandExternal and removeAboveExternal do not permit the creation of an improper binary tree. Give pseudo-code descriptions for alternate update operations suitable for improper binary trees. You may need to define new query operations as well.

C-7.11 The ***balance factor*** of an internal node v of a binary tree is the difference between the heights of the right and left subtrees of v. Show how to specialize the Euler tour traversal of Section 7.3.7 to print the balance factors of all the nodes of a binary tree.

C-7.12 Two ordered trees T' and T'' are said to be ***isomorphic*** if one of the following holds:
- Both T' and T'' consist of a single node
- Both T' and T'' have the same number k of subtrees, and the ith subtree of T' is isomorphic to the ith subtree of T'', for $i = 1, \ldots, k$.

Design an algorithm that tests whether two given ordered trees are isomorphic. What is the running time of your algorithm?

C-7.13 Extend the concept of an Euler tour to an ordered tree that is not necessarily a binary tree.

C-7.14 As mentioned in Exercise C-5.8, ***postfix notation*** is an unambiguous way of writing an arithmetic expression without parentheses. It is defined so that if "$(exp_1) \circ (exp_2)$" is a normal (infix) fully parenthesized expression with operation "$\circ$," then its postfix equivalent is "$pexp_1 \ pexp_2 \circ$," where $pexp_1$ is the postfix version of exp_1 and $pexp_2$ is the postfix version of exp_2. The postfix version of a single number of variables is just that number or variable. So, for example, the postfix version of the infix expression "$((5+2)*(8-3))/4$" is "$5 \ 2 + 8 \ 3 - *4 \ /$." Give an efficient algorithm, that when given an expression tree, outputs the expression in postfix notation.

C-7.15 Given a proper binary tree T, define the ***reflection*** of T to be the binary tree T' such that each node v in T is also in T', but the left child of v in T is v's right child in T' and the right child of v in T is v's left child in T'. Show that a preorder traversal of a proper binary tree T is the same as the postorder traversal of T's reflection, but in reverse order.

C-7.16 Algorithm preorderDraw draws a binary tree T by assigning x- and y-coordinates to each node v such that $x(v)$ is the number of nodes preceding v in the preorder traversal of T and $y(v)$ is the depth of v in T. Algorithm postorderDraw is similar to preorderDraw but assigns x-coordinates using a postorder traversal.

a. Show that the drawing of T produced by preorderDraw has no pairs of crossing edges.

b. Redraw the binary tree of Figure 7.20 using preorderDraw.

c. Show that the drawing of T produced by postorderDraw has no pairs of crossing edges.

d. Redraw the binary tree of Figure 7.20 using postorderDraw.

C-7.17 Let a visit action in the Euler tour traversal be denoted by a pair (v, a), where v is the visited node and a is one of **left**, **below**, or **right**. Design an algorithm for performing operation tourNext(v, a), which returns the visit action (w, b) following (v, a). What is the worst-case running time of your algorithm?

C-7.18 Algorithm preorderDraw draws a binary tree T by assigning x- and y-coordinates to each node v as follows:

- Set $x(v)$ equal to the number of nodes preceding v in the preorder traversal of T.
- Set $y(v)$ equal to the depth of v in T.

a. Show that the drawing of T produced by algorithm preorderDraw has no pairs of crossing edges.

b. Use algorithm preorderDraw to redraw the binary tree shown in Figure 7.20.

c. Use algorithm postorderDraw, which is similar to preorderDraw but assigns x-coordinates using a postorder traversal, to redraw the binary tree of Figure 7.20.

C-7.19 Design an algorithm for drawing general trees that generalizes the inorder traversal approach for drawing binary trees.

C-7.20 Consider a variation of the linked data structure for binary trees where each node object has pointers to the node objects of the children but not to the node object of the parent. Describe an implementation of the functions of a binary tree with this data structure and analyze the time complexity for these functions.

C-7.21 Design an alternative implementation of the linked data structure for binary trees using a class for nodes that specializes into subclasses for an internal node, an external node, and the root node.

C-7.22 Provide the missing housekeeping functions (destructor, copy constructor, and assignment operator) for the class LinkedBinaryTree given in Code Fragment 7.19.

C-7.23 Our linked binary tree implementation given in Code Fragment 7.19 assumes that the tree is proper. Design an alternative implementation of the linked data structure for a general (possibly improper) binary tree.

C-7.24 Let T be a tree with n nodes. Define the ***lowest common ancestor*** (LCA) between two nodes v and w as the lowest node in T that has both v and w as descendents (where we allow a node to be a descendent of itself). Given two nodes v and w, describe an efficient algorithm for finding the LCA of v and w. What is the running time of your method?

C-7.25 Let T be a tree with n nodes, and, for any node v in T, let d_v denote the depth of v in T. The ***distance*** between two nodes v and w in T is $d_v + d_w - 2d_u$, where u is the LCA u of v and w (as defined in the previous exercise). The ***diameter*** of T is the maximum distance between two nodes in T. Describe an efficient algorithm for finding the diameter of T. What is the running time of your method?

C-7.26 Suppose each node v of a binary tree T is labeled with its value $f(v)$ in a level numbering of T. Design a fast method for determining $f(u)$ for the lowest common ancestor (LCA), u, of two nodes v and w in T, given $f(v)$ and $f(w)$. You do not need to find node u, just compute its level-numbering label.

C-7.27 Justify Table 7.1, summarizing the running time of the functions of a tree represented with a linked structure, by providing, for each function, a description of its implementation, and an analysis of its running time.

C-7.28 Describe efficient implementations of the expandExternal and remove-AboveExternal binary tree update functions, described in Section 7.3.4, for the case when the binary tree is implemented using a vector S, where S is realized using an expandable array. Your functions should work even for null external nodes, assuming we represent such a node as a wrapper object storing an index to an empty or nonexistent cell in S. What are the worst-case running times of these functions? What is the running time of removeAboveExternal if the internal node removed has only external node children?

C-7.29 Describe a nonrecursive method for evaluating a binary tree representing an arithmetic expression.

C-7.30 Let T be a binary tree with n nodes. Define a ***Roman node*** to be a node v in T, such that the number of descendents in v's left subtree differ from the number of descendents in v's right subtree by at most 5. Describe a linear-time method for finding each node v of T, such that v is not a Roman node, but all of v descendents are Roman nodes.

C-7.31 Let T' be the binary tree representing a tree T. (See Section 7.3.8.)
   a. Is a preorder traversal of T' equivalent to a preorder traversal of T?
   b. Is a postorder traversal of T' equivalent to a postorder traversal of T?
   c. Is an inorder traversal of T' equivalent to some well-structured traversal of T?

C-7.32 Describe a nonrecursive method for performing an Euler tour traversal of a binary tree that runs in linear time and does not use a stack.

(Hint: You can tell which visit action to perform at a node by taking note of where you are coming from.)

C-7.33 Describe, in pseudo-code, a nonrecursive method for performing an in-order traversal of a binary tree in linear time.

(Hint: Use a stack.)

C-7.34 Let T be a binary tree with n nodes (T may or may not be realized with a vector). Give a linear-time method that uses the functions of the Binary-Tree interface to traverse the nodes of T by increasing values of the level numbering function f given in Section 7.3.5. This traversal is known as the *level order traversal*.

(Hint: Use a queue.)

C-7.35 The *path length* of a tree T is the sum of the depths of all the nodes in T. Describe a linear-time method for computing the path length of a tree T (which is not necessarily binary).

C-7.36 Define the *internal path length*, $I(T)$, of a tree T, to be the sum of the depths of all the internal nodes in T. Likewise, define the *external path length*, $E(T)$, of a tree T, to be the sum of the depths of all the external nodes in T. Show that if T is a binary tree with n internal nodes, then $E(T) = I(T) + 2n$.

(Hint: Use the fact that we can build T from a single root node via a series of n expandExternal operations.)

Projects

P-7.1 Write a program that takes as input a rooted tree T and a node v of T and converts T to another tree with the same set of node adjacencies but now rooted at v.

P-7.2 Give a fully generic implementation of the class LinkedBinaryTree using class templates and taking into account error conditions.

P-7.3 Implement the binary tree ADT using a vector.

P-7.4 Implement the binary tree ADT using a linked structure.

P-7.5 Write a program that draws a binary tree.

P-7.6 Write a program that draws a general tree.

P-7.7 Write a program that can input and display a person's family tree.

P-7.8 Implement the binary tree representation of the tree ADT. You may reuse the LinkedBinaryTree implementation of a binary tree.

P-7.9 A ***slicing floorplan*** is a decomposition of a rectangle with horizontal and vertical sides using horizontal and vertical ***cuts*** (see Figure 7.24(a)). A slicing floorplan can be represented by a binary tree, called a ***slicing tree***, whose internal nodes represent the cuts, and whose external nodes represent the ***basic rectangles*** into which the floorplan is decomposed by the cuts (see Figure 7.24(b)). The ***compaction problem*** is defined as follows. Assume that each basic rectangle of a slicing floorplan is assigned a minimum width w and a minimum height h. The compaction problem is to find the smallest possible height and width for each rectangle of the slicing floorplan that is compatible with the minimum dimensions of the basic rectangles. Namely, this problem requires the assignment of values $h(v)$ and $w(v)$ to each node v of the slicing tree, such that

$$
w(v) = \begin{cases}
w & \text{if } v \text{ is an external node whose basic rectangle has minimum width } w \\[2ex]
\max(w(w), w(z)) & \text{if } v \text{ is an internal node associated with a horizontal cut with left child } w \text{ and right child } z \\[2ex]
w(w) + w(z) & \text{if } v \text{ is an internal node associated with a vertical cut with left child } w \text{ and right child } z
\end{cases}
$$

$$
h(v) = \begin{cases}
h & \text{if } v \text{ is an external node whose basic rectangle has minimum height } h \\[2ex]
h(w) + h(z) & \text{if } v \text{ is an internal node associated with a horizontal cut with left child } w \text{ and right child } z \\[2ex]
\max(h(w), h(z)) & \text{if } v \text{ is an internal node associated with a vertical cut with left child } w \text{ and right child } z
\end{cases}
$$

Design a data structure for slicing floorplans that supports the following operations:

- Create a floorplan consisting of a single basic rectangle
- Decompose a basic rectangle by means of a horizontal cut
- Decompose a basic rectangle by means of a vertical cut
- Assign minimum height and width to a basic rectangle
- Draw the slicing tree associated with the floorplan
- Compact the floorplan
- Draw the compacted floorplan
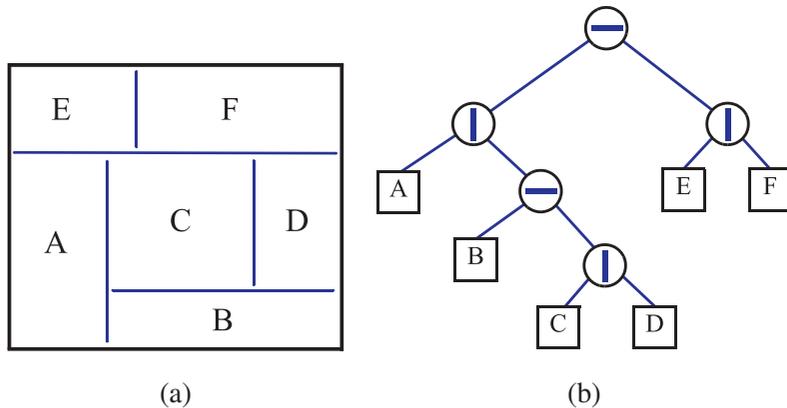
Figure 7.24: (a) Slicing floorplan; (b) slicing tree associated with the floorplan.

P-7.10 Write a program that takes, as input, a fully parenthesized, arithmetic expression and converts it to a binary expression tree. Your program should display the tree in some way and also print the value associated with the root. For an additional challenge, allow for the leaves to store variables of the form x_1, x_2, x_3, and so on, which are initially 0 and which can be updated interactively by your program, with the corresponding update in the printed value of the root of the expression tree.

P-7.11 Write a program that can play Tic-Tac-Toe effectively. (See Section 3.1.3.) To do this, you will need to create a *game tree T*, which is a tree where each node corresponds to a *game configuration*, which, in this case, is a representation of the tic-tac-toe board. The root node corresponds to the initial configuration. For each internal node v in T, the children of v correspond to the game states we can reach from v's game state in a single legal move for the appropriate player, A (the first player) or B (the second player). Nodes at even depths correspond to moves for A and nodes at odd depths correspond to moves for B. External nodes are either final game states or are at a depth beyond which we don't want to explore. We score each external node with a value that indicates how good this state is for player A. In large games, like chess, we have to use a heuristic scoring function, but for small games, like tic-tac-toe, we can construct the entire game tree and score external nodes as $+1$, 0, -1, indicating whether player A has a win, draw, or lose in that configuration. A good algorithm for choosing moves is *minimax*. In this algorithm, we assign a score to each internal node v in T, such that if v represents A's turn, we compute v's score as the maximum of the scores of v's children (which corresponds to A's optimal play from v). If an internal node v represents B's turn, then we compute v's score as the minimum of the scores of v's children (which corresponds to B's optimal play from v).

Chapter Notes

Our use of the position abstraction derives from the ***position*** and ***node*** abstractions introduced by Aho, Hopcroft, and Ullman [5]. Discussions of the classic preorder, inorder, and postorder tree traversal methods can be found in Knuth's *Fundamental Algorithms* book [56]. The Euler tour traversal technique comes from the parallel algorithms community, as it is introduced by Tarjan and Vishkin [93] and is discussed by JáJá [49] and by Karp and Ramachandran [53]. The algorithm for drawing a tree is generally considered to be a part of the "folklore" of graph drawing algorithms. The reader interested in graph drawing is referred to the book by Di Battista, Eades, Tamassia and Tollis [28] and the survey by Tamassia and Liotta [92]. The puzzler in Exercise R-7.11 was communicated by Micha Sharir.
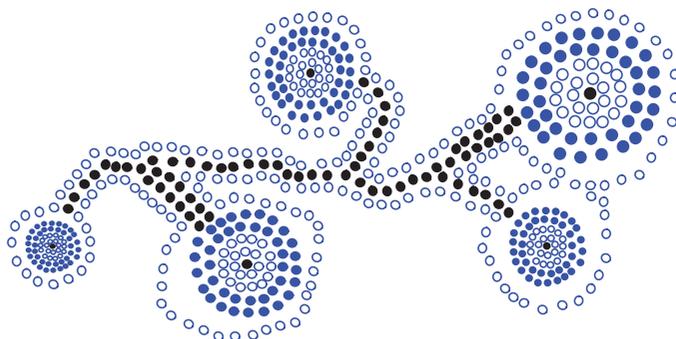
Contents

8.1   The Priority Queue Abstract Data Type

A *priority queue* is an abstract data type for storing a collection of prioritized elements that supports arbitrary element insertion but supports removal of elements in order of priority, that is, the element with first priority can be removed at any time. This ADT is fundamentally different from the position-based data structures such as stacks, queues, deques, lists, and even trees, we discussed in previous chapters. These other data structures store elements at specific positions, which are often positions in a linear arrangement of the elements determined by the insertion and deletion operations performed. The priority queue ADT stores elements according to their priorities, and has no external notion of "position."

8.1.1   Keys, Priorities, and Total Order Relations

Applications commonly require comparing and ranking objects according to parameters or properties, called "keys," that are assigned to each object in a collection. Formally, we define a *key* to be an object that is assigned to an element as a specific attribute for that element and that can be used to identify, rank, or weigh that element. Note that the key is assigned to an element, typically by a user or application; hence, a key might represent a property that an element did not originally possess.

The key an application assigns to an element is not necessarily unique, however, and an application may even change an element's key if it needs to. For example, we can compare companies by earnings or by number of employees; hence, either of these parameters can be used as a key for a company, depending on the information we wish to extract. Likewise, we can compare restaurants by a critic's food quality rating or by average entrée price. To achieve the most generality then, we allow a key to be of any type that is appropriate for a particular application.

As in the examples above, the key used for comparisons is often more than a single numerical value, such as price, length, weight, or speed. That is, a key can sometimes be a more complex property that cannot be quantified with a single number. For example, the priority of standby passengers is usually determined by taking into account a host of different factors, including frequent-flyer status, the fare paid, and check-in time. In some applications, the key for an object is data extracted from the object itself (for example, it might be a member variable storing the list price of a book, or the weight of a car). In other applications, the key is not part of the object but is externally generated by the application (for example, the quality rating given to a stock by a financial analyst, or the priority assigned to a standby passenger by a gate agent).

Comparing Keys with Total Orders

A priority queue needs a comparison rule that never contradicts itself. In order for a comparison rule, which we denote by $\leq$, to be robust in this way, it must define a **total order** relation, which is to say that the comparison rule is defined for every pair of keys and it must satisfy the following properties:

- **Reflexive property** : $k \leq k$
- **Antisymmetric property**: if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$
- **Transitive property**: if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$

Any comparison rule, $\leq$, that satisfies these three properties never leads to a comparison contradiction. In fact, such a rule defines a linear ordering relationship among a set of keys. If a finite collection of keys has a total order defined for it, then the notion of the **smallest** key, k_{min}, is well defined as the key, such that $k_{min} \leq k$, for any other key k in our collection.

A **priority queue** is a container of elements, each associated with a key. The name "priority queue" comes from the fact that keys determine the "priority" used to pick elements to be removed. The fundamental functions of a priority queue P are as follows:

insert(e): Insert the element e (with an implicit associated key value) into P.

min(): Return an element of P with the smallest associated key value, that is, an element whose key is less than or equal to that of every other element in P.

removeMin(): Remove from P the element min().

Note that more than one element can have the same key, which is why we were careful to define removeMin to remove not just any minimum element, but the same element returned by min. Some people refer to the removeMin function as extractMin.

There are many applications where operations insert and removeMin play an important role. We consider such an application in the example that follows.

Example 8.1: *Suppose a certain flight is fully booked an hour prior to departure. Because of the possibility of cancellations, the airline maintains a priority queue of standby passengers hoping to get a seat. The priority of each passenger is determined by the fare paid, the frequent-flyer status, and the time when the passenger is inserted into the priority queue. When a passenger requests to fly standby, the associated passenger object is inserted into the priority queue with an insert operation. Shortly before the flight departure, if seats become available (for example, due to last-minute cancellations), the airline repeatedly removes a standby passenger with first priority from the priority queue, using a combination of min and removeMin operations, and lets this person board.*

8.1.2   Comparators

An important issue in the priority queue ADT that we have so far left undefined is how to specify the total order relation for comparing the keys associated with each element. There are a number of ways of doing this, each having its particular advantages and disadvantages.

The most direct solution is to implement a different priority queue based on the element type and the manner of comparing elements. While this approach is arguably simple, it is certainly not very general, since it would require that we make many copies of essentially the same code. Maintaining multiple copies of the nearly equivalent code is messy and error prone.

A better approach would be to design the priority queue as a templated class, where the element type is specified by an abstract template argument, say E. We assume that each concrete class that could serve as an element of our priority queue provides a means for comparing two objects of type E. This could be done in many ways. Perhaps we require that each object of type E provides a function called comp that compares two objects of type E and determines which is larger. Perhaps we require that the programmer defines a function that overloads the C++ comparison operator "<" for two objects of type E. (Recall Section 1.4.2 for a discussion of operator overloading). In C++ jargon this is called a ***function object***.

Let us consider a more concrete example. Suppose that class Point2D defines a two-dimensional point. It has two public member functions, getX and getY, which access its *x* and *y* coordinates, respectively. We could define a lexicographical less-than operator as follows. If the *x* coordinates differ we use their relative values; otherwise, we use the relative values of the *y* coordinates.

```
bool operator<(const Point2D& p, const Point2D& q) {
  if (p.getX() == q.getX())    return p.getY() < q.getY();
  else                         return p.getX() < q.getX();
}
```

This approach of overloading the relational operators is general enough for many situations, but it relies on the assumption that objects of the same type are always compared in the same way. There are situations, however, where it is desirable to apply different comparisons to objects of the same type. Consider the following examples.

Example 8.2: *There are at least two ways of comparing the* C++ *character strings,* "4" *and* "12"*. In the* **lexicographic ordering***, which is an extension of the alphabetic ordering to character strings, we have* "4" > "12"*. But if we interpret these strings as integers, then* "4" < "12"*.*

Example 8.3: *A geometric algorithm may compare points p and q in two-dimensional space, by their x-coordinate (that is, $p \le q$ if $p_x \le q_x$), to sort them from left to right, while another algorithm may compare them by their y-coordinate (that is, $p \le q$ if $p_y \le q_y$), to sort them from bottom to top. In principle, there is nothing pertaining to the concept of a point that says whether points should be compared by x- or y-coordinates. Also, many other ways of comparing points can be defined (for example, we can compare the distances of p and q from the origin).*

There are a couple of ways to achieve the goal of independence of element type and comparison method. The most general approach, called the ***composition method***, is based on defining each entry of our priority queue to be a pair (e, k), consisting of an element e and a key k. The element part stores the data, and the key part stores the information that defines the priority ordering. Each key object defines its own comparison function. By changing the key class, we can change the way in which the queue is ordered. This approach is very general, because the key part does not need to depend on the data present in the element part. We study this approach in greater detail in Chapter 9.

The approach that we use is a bit simpler than the composition method. It is based on defining a special object, called a ***comparator***, whose job is to provide a definition of the comparison function between any two elements. This can be done in various ways. In C++, a comparator for element type E can be implemented as a class that defines a single function whose job is to compare two objects of type E. One way to do this is to overload the "()" operator. The resulting function takes two arguments, a and b, and returns a boolean whose value is true if $a < b$. For example, if "isLess" is the name of our comparator object, the comparison function is invoked using the following operation:

isLess(a, b): Return true if $a < b$ and false otherwise.

It might seem at first that defining just a less-than function is rather limited, but note that it is possible to derive all the other relational operators by combining less-than comparisons with other boolean operators. For example, we can test whether a and b are equal with (!isLess(a,b) && !isLess(b, a)). (See Exercise R-8.3.)

Defining and Using Comparator Objects

Let us consider a more concrete example of a comparator class. As mentioned in the above example, let us suppose that we have defined a class structure, called Point2D, for storing a two-dimensional point. In Code Fragment 8.1, we present two comparators. The comparator LeftRight implements a left-to-right order by comparing the x-coordinates of the points, and the comparator BottomTop implements a bottom-to-top order by comparing the y-coordinates of the points.

To use these comparators, we would declare two objects, one of each type. Let us call them leftRight and bottomTop. Observe that these objects store no

```
class LeftRight {                              // a left-right comparator
public:
  bool operator()(const Point2D& p, const Point2D& q) const
    { return p.getX() < q.getX(); }
};

class BottomTop {                             // a bottom-top comparator
public:
  bool operator()(const Point2D& p, const Point2D& q) const
    { return p.getY() < q.getY(); }
};
```

Code Fragment 8.1: Two comparator classes for comparing points. The first implements a left-to-right order and the second implements a bottom-to-top order.

data members. They are used solely for the purposes of specifying a particular comparison operator. Given two objects p and q, each of type Point2D, to test whether p is to the left of q, we would invoke leftRight(p,q), and to test whether p is below q, we would invoke bottomTop(p,q). Each invokes the "()" operator for the corresponding class.

Next, let us see how we can use our comparators to implement two different behaviors. Consider the generic function printSmaller shown in Code Fragment 8.2. It prints the smaller of its two arguments. The function definition is templated by the element type E and the comparator type C. The comparator class is assumed to implement a less-than function for two objects of type E. The function is given three arguments, the two elements p and q to be compared and an instance isLess of a comparator for these elements. The function invokes the comparator to determine which element is smaller, and then prints this value.

```
template <typename E, typename C>     // element type and comparator
void printSmaller(const E& p, const E& q, const C& isLess) {
  cout << (isLess(p, q) ? p : q) << endl; // print the smaller of p and q
}
```

Code Fragment 8.2: A generic function that prints the smaller of two elements, given a comparator for these elements.

Finally, let us see how we can apply our function on two points. The code is shown in Code Fragment 8.3. We declare to points p and q and initialize their coordinates. (We have not presented the class definition for Point2D, but let us assume that the constructor is given the x- and y-coordinates, and we have provided an output operator.) We then declare two comparator objects, one for a left-to-right ordering and the other for a bottom-to-top ordering. Finally, we invoke the function printSmaller on the two points, changing only the comparator objects in each case.

Observe that, depending on which comparator is provided, the call to the func-

```
Point2D p(1.3, 5.7), q(2.5, 0.6);        // two points
LeftRight leftRight;                      // a left-right comparator
BottomTop bottomTop;                      // a bottom-top comparator
printSmaller(p, q, leftRight);           // outputs: (1.3, 5.7)
printSmaller(p, q, bottomTop);           // outputs: (2.5, 0.6)
```

Code Fragment 8.3: The use of two comparators to implement different behaviors from the function printSmaller.

tion isLess in function printSmaller invokes either the "()" operator of class Left-Right or BottomTop. In this way, we obtain the desired result, two different behaviors for the same two-dimensional point class.

Through the use of comparators, a programmer can write a general priority queue implementation that works correctly in a wide variety of contexts. In particular, the priority queues presented in this chapter are generic classes that are templated by two types, the element E and the comparator C.

The comparator approach is a bit less general than the composition method, because the comparator bases its decisions on the contents of the elements themselves. In the composition method, the key may contain information that is not part of the element object. The comparator approach has the advantage of being simpler, since we can insert elements directly into our priority queue without creating element-key pairs. Furthermore, in Exercise R-8.4 we show that there is no real loss of generality in using comparators.

8.1.3 The Priority Queue ADT

Having described the priority queue abstract data type at an intuitive level, we now describe it in more detail. As an ADT, a priority queue P supports the following functions:

size(): Return the number of elements in P.

empty(): Return true if P is empty and false otherwise.

insert(e): Insert a new element e into P.

min(): Return a reference to an element of P with the smallest associated key value (but do not remove it); an error condition occurs if the priority queue is empty.

removeMin(): Remove from P the element referenced by min(); an error condition occurs if the priority queue is empty.

As mentioned above, the primary functions of the priority queue ADT are the insert, min, and removeMin operations. The other functions, size and empty, are generic collection operations. Note that we allow a priority queue to have multiple entries with the same key.

Example 8.4: *The following table shows a series of operations and their effects on an initially empty priority queue P. Each element consists of an integer, which we assume to be sorted according to the natural ordering of the integers. Note that each call to* min *returns a reference to an entry in the queue, not the actual value. Although the "Priority Queue" column shows the items in sorted order, the priority queue need not store elements in this order.*

Operation	*Output*	*Priority Queue*
insert(5)	–	{5}
insert(9)	–	{5,9}
insert(2)	–	{2,5,9}
insert(7)	–	{2,5,7,9}
min()	[2]	{2,5,7,9}
removeMin()	–	{5,7,9}
size()	3	{5,7,9}
min()	[5]	{5,7,9}
removeMin()	–	{7,9}
removeMin()	–	{9}
removeMin()	–	{}
empty()	*true*	{}
removeMin()	*"error"*	{}

8.1.4   A C++ Priority Queue Interface

Before discussing specific implementations of the priority queue, we first define an informal C++ interface for a priority queue in Code Fragment 8.4. It is not a complete C++ class, just a declaration of the public functions.

```
template <typename E, typename C>        // element and comparator
class PriorityQueue {                    // priority-queue interface
public:
  int size() const;                      // number of elements
  bool isEmpty() const;                  // is the queue empty?
  void insert(const E& e);               // insert element
  const E& min() const throw(QueueEmpty); // minimum element
  void removeMin() throw(QueueEmpty);    // remove minimum
};
```

Code Fragment 8.4: An informal PriorityQueue interface (not a complete class).

Although the comparator type *C* is included as a template argument, it does not appear in the public interface. Of course, its value is relevant to any concrete implementation. Observe that the function min returns a constant reference to the element

in the queue, which means that its value may be read and copied but not modified. This is important because otherwise a user of the class might inadvertently modify the element's associated key value, and this could corrupt the integrity of the data structure. The member functions size, empty, and min are all declared to be **const**, which informs the compiler that they do not alter the contents of the queue.

An error condition occurs if either of the functions min or removeMin is called on an empty priority queue. This is signaled by throwing an exception of type QueueEmpty. Its definition is similar to others we have seen. (See Code Fragment 5.2.)

8.1.5 Sorting with a Priority Queue

Another important application of a priority queue is sorting, where we are given a collection L of n elements that can be compared according to a total order relation, and we want to rearrange them in increasing order (or at least in nondecreasing order if there are ties). The algorithm for sorting L with a priority queue Q, called PriorityQueueSort, is quite simple and consists of the following two phases:

1. In the first phase, we put the elements of L into an initially empty priority queue P through a series of n insert operations, one for each element.
2. In the second phase, we extract the elements from P in nondecreasing order by means of a series of n combinations of min and removeMin operations, putting them back into L in order.

Pseudo-code for this algorithm is given in Code Fragment 8.5. It assumes that L is given as an STL list, but the code can be adapted to other containers.

Algorithm PriorityQueueSort(L, P):

    Input: An STL list L of n elements and a priority queue, P, that compares elements using a total order relation

    Output: The sorted list L

    while !L.empty() **do**

        $e \leftarrow L$.front

        L.pop_front()   {remove an element e from the list}

        P.insert(e)   {...and it to the priority queue}

    while !P.empty() **do**

        $e \leftarrow P$.min()

        P.removeMin()   {remove the smallest element e from the queue}

        L.push_back(e)   {...and append it to the back of L}

Code Fragment 8.5: Algorithm PriorityQueueSort, which sorts an STL list L with the aid of a priority queue P.

The algorithm works correctly for any priority queue P, no matter how P is implemented. However, the running time of the algorithm is determined by the running times of operations insert, min, and removeMin, which do depend on how P is implemented. Indeed, PriorityQueueSort should be considered more a sorting "scheme" than a sorting "algorithm," because it does not specify how the priority queue P is implemented. The PriorityQueueSort scheme is the paradigm of several popular sorting algorithms, including selection-sort, insertion-sort, and heap-sort, which we discuss in this chapter.

8.1.6   The STL priority_queue Class

The C++ Standard Template Library (STL) provides an implementation of a priority queue, called priority_queue. As with the other STL classes we have seen, such as stacks and queues, the STL priority queue is an example of a container. In order to declare an object of type priority_queue, it is necessary to first include the definition file, which is called "queue." As with other STL objects, the priority queue is part of the std namespace, and hence it is necessary either to use "std::priority_queue" or to provide an appropriate "**using**" statement.

The priority_queue class is templated with three parameters: the base type of the elements, the underlying STL container in which the priority queue is stored, and the comparator object. Only the first template argument is required. The second parameter (the underlying container) defaults to the STL vector. The third parameter (the comparator) defaults to using the standard C++ less-than operator ("**<**"). The STL priority queue uses comparators in the same manner as we defined in Section 8.1.2. In particular, a comparator is a class that overrides the "()" operator in order to define a boolean function that implements the less-than operator.

The code fragment below defines two STL priority queues. The first stores integers. The second stores two-dimensional points under the left-to-right ordering (recall Section 8.1.2).

```
#include <queue>
using namespace std;                    // make std accessible
priority_queue<int> p1;                 // a priority queue of integers
                        // a priority queue of points with left-to-right order
priority_queue<Point2D, vector<Point2D>, LeftRight> p2;
```

The principal member functions of the STL priority queue are given below. Let p be declared to be an STL priority_queue, and let e denote a single object whose type is the same as the base type of the priority queue. (For example, p is a priority queue of integers, and e is an integer.)

> size(): Return the number of elements in the priority queue.
>
> empty(): Return true if the priority queue is empty and false otherwise.
>
> push(e): Insert e in the priority queue.
>
> top(): Return a constant reference to the largest element of the priority queue.
>
> pop(): Remove the element at the top of the priority queue.

Other than the differences in function names, the most significant difference between our interface and the STL priority queue is that the functions top and pop access the *largest* item in the queue according to priority order, rather than the smallest. An example of the usage of the STL priority queue is shown in Code Fragment 8.6.

```
priority_queue<Point2D, vector<Point2D>, LeftRight> p2;
p2.push( Point2D(8.5, 4.6) );               // add three points to p2
p2.push( Point2D(1.3, 5.7) );
p2.push( Point2D(2.5, 0.6) );
cout << p2.top() << endl; p2.pop();         // output: (8.5, 4.6)
cout << p2.top() << endl; p2.pop();         // output: (2.5, 0.6)
cout << p2.top() << endl; p2.pop();         // output: (1.3, 5.7)
```

Code Fragment 8.6: An example of the use of the STL priority queue.

Of course, it is possible to simulate the same behavior as our priority queue by defining the comparator object so that it implements the greater-than relation rather than the less-than relation. This effectively reverses all order relations, and thus the top function would instead return the smallest element, just as function min does in our interface. Note that the STL priority queue does not perform any error checking.

8.2 Implementing a Priority Queue with a List

In this section, we show how to implement a priority queue by storing its elements in an STL list. (Recall this data structure from Section 6.2.4.) We consider two realizations, depending on whether we sort the elements of the list.

Implementation with an Unsorted List

Let us first consider the implementation of a priority queue P by an unsorted doubly linked list L. A simple way to perform the operation insert(e) on P is by adding each new element at the end of L by executing the function L.push_back(e). This implementation of insert takes $O(1)$ time.

Since the insertion does not consider key values, the resulting list L is unsorted. As a consequence, in order to perform either of the operations min or removeMin on P, we must inspect all the entries of the list to find one with the minimum key value. Thus, functions min and removeMin take $O(n)$ time each, where n is the number of elements in P at the time the function is executed. Moreover, each of these functions runs in time proportional to n even in the best case, since they each require searching the entire list to find the smallest element. Using the notation of Section 4.2.3, we can say that these functions run in $\Theta(n)$ time. We implement functions size and empty by simply returning the output of the corresponding functions executed on list L. Thus, by using an unsorted list to implement a priority queue, we achieve constant-time insertion, but linear-time search and removal.

Implementation with a Sorted List

An alternative implementation of a priority queue P also uses a list L, except that this time let us store the elements sorted by their key values. Specifically, we represent the priority queue P by using a list L of elements sorted by nondecreasing key values, which means that the first element of L has the smallest key.

We can implement function min in this case by accessing the element associated with the first element of the list with the begin function of L. Likewise, we can implement the removeMin function of P as L.pop_front(). Assuming that L is implemented as a doubly linked list, operations min and removeMin in P take $O(1)$ time, so are quite efficient.

This benefit comes at a cost, however, for now function insert of P requires that we scan through the list L to find the appropriate position in which to insert the new entry. Thus, implementing the insert function of P now takes $O(n)$ time, where n is the number of entries in P at the time the function is executed. In summary, when using a sorted list to implement a priority queue, insertion runs in linear time whereas finding and removing the minimum can be done in constant time.

Table 8.1 compares the running times of the functions of a priority queue realized by means of an unsorted and sorted list, respectively. There is an interesting contrast between the two functions. An unsorted list allows for fast insertions but slow queries and deletions, while a sorted list allows for fast queries and deletions, but slow insertions.

Operation	Unsorted List	Sorted List
size, empty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

Table 8.1: Worst-case running times of the functions of a priority queue of size n, realized by means of an unsorted or sorted list, respectively. We assume that the list is implemented by a doubly linked list. The space requirement is $O(n)$.

8.2.1 A C++ Priority Queue Implementation using a List

In Code Fragments 8.7 through 8.10, we present a priority queue implementation that stores the elements in a sorted list. The list is implemented using an STL list object (see Section 6.3.2), but any implementation of the list ADT would suffice.

In Code Fragment 8.7, we present the class definition for our priority queue. The public part of the class is essentially the same as the interface that was presented earlier in Code Fragment 8.4. In order to keep the code as simple as possible, we have omitted error checking. The class's data members consists of a list, which holds the priority queue's contents, and an instance of the comparator object, which we call isLess.

```
template <typename E, typename C>
class ListPriorityQueue {
public:
    int size() const;                      // number of elements
    bool empty() const;                    // is the queue empty?
    void insert(const E& e);               // insert element
    const E& min() const;                  // minimum element
    void removeMin();                      // remove minimum
private:
    std::list<E> L;                        // priority queue contents
    C isLess;                              // less-than comparator
};
```

Code Fragment 8.7: The class definition for a priority queue based on an STL list.

We have not bothered to give an explicit constructor for our class, relying instead on the default constructor. The default constructor for the STL list produces an empty list, which is exactly what we want.

Next, in Code Fragment 8.8, we present the implementations of the simple member functions size and empty. Recall that, when dealing with templated classes, it is necessary to repeat the full template specifications when defining member functions outside the class. Each of these functions simply invokes the corresponding function for the STL list.

```
template <typename E, typename C>      // number of elements
int ListPriorityQueue<E,C>::size() const
    { return L.size(); }


template <typename E, typename C>      // is the queue empty?
bool ListPriorityQueue<E,C>::empty() const
    { return L.empty(); }
```

Code Fragment 8.8: Implementations of the functions size and empty.

Let us now consider how to insert an element *e* into our priority queue. We define *p* to be an iterator for the list. Our approach is to walk through the list until we first find an element whose key value is larger than *e*'s, and then we insert *e* just prior to *p*. Recall that *∗p* accesses the element referenced by *p*, and ++*p* advances *p* to the next element of the list. We stop the search either when we reach the end of the list or when we first encounter a larger element, that is, one satisfying isLess(*e, ∗p*). On reaching such an entry, we insert *e* just prior to it, by invoking the STL list function insert. The code is shown in Code Fragment 8.9.

```
template <typename E, typename C>           // insert element
void ListPriorityQueue<E,C>::insert(const E& e) {
  typename std::list<E>::iterator p;
  p = L.begin();
  while (p != L.end() && !isLess(e, *p)) ++p;   // find larger element
  L.insert(p, e);                             // insert e before p
}
```

Code Fragment 8.9: Implementation of the priority queue function insert.

Consider how the above function behaves when *e* has a key value larger than any in the queue. In such a case, the while loop exits under the condition that *p* is equal to *L*.end(). Recall that *L*.end() refers to an imaginary element that lies just beyond the end of the list. Thus, by inserting before this element, we effectively append *e* to the back of the list, as desired.

You might notice the use of the keyword "**typename**" in the declaration of the iterator *p*. This is due to a subtle issue in C++ involving ***dependent names***, which arises when processing name bindings within templated objects in C++. We do not delve into the intricacies of this issue. For now, it suffices to remember to simply

Caution

include the keyword **typename** when using a template parameter (in this case *E*) to define another type.

Finally, let us consider the operations min and removeMin. Since the list is sorted in ascending order by key values, in order to implement min, we simply return a reference to the front of the list. To implement removeMin, we remove the front element of the list. The implementations are given in Code Fragment 8.10.

```
template <typename E, typename C>           // minimum element
const E& ListPriorityQueue<E,C>::min() const
  { return L.front(); }                     // minimum is at the front


template <typename E, typename C>           // remove minimum
void ListPriorityQueue<E,C>::removeMin()
  { L.pop_front(); }
```

Code Fragment 8.10: Implementations of the priority queue functions min and removeMin.

8.2.2 Selection-Sort and Insertion-Sort

Recall the PriorityQueueSort scheme introduced in Section 8.1.5. We are given an unsorted list L containing n elements, which we sort using a priority queue P in two phases. In the first phase, we insert all the elements, and in the second phase, we repeatedly remove elements using the min and removeMin operations.

Selection-Sort

If we implement the priority queue P with an unsorted list, then the first phase of PriorityQueueSort takes $O(n)$ time, since we can insert each element in constant time. In the second phase, the running time of each min and removeMin operation is proportional to the number of elements currently in P. Thus, the bottleneck computation in this implementation is the repeated "selection" of the minimum element from an unsorted list in the second phase. For this reason, this algorithm is better known as *selection-sort*. (See Figure 8.1.)

		List L	*Priority Queue P*
Input		$(7,4,8,2,5,3,9)$	$()$
Phase 1	(a)	$(4,8,2,5,3,9)$	(7)
	(b)	$(8,2,5,3,9)$	$(7,4)$
	$\vdots$	$\vdots$	$\vdots$
	(g)	$()$	$(7,4,8,2,5,3,9)$
Phase 2	(a)	(2)	$(7,4,8,5,3,9)$
	(b)	$(2,3)$	$(7,4,8,5,9)$
	(c)	$(2,3,4)$	$(7,8,5,9)$
	(d)	$(2,3,4,5)$	$(7,8,9)$
	(e)	$(2,3,4,5,7)$	$(8,9)$
	(f)	$(2,3,4,5,7,8)$	(9)
	(g)	$(2,3,4,5,7,8,9)$	$()$

Figure 8.1: Execution of selection-sort on list $L = (7,4,8,2,5,3,9)$.

As noted above, the bottleneck is the second phase, where we repeatedly remove an element with smallest key from the priority queue P. The size of P starts at n and decreases to 0 with each removeMin. Thus, the first removeMin operation takes time $O(n)$, the second one takes time $O(n-1)$, and so on. Therefore, the total time needed for the second phase is

$$O(n + (n-1) + \cdots + 2 + 1) = O\left(\sum_{i=1}^{n} i\right).$$

By Proposition 4.3, we have $\sum_{i=1}^{n} i = n(n+1)/2$. Thus, phase two takes $O(n^2)$ time, as does the entire selection-sort algorithm.

Insertion-Sort

If we implement the priority queue P using a sorted list, then we improve the running time of the second phase to $O(n)$, because each operation min and removeMin on P now takes $O(1)$ time. Unfortunately, the first phase now becomes the bottleneck for the running time, since, in the worst case, each insert operation takes time proportional to the size of P. This sorting algorithm is therefore better known as **insertion-sort** (see Figure 8.2), for the bottleneck in this sorting algorithm involves the repeated "insertion" of a new element at the appropriate position in a sorted list.

		List L	*Priority Queue P*
Input		$(7,4,8,2,5,3,9)$	$()$
Phase 1	(a)	$(4,8,2,5,3,9)$	(7)
	(b)	$(8,2,5,3,9)$	$(4,7)$
	(c)	$(2,5,3,9)$	$(4,7,8)$
	(d)	$(5,3,9)$	$(2,4,7,8)$
	(e)	$(3,9)$	$(2,4,5,7,8)$
	(f)	(9)	$(2,3,4,5,7,8)$
	(g)	$()$	$(2,3,4,5,7,8,9)$
Phase 2	(a)	(2)	$(3,4,5,7,8,9)$
	(b)	$(2,3)$	$(4,5,7,8,9)$
	⋮	⋮	⋮
	(g)	$(2,3,4,5,7,8,9)$	$()$

Figure 8.2: Execution of insertion-sort on list $L = (7,4,8,2,5,3,9)$. In Phase 1, we repeatedly remove the first element of L and insert it into P, by scanning the list implementing P until we find the correct place for this element. In Phase 2, we repeatedly perform removeMin operations on P, each of which returns the first element of the list implementing P, and we add the element at the end of L.

Analyzing the running time of Phase 1 of insertion-sort, we note that

$$O(1+2+\ldots+(n-1)+n) = O\left(\textstyle\sum_{i=1}^{n} i\right).$$

Again, by recalling Proposition 4.3, the first phase runs in $O(n^2)$ time; hence, so does the entire algorithm.

Alternately, we could change our definition of insertion-sort so that we insert elements starting from the end of the priority-queue sequence in the first phase, in which case performing insertion-sort on a list that is already sorted would run in $O(n)$ time. Indeed, the running time of insertion-sort is $O(n+I)$ in this case, where I is the number of **inversions** in the input list, that is, the number of pairs of elements that start out in the input list in the wrong relative order.

8.3 Heaps

The two implementations of the PriorityQueueSort scheme presented in the previous section suggest a possible way of improving the running time for priority-queue sorting. One algorithm (selection-sort) achieves a fast running time for the first phase, but has a slow second phase, whereas the other algorithm (insertion-sort) has a slow first phase, but achieves a fast running time for the second phase. If we could somehow balance the running times of the two phases, we might be able to significantly speed up the overall running time for sorting. This approach is, in fact, exactly what we can achieve using the priority-queue implementation discussed in this section.

An efficient realization of a priority queue uses a data structure called a ***heap***. This data structure allows us to perform both insertions and removals in logarithmic time, which is a significant improvement over the list-based implementations discussed in Section 8.2. The fundamental way the heap achieves this improvement is to abandon the idea of storing elements and keys in a list and take the approach of storing elements and keys in a binary tree instead.

8.3.1 The Heap Data Structure

A heap (see Figure 8.3) is a binary tree T that stores a collection of elements with their associated keys at its nodes and that satisfies two additional properties: a relational property, defined in terms of the way keys are stored in T, and a structural property, defined in terms of the nodes of T itself. We assume that a total order relation on the keys is given, for example, by a comparator.

The relational property of T, defined in terms of the way keys are stored, is the following:

Heap-Order Property: In a heap T, for every node v other than the root, the key associated with v is greater than or equal to the key associated with v's parent.

As a consequence of the heap-order property, the keys encountered on a path from the root to an external node of T are in nondecreasing order. Also, a minimum key is always stored at the root of T. This is the most important key and is informally said to be "at the top of the heap," hence, the name "heap" for the data structure. By the way, the heap data structure defined here has nothing to do with the free-store memory heap (Section 14.1.1) used in the run-time environment supporting programming languages like C++.

You might wonder why heaps are defined with the smallest key at the top, rather than the largest. The distinction is arbitrary. (This is evidenced by the fact that the STL priority queue does exactly the opposite.) Recall that a comparator
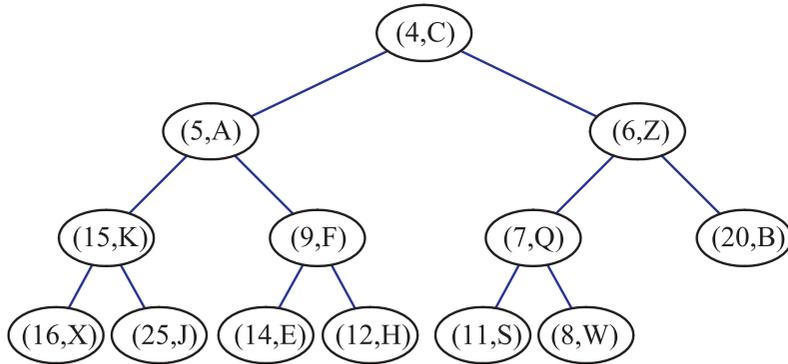
Figure 8.3: Example of a heap storing 13 elements. Each element is a key-value pair of the form (k, v). The heap is ordered based on the key value, k, of each element.

implements the less-than operator between two keys. Suppose that we had instead defined our comparator to indicate the ***opposite*** of the standard total order relation between keys (so that, for example, isLess(x, y) would return true if x were ***greater than*** y). Then the root of the resulting heap would store the largest key. This versatility comes essentially for free from our use of the comparator pattern. By defining the minimum key in terms of the comparator, the "minimum" key with a "reverse" comparator is in fact the largest. Thus, without loss of generality, we assume that we are always interested in the minimum key, which is always at the root of the heap.

Caution

For the sake of efficiency, which becomes clear later, we want the heap T to have as small a height as possible. We enforce this desire by insisting that the heap T satisfy an additional structural property, it must be ***complete***. Before we define this structural property, we need some definitions. We recall from Section 7.3.3 that level i of a binary tree T is the set of nodes of T that have depth i. Given nodes v and w on the same level of T, we say that v is ***to the left of*** w if v is encountered before w in an inorder traversal of T. That is, there is a node u of T such that v is in the left subtree of u and w is in the right subtree of u. For example, in the binary tree of Figure 8.3, the node storing entry $(15, K)$ is to the left of the node storing entry $(7, Q)$. In a standard drawing of a binary tree, the "to the left of" relation is visualized by the relative horizontal placement of the nodes.

Complete Binary Tree Property: A heap T with height h is a ***complete*** binary tree, that is, levels $0, 1, 2, \ldots, h - 1$ of T have the maximum number of nodes possible (namely, level i has 2^i nodes, for $0 \le i \le h - 1$) and the nodes at level h fill this level from left to right.

The Height of a Heap

Let h denote the height of T. Another way of defining the last node of T is that it is the node on level h such that all the other nodes of level h are to the left of it. Insisting that T be complete also has an important consequence as shown in Proposition 8.5.

Proposition 8.5: *A heap T storing n entries has height*

$$h = \lfloor \log n \rfloor.$$

Justification: From the fact that T is complete, we know that there are 2^i nodes in level, i for $0 \le i \le h-1$, and level h has at least 1 node. Thus, the number of nodes of T is at least

$$(1+2+4+\cdots+2^{h-1})+1 \quad = \quad (2^h - 1)+1$$
$$= \quad 2^h.$$

Level h has at most 2^h nodes, and thus the number of nodes of T is at most

$$(1+2+4+\cdots+2^{h-1})+2^h \quad = \quad 2^{h+1}-1.$$

Since the number of nodes is equal to the number n of entries, we obtain

$$2^h \le n$$

and

$$n \le 2^{h+1}-1.$$

Thus, by taking logarithms of both sides of these two inequalities, we see that

$$h \le \log n$$

and

$$\log(n+1)-1 \le h.$$

Since h is an integer, the two inequalities above imply that

$$h = \lfloor \log n \rfloor.$$

■

Proposition 8.5 has an important consequence. It implies that if we can perform update operations on a heap in time proportional to its height, then those operations will run in logarithmic time. Therefore, let us turn to the problem of how to efficiently perform various priority queue functions using a heap.

8.3.2   Complete Binary Trees and Their Representation

Let us discuss more about complete binary trees and how they are represented.

The Complete Binary Tree ADT

As an abstract data type, a complete binary tree T supports all the functions of the binary tree ADT (Section 7.3.1), plus the following two functions:

> add(e): Add to T and return a new external node v storing element e, such that the resulting tree is a complete binary tree with last node v.
>
> remove(): Remove the last node of T and return its element.

By using only these update operations, the resulting tree is guaranteed to be a complete binary. As shown in Figure 8.4, there are essentially two cases for the effect of an add (and remove is similar).

- If the bottom level of T is not full, then add inserts a new node on the bottom level of T, immediately after the rightmost node of this level (that is, the last node); hence, T's height remains the same.
- If the bottom level is full, then add inserts a new node as the left child of the leftmost node of the bottom level of T; hence, T's height increases by one.
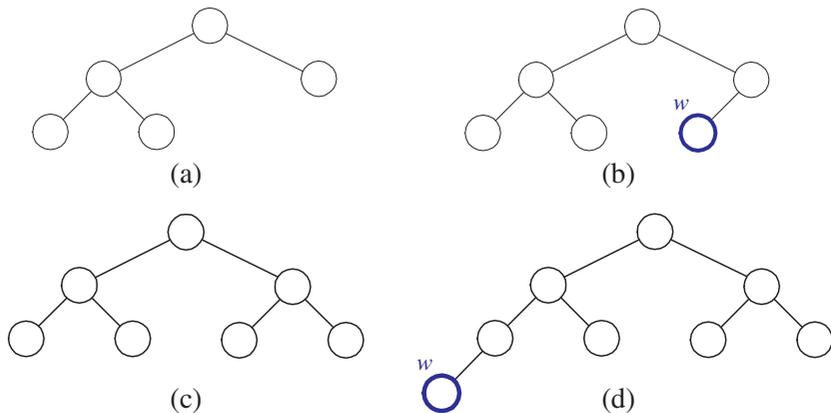
Figure 8.4: Examples of operations add and remove on a complete binary tree, where w denotes the node inserted by add or deleted by remove. The trees shown in (b) and (d) are the results of performing add operations on the trees in (a) and (c), respectively. Likewise, the trees shown in (a) and (c) are the results of performing remove operations on the trees in (b) and (d), respectively.

A Vector Representation of a Complete Binary Tree

The vector-based binary tree representation (recall Section 7.3.5) is especially suitable for a complete binary tree T. We recall that in this implementation, the nodes of T are stored in a vector A such that node v in T is the element of A with index equal to the level number $f(v)$ defined as follows:

- If v is the root of T, then $f(v) = 1$

- If v is the left child of node u, then $f(v) = 2f(u)$

- If v is the right child of node u, then $f(v) = 2f(u) + 1$

With this implementation, the nodes of T have contiguous indices in the range $[1, n]$ and the last node of T is always at index n, where n is the number of nodes of T. Figure 8.5 shows two examples illustrating this property of the last node.
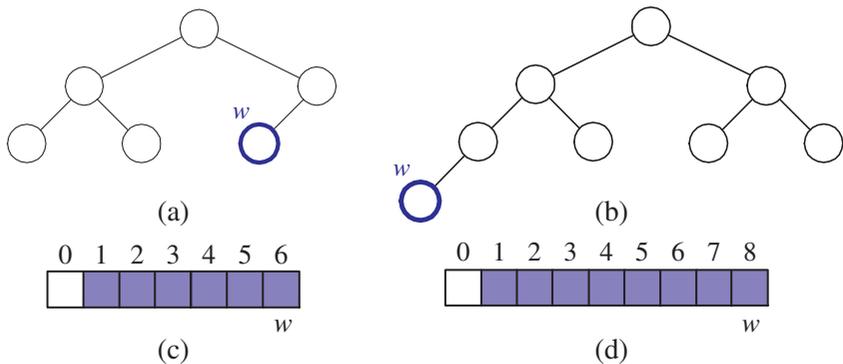
Figure 8.5: Two examples showing that the last node w of a heap with n nodes has level number n: (a) heap T_1 with more than one node on the bottom level; (b) heap T_2 with one node on the bottom level; (c) vector-based representation of T_1; (d) vector-based representation of T_2.

The simplifications that come from representing a complete binary tree T with a vector aid in the implementation of functions add and remove. Assuming that no array expansion is necessary, functions add and remove can be performed in $O(1)$ time because they simply involve adding or removing the last element of the vector. Moreover, the vector associated with T has $n+1$ elements (the element at index 0 is a placeholder). If we use an extendable array that grows and shrinks for the implementation of the vector (for example, the STL vector class), the space used by the vector-based representation of a complete binary tree with n nodes is $O(n)$ and operations add and remove take $O(1)$ amortized time.

A C++ Implementation of a Complete Binary Tree

We present the complete binary tree ADT as an informal interface, called CompleteTree, in Code Fragment 8.11. As with our other informal interfaces, this is not a complete C++ class. It just gives the public portion of the class.

The interface defines a nested class, called Position, which represents a node of the tree. We provide the necessary functions to access the root and last positions and to navigate through the tree. The modifier functions add and remove are provided, along with a function swap, which swaps the contents of two given nodes.

```cpp
template <typename E>
class CompleteTree {                        // left-complete tree interface
public:                                     // publicly accessible types
  class Position;                           // node position type
  int size() const;                         // number of elements
  Position left(const Position& p);         // get left child
  Position right(const Position& p);        // get right child
  Position parent(const Position& p);       // get parent
  bool hasLeft(const Position& p) const;    // does node have left child?
  bool hasRight(const Position& p) const;   // does node have right child?
  bool isRoot(const Position& p) const;     // is this the root?
  Position root();                          // get root position
  Position last();                          // get last node
  void addLast(const E& e);                 // add a new last node
  void removeLast();                        // remove the last node
  void swap(const Position& p, const Position& q); // swap node contents
};
```

Code Fragment 8.11: Interface CompleteBinaryTree for a complete binary tree.

In order to implement this interface, we store the elements in an STL vector, called V. We implement a tree position as an iterator to this vector. To convert from the index representation of a node to this positional representation, we provide a function pos. The reverse conversion is provided by function idx. This portion of the class definition is given in Code Fragment 8.12.

```cpp
private:                                    // member data
  std::vector<E> V;                         // tree contents
public:                                     // publicly accessible types
  typedef typename std::vector<E>::iterator Position; // a position in the tree
protected:                                  // protected utility functions
  Position pos(int i)                       // map an index to a position
    { return V.begin() + i; }
  int idx(const Position& p) const          // map a position to an index
    { return p − V.begin(); }
```

Code Fragment 8.12: Member data and private utilities for a complete tree class.

Given the index of a node i, the function pos maps it to a position by adding i to V.begin(). Here we are exploiting the fact that the STL vector supports a **random-access iterator** (recall Section 6.2.5). In particular, given an integer i, the expression V.begin() $+ i$ yields the position of the ith element of the vector, and, given a position p, the expression $p - V$.begin() yields the index of position p.

We present a full implementation of a vector-based complete tree ADT in Code Fragment 8.13. Because the class consists of a large number of small one-line functions, we have chosen to violate our normal coding conventions by placing all the function definitions inside the class definition.

```
template <typename E>
class VectorCompleteTree {
  //... insert private member data and protected utilities here
public:
  VectorCompleteTree() : V(1) {}                          // constructor
  int size() const                          { return V.size() − 1; }
  Position left(const Position& p)          { return pos(2*idx(p)); }
  Position right(const Position& p)         { return pos(2*idx(p) + 1); }
  Position parent(const Position& p)        { return pos(idx(p)/2); }
  bool hasLeft(const Position& p) const     { return 2*idx(p) <= size(); }
  bool hasRight(const Position& p) const    { return 2*idx(p) + 1 <= size(); }
  bool isRoot(const Position& p) const      { return idx(p) == 1; }
  Position root()                           { return pos(1); }
  Position last()                           { return pos(size()); }
  void addLast(const E& e)                  { V.push_back(e); }
  void removeLast()                         { V.pop_back(); }
  void swap(const Position& p, const Position& q)
                                            { E e = *q; *q = *p; *p = e; }
};
```

Code Fragment 8.13: A vector-based implementation of the complete tree ADT.

Recall from Section 7.3.5 that the root node is at index 1 of the vector. Since STL vectors are indexed starting at 0, our constructor creates the initial vector with one element. This element at index 0 is never used. As a consequence, the size of the priority queue is one less than the size of the vector.

Recall from Section 7.3.5 that, given a node at index i, its left and right children are located at indices $2i$ and $2i+1$, respectively. Its parent is located at index $\lfloor i/2 \rfloor$. Given a position p, the functions left, right, and parent first convert p to an index using the utility idx, which is followed by the appropriate arithmetic operation on this index, and finally they convert the index back to a position using the utility pos.

We determine whether a node has a child by evaluating the index of this child and testing whether the node at that index exists in the vector. Operations add and remove are implemented by adding or removing the last entry of the vector, respectively.

8.3.3 Implementing a Priority Queue with a Heap

We now discuss how to implement a priority queue using a heap. Our heap-based representation for a priority queue P consists of the following (see Figure 8.6):

- **heap**: A complete binary tree T whose nodes store the elements of the queue and whose keys satisfy the heap-order property. We assume the binary tree T is implemented using a vector, as described in Section 8.3.2. For each node v of T, we denote the associated key by $k(v)$.
- **comp**: A comparator that defines the total order relation among the keys.
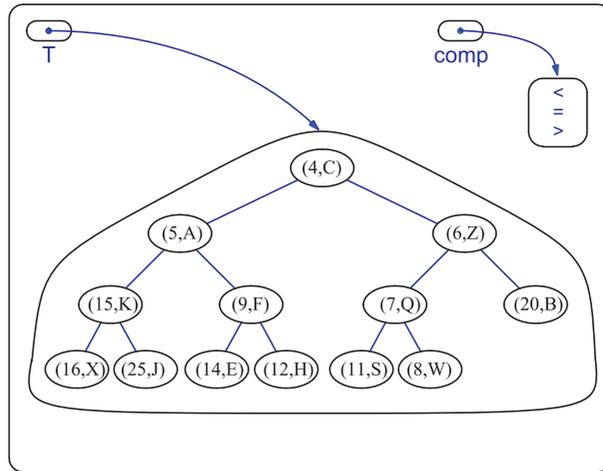
Figure 8.6: Illustration of the heap-based implementation of a priority queue.

With this data structure, functions size and empty take $O(1)$ time, as usual. In addition, function min can also be easily performed in $O(1)$ time by accessing the entry stored at the root of the heap (which is at index 1 in the vector).

Insertion

Let us consider how to perform insert on a priority queue implemented with a heap T. To store a new element e in T, we add a new node z to T with operation add, so that this new node becomes the last node of T, and then store e in this node.

After this action, the tree T is complete, but it may violate the heap-order property. Hence, unless node z is the root of T (that is, the priority queue was empty before the insertion), we compare key $k(z)$ with the key $k(u)$ stored at the parent u of z. If $k(z) \geq k(u)$, the heap-order property is satisfied and the algorithm terminates. If instead $k(z) < k(u)$, then we need to restore the heap-order property, which can be locally achieved by swapping the entries stored at z and u. (See Figures 8.7(c) and (d).) This swap causes the new entry (k,e) to move up one level. Again, the heap-order property may be violated, and we continue swapping, going
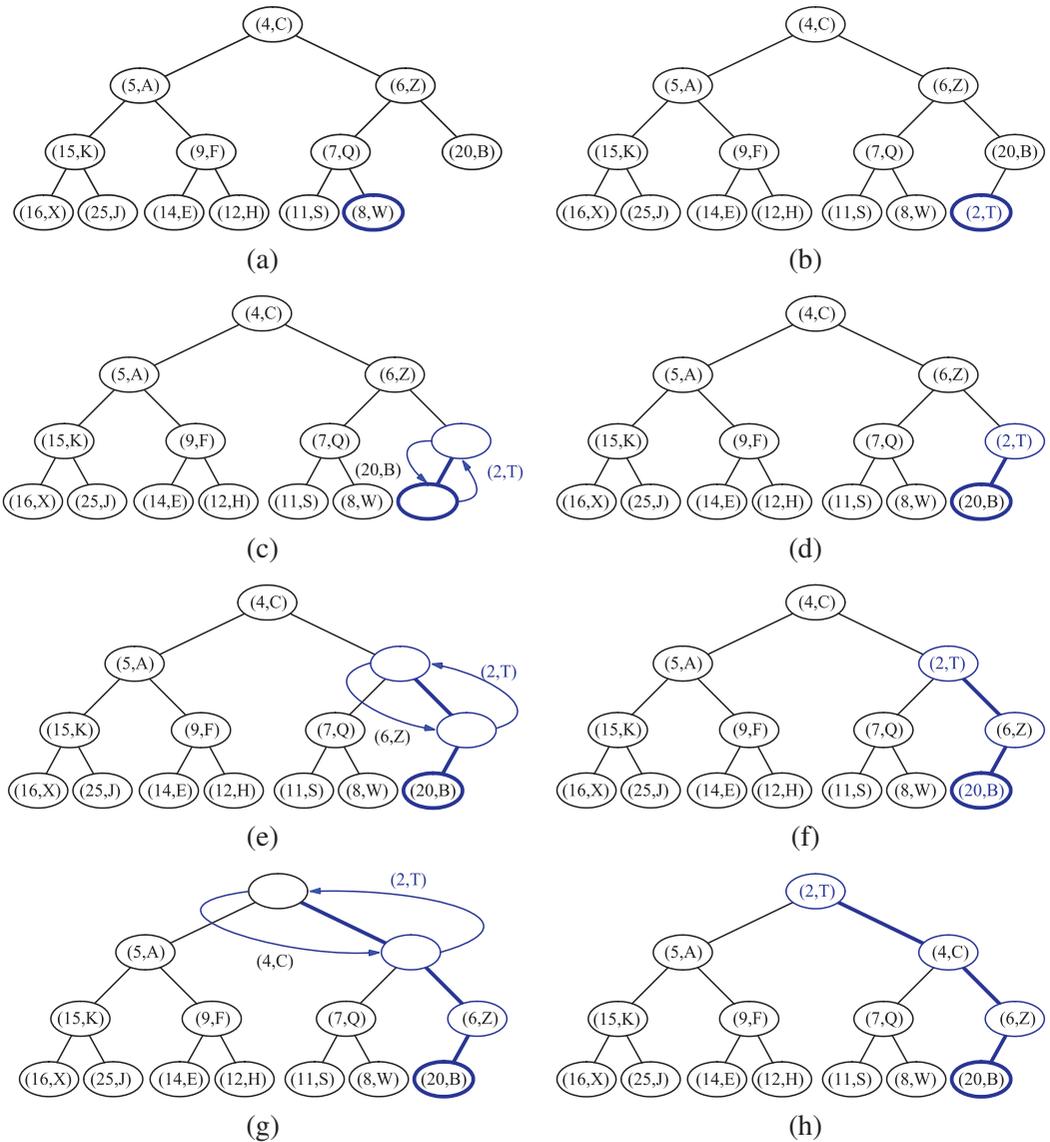
Figure 8.7: Insertion of a new entry with key 2 into the heap of Figure 8.6: (a) initial heap; (b) after performing operation add; (c) and (d) swap to locally restore the partial order property; (e) and (f) another swap; (g) and (h) final swap.

up in T until no violation of the heap-order property occurs. (See Figures 8.7(e) and (h).)

The upward movement of the newly inserted entry by means of swaps is conventionally called ***up-heap bubbling***. A swap either resolves the violation of the heap-order property or propagates it one level up in the heap. In the worst case, up-heap bubbling causes the new entry to move all the way up to the root of heap T. (See Figure 8.7.) Thus, in the worst case, the number of swaps performed in the execution of function insert is equal to the height of T, that is, it is $\lfloor \log n \rfloor$ by Proposition 8.5.

Removal

Let us now turn to function removeMin of the priority queue ADT. The algorithm for performing function removeMin using heap T is illustrated in Figure 8.8.

We know that an element with the smallest key is stored at the root r of T (even if there is more than one entry with the smallest key). However, unless r is the only node of T, we cannot simply delete node r, because this action would disrupt the binary tree structure. Instead, we access the last node w of T, copy its entry to the root r, and then delete the last node by performing operation remove of the complete binary tree ADT. (See Figure 8.8(a) and (b).)

Down-Heap Bubbling after a Removal

We are not necessarily done, however, for, even though T is now complete, T may now violate the heap-order property. If T has only one node (the root), then the heap-order property is trivially satisfied and the algorithm terminates. Otherwise, we distinguish two cases, where r denotes the root of T:

- If r has no right child, let s be the left child of r
- Otherwise (r has both children), let s be a child of r with the smaller key

If $k(r) \leq k(s)$, the heap-order property is satisfied and the algorithm terminates. If instead $k(r) > k(s)$, then we need to restore the heap-order property, which can be locally achieved by swapping the entries stored at r and s. (See Figure 8.8(c) and (d).) (Note that we shouldn't swap r with s's sibling.) The swap we perform restores the heap-order property for node r and its children, but it may violate this property at s; hence, we may have to continue swapping down T until no violation of the heap-order property occurs. (See Figure 8.8(e) and (h).)

This downward swapping process is called ***down-heap bubbling***. A swap either resolves the violation of the heap-order property or propagates it one level down in the heap. In the worst case, an entry moves all the way down to the bottom level. (See Figure 8.8.) Thus, the number of swaps performed in the execution of function removeMin is, in the worst case, equal to the height of heap T, that is, it is $\lfloor \log n \rfloor$ by Proposition 8.5
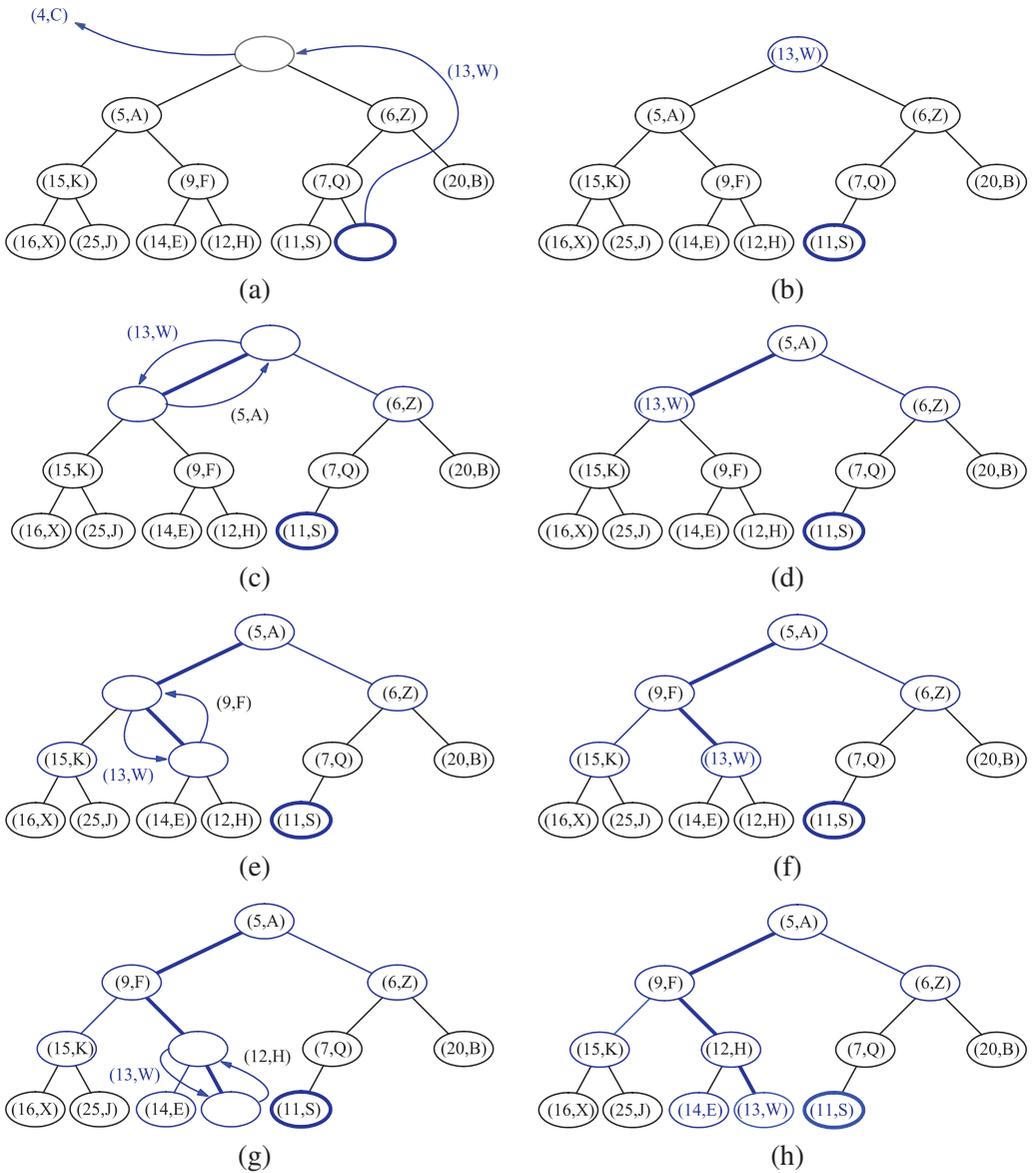
Figure 8.8: Removing the element with the smallest key from a heap: (a) and (b) deletion of the last node, whose element is moved to the root; (c) and (d) swap to locally restore the heap-order property; (e) and (f) another swap; (g) and (h) final swap.

Analysis

Table 8.2 shows the running time of the priority queue ADT functions for the heap implementation of a priority queue, assuming that two keys can be compared in $O(1)$ time and that the heap T is implemented with either a vector or linked structure.

Operation	*Time*
size, empty	$O(1)$
min	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

Table 8.2: Performance of a priority queue realized by means of a heap, which is in turn implemented with a vector or linked structure. We denote with n the number of entries in the priority queue at the time a method is executed. The space requirement is $O(n)$. The running time of operations insert and removeMin is worst case for the array-list implementation of the heap and amortized for the linked representation.

In short, each of the priority queue ADT functions can be performed in $O(1)$ time or in $O(\log n)$ time, where n is the number of elements at the time the function is executed. This analysis is based on the following:

- The heap T has n nodes, each storing a reference to an entry
- Operations add and remove on T take either $O(1)$ amortized time (vector representation) or $O(\log n)$ worst-case time
- In the worst case, up-heap and down-heap bubbling perform a number of swaps equal to the height of T
- The height of heap T is $O(\log n)$, since T is complete (Proposition 8.5)

Thus, if heap T is implemented with the linked structure for binary trees, the space needed is $O(n)$. If we use a vector-based implementation for T instead, then the space is proportional to the size N of the array used for the vector representing T.

We conclude that the heap data structure is a very efficient realization of the priority queue ADT, independent of whether the heap is implemented with a linked structure or a vector. The heap-based implementation achieves fast running times for both insertion and removal, unlike the list-based priority queue implementations. Indeed, an important consequence of the efficiency of the heap-based implementation is that it can speed up priority-queue sorting to be much faster than the list-based insertion-sort and selection-sort algorithms.

8.3.4 C++ Implementation

In this section, we present a heap-based priority queue implementation. The heap is implemented using the vector-based complete tree implementation, which we presented in Section 8.3.2.

In Code Fragment 8.7, we present the class definition. The public part of the class is essentially the same as the interface, but, in order to keep the code simple, we have ignored error checking. The class's data members consists of the complete tree, named T, and an instance of the comparator object, named *isLess*. We have also provided a type definition for a node position in the tree, called Position.

```
template <typename E, typename C>
class HeapPriorityQueue {
public:
  int size() const;                       // number of elements
  bool empty() const;                     // is the queue empty?
  void insert(const E& e);                // insert element
  const E& min();                         // minimum element
  void removeMin();                       // remove minimum
private:
  VectorCompleteTree<E> T;                // priority queue contents
  C isLess;                               // less-than comparator
                                          // shortcut for tree position
  typedef typename VectorCompleteTree<E>::Position Position;
};
```

Code Fragment 8.14: A heap-based implementation of a priority queue.

In Code Fragment 8.15, we present implementations of the simple member functions size, empty, and min. The function min returns a reference to the root's element through the use of the "*" operator, which is provided by the Position class of VectorCompleteTree.

```
template <typename E, typename C>      // number of elements
int HeapPriorityQueue<E,C>::size() const
  { return T.size(); }

template <typename E, typename C>      // is the queue empty?
bool HeapPriorityQueue<E,C>::empty() const
  { return size() == 0; }

template <typename E, typename C>      // minimum element
const E& HeapPriorityQueue<E,C>::min()
  { return *(T.root()); }                // return reference to root element
```

Code Fragment 8.15: The member functions size, empty, and min.

Next, in Code Fragment 8.16, we present an implementation of the insert operation. As outlined in the previous section, this works by adding the new element to the last position of the tree and then it performs up-heap bubbling by repeatedly swapping this element with its parent until its parent has a smaller key value.

```
template <typename E, typename C>        // insert element
void HeapPriorityQueue<E,C>::insert(const E& e) {
  T.addLast(e);                          // add e to heap
  Position v = T.last();                 // e's position
  while (!T.isRoot(v)) {                 // up-heap bubbling
    Position u = T.parent(v);
    if (!isLess(*v, *u)) break;          // if v in order, we're done
    T.swap(v, u);                        // ...else swap with parent
    v = u;
  }
}
```

Code Fragment 8.16: An implementation of the function insert.

Finally, let us consider the removeMin operation. If the tree has only one node, then we simply remove it. Otherwise, we swap the root's element with the last element of the tree and remove the last element. We then apply down-heap bubbling to the root. Letting *u* denote the current node, this involves determining *u*'s smaller child, which is stored in *v*. If the child's key is smaller than *u*'s, we swap *u*'s contents with this child's. The code is presented in Code Fragment 8.17.

```
template <typename E, typename C>        // remove minimum
void HeapPriorityQueue<E,C>::removeMin() {
  if (size() == 1)                       // only one node?
    T.removeLast();                      // ...remove it
  else {
    Position u = T.root();               // root position
    T.swap(u, T.last());                 // swap last with root
    T.removeLast();                      // ...and remove last
    while (T.hasLeft(u)) {               // down-heap bubbling
      Position v = T.left(u);
      if (T.hasRight(u) && isLess(*(T.right(u)), *v))
        v = T.right(u);                  // v is u's smaller child
      if (isLess(*v, *u)) {              // is u out of order?
        T.swap(u, v);                    // ...then swap
        u = v;
      }
      else break;                        // else we're done
    }
  }
}
```

Code Fragment 8.17: A heap-based implementation of a priority queue.

8.3.5 Heap-Sort

As we have previously observed, realizing a priority queue with a heap has the advantage that all the functions in the priority queue ADT run in logarithmic time or better. Hence, this realization is suitable for applications where fast running times are sought for all the priority queue functions. Therefore, let us again consider the PriorityQueueSort sorting scheme from Section 8.1.5, which uses a priority queue P to sort a list L.

During Phase 1, the i-th insert operation $(1 \le i \le n)$ takes $O(1 + \log i)$ time, since the heap has i entries after the operation is performed. Likewise, during Phase 2, the j-th removeMin operation $(1 \le j \le n)$ runs in time $O(1 + \log(n - j + 1))$, since the heap has $n - j + 1$ entries at the time the operation is performed. Thus, each phase takes $O(n \log n)$ time, so the entire priority-queue sorting algorithm runs in $O(n \log n)$ time when we use a heap to implement the priority queue. This sorting algorithm is better known as **_heap-sort_**, and its performance is summarized in the following proposition.

Proposition 8.6: *The heap-sort algorithm sorts a list L of n elements in $O(n \log n)$ time, assuming two elements of L can be compared in $O(1)$ time.*

Let us stress that the $O(n \log n)$ running time of heap-sort is considerably better than the $O(n^2)$ running time of selection-sort and insertion-sort (Section 8.2.2) and is essentially the best possible for any sorting algorithm.

Implementing Heap-Sort In-Place

If the list L to be sorted is implemented by means of an array, we can speed up heap-sort and reduce its space requirement by a constant factor using a portion of the list L itself to store the heap, thus avoiding the use of an external heap data structure. This performance is accomplished by modifying the algorithm as follows:

1. We use a reverse comparator, which corresponds to a heap where the largest element is at the top. At any time during the execution of the algorithm, we use the left portion of L, up to a certain rank $i - 1$, to store the elements in the heap, and the right portion of L, from rank i to $n - 1$ to store the elements in the list. Thus, the first i elements of L (at ranks $0, \ldots, i - 1$) provide the vector representation of the heap (with modified level numbers starting at 0 instead of 1), that is, the element at rank k is greater than or equal to its "children" at ranks $2k + 1$ and $2k + 2$.
2. In the first phase of the algorithm, we start with an empty heap and move the boundary between the heap and the list from left to right, one step at a time. In step i $(i = 1, \ldots, n)$, we expand the heap by adding the element at rank $i - 1$ and perform up-heap bubbling.

3. In the second phase of the algorithm, we start with an empty list and move the boundary between the heap and the list from right to left, one step at a time. At step i ($i = 1, \ldots, n$), we remove a maximum element from the heap and store it at rank $n - i$.

The above variation of heap-sort is said to be ***in-place***, since we use only a constant amount of space in addition to the list itself. Instead of transferring elements out of the list and then back in, we simply rearrange them. We illustrate in-place heap-sort in Figure 8.9. In general, we say that a sorting algorithm is in-place if it uses only a constant amount of memory in addition to the memory needed for the objects being sorted themselves. A sorting algorithm is considered space-efficient if it can be implemented in-place.
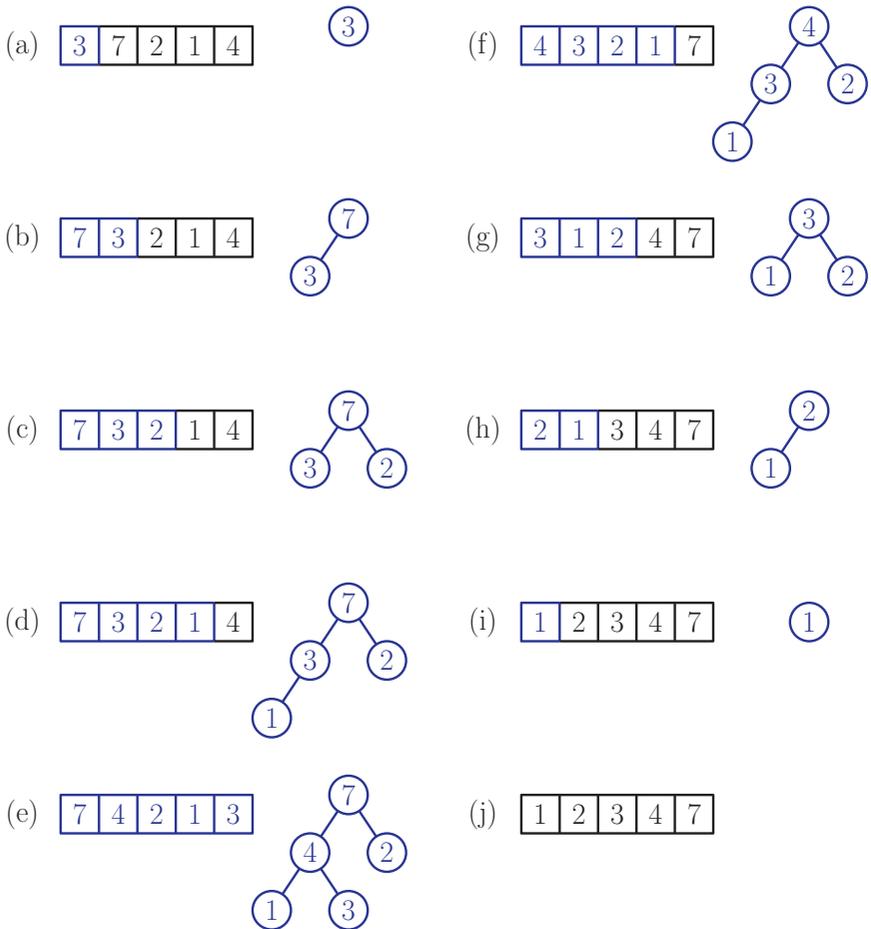
Figure 8.9: In-place heap-sort. Parts (a) through (e) show the addition of elements to the heap; (f) through (j) show the removal of successive elements. The portions of the array that are used for the heap structure are shown in blue.

8.3.6 Bottom-Up Heap Construction ⋆

The analysis of the heap-sort algorithm shows that we can construct a heap storing n elements in $O(n \log n)$ time, by means of n successive insert operations, and then use that heap to extract the elements in order. However, if all the elements to be stored in the heap are given in advance, there is an alternative ***bottom-up*** construction function that runs in $O(n)$ time. We describe this function in this section, observing that it can be included as one of the constructors in a Heap class instead of filling a heap using a series of n insert operations. For simplicity, we describe this bottom-up heap construction assuming the number n of keys is an integer of the type $n = 2^h - 1$. That is, the heap is a complete binary tree with every level being full, so the heap has height $h = \log(n+1)$. Viewed nonrecursively, bottom-up heap construction consists of the following $h = \log(n+1)$ steps:

1. In the first step (see Figure 8.10(a)), we construct $(n+1)/2$ elementary heaps storing one entry each.

2. In the second step (see Figure 8.10(b)–(c)), we form $(n+1)/4$ heaps, each storing three entries, by joining pairs of elementary heaps and adding a new entry. The new entry is placed at the root and may have to be swapped with the entry stored at a child to preserve the heap-order property.

3. In the third step (see Figure 8.10(d)–(e)), we form $(n+1)/8$ heaps, each storing 7 entries, by joining pairs of 3-entry heaps (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.

$\vdots$

i. In the generic *i*th step, $2 \leq i \leq h$, we form $(n+1)/2^i$ heaps, each storing $2^i - 1$ entries, by joining pairs of heaps storing $(2^{i-1} - 1)$ entries (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.

$\vdots$

$h+1$. In the last step (see Figure 8.10(f)–(g)), we form the final heap, storing all the n entries, by joining two heaps storing $(n-1)/2$ entries (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.

We illustrate bottom-up heap construction in Figure 8.10 for $h = 3$.
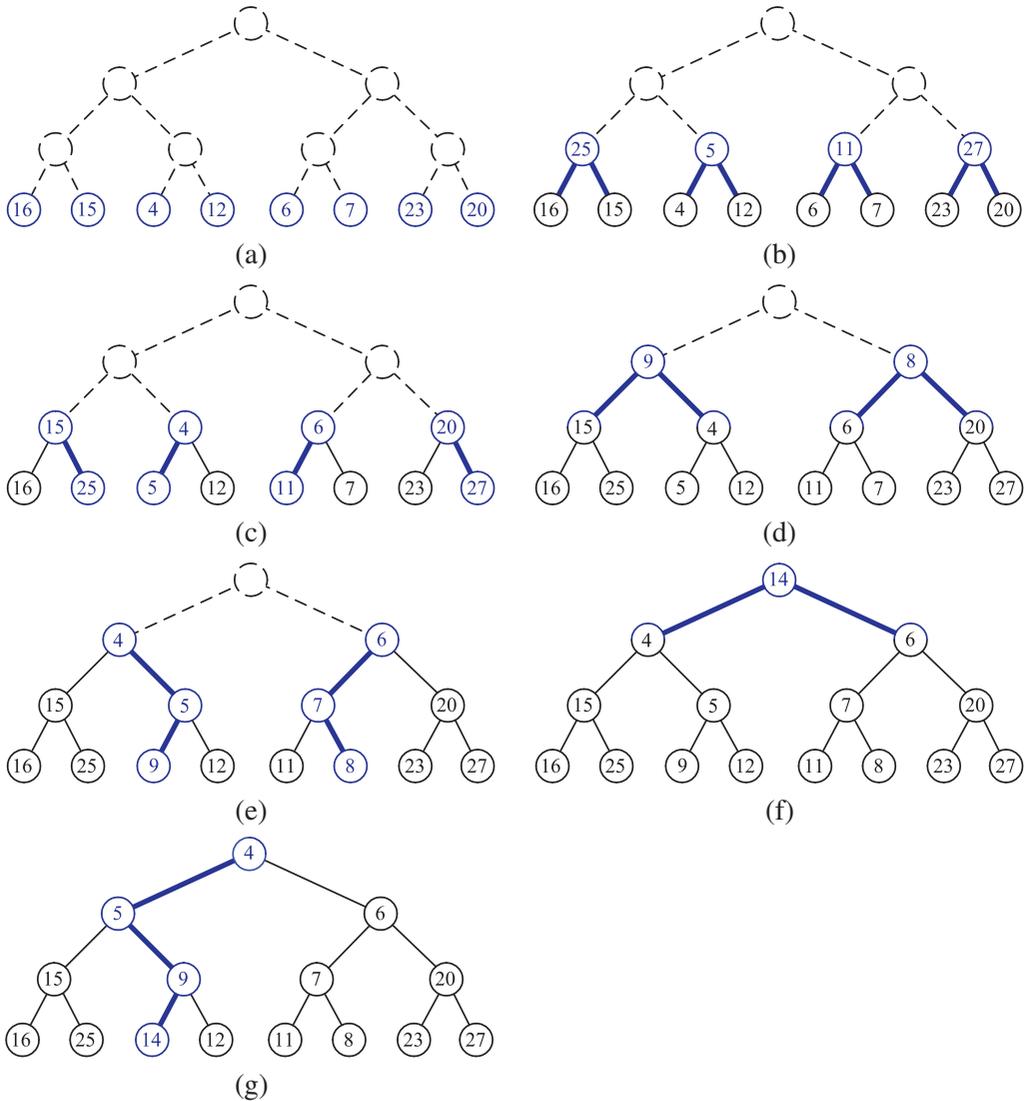
Figure 8.10: Bottom-up construction of a heap with 15 entries: (a) we begin by constructing one-entry heaps on the bottom level; (b) and (c) we combine these heaps into three-entry heaps; (d) and (e) seven-entry heaps; (f) and (g) we create the final heap. The paths of the down-heap bubblings are highlighted in blue. For simplicity, we only show the key within each node instead of the entire entry.

Recursive Bottom-Up Heap Construction

We can also describe bottom-up heap construction as a recursive algorithm, as shown in Code Fragment 8.18, which we call by passing a list storing the keys for which we wish to build a heap.

Algorithm BottomUpHeap(L):
    Input: An STL list L storing $n = 2^{h+1} - 1$ entries
    Output: A heap T storing the entries of L.
    if L.empty() **then**
        return an empty heap
    $e \leftarrow L$.front()
    L.pop_front()
    Split L into two lists, L_1 and L_2, each of size $(n-1)/2$
    $T_1 \leftarrow$ BottomUpHeap(L_1)
    $T_2 \leftarrow$ BottomUpHeap(L_2)
    Create binary tree T with root r storing e, left subtree T_1, and right subtree T_2
    Perform a down-heap bubbling from the root r of T, if necessary
    return T

Code Fragment 8.18: Recursive bottom-up heap construction.

Although the algorithm has been expressed in terms of an STL list, the construction could have been performed equally well with a vector. In such a case, the splitting of the vector is performed conceptually, by defining two ranges of indices, one representing the front half L_1 and the other representing the back half L_2.

At first glance, it may seem that there is no substantial difference between this algorithm and the incremental heap construction used in the heap-sort algorithm of Section 8.3.5. One works by down-heap bubbling and the other uses up-heap bubbling. It is somewhat surprising, therefore, that the bottom-up heap construction is actually asymptotically faster than incrementally inserting n keys into an initially empty heap. The following proposition shows this.

Proposition 8.7: *Bottom-up construction of a heap with n entries takes $O(n)$ time, assuming two keys can be compared in $O(1)$ time.*

Justification: We analyze bottom-up heap construction using a "visual" approach, which is illustrated in Figure 8.11.

Let T be the final heap, let v be a node of T, and let $T(v)$ denote the subtree of T rooted at v. In the worst case, the time for forming $T(v)$ from the two recursively formed subtrees rooted at v's children is proportional to the height of $T(v)$. The worst case occurs when down-heap bubbling from v traverses a path from v all the way to a bottommost node of $T(v)$.

Now consider the path $p(v)$ of T from node v to its inorder successor external node, that is, the path that starts at v, goes to the right child of v, and then goes down leftward until it reaches an external node. We say that path $p(v)$ is **associated with** node v. Note that $p(v)$ is not necessarily the path followed by down-heap bubbling when forming $T(v)$. Clearly, the size (number of nodes) of $p(v)$ is equal to the height of $T(v)$ plus one. Hence, forming $T(v)$ takes time proportional to the size of $p(v)$, in the worst case. Thus, the total running time of bottom-up heap construction is proportional to the sum of the sizes of the paths associated with the nodes of T.

Observe that each node v of T belongs to at most two such paths: the path $p(v)$ associated with v itself and possibly also the path $p(u)$ associated with the closest ancestor u of v preceding v in an inorder traversal. (See Figure 8.11.) In particular, the root r of T and the nodes on the leftmost root-to-leaf path each belong only to one path, the one associated with the node itself. Therefore, the sum of the sizes of the paths associated with the internal nodes of T is at most $2n - 1$. We conclude that the bottom-up construction of heap T takes $O(n)$ time. ∎
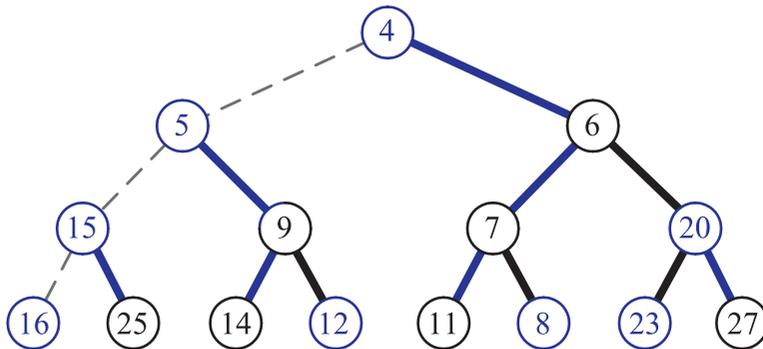
Figure 8.11: Visual justification of the linear running time of bottom-up heap construction, where the paths associated with the internal nodes have been highlighted with alternating colors. For example, the path associated with the root consists of the nodes storing keys 4, 6, 7, and 11. Also, the path associated with the right child of the root consists of the internal nodes storing keys 6, 20, and 23.

To summarize, Proposition 8.7 states that the running time for the first phase of heap-sort can be reduced to be $O(n)$. Unfortunately, the running time of the second phase of heap-sort cannot be made asymptotically better than $O(n \log n)$ (that is, it will always be $\Omega(n \log n)$ in the worst case). We do not justify this lower bound until Chapter 11, however. Instead, we conclude this chapter by discussing a design pattern that allows us to extend the priority queue ADT to have additional functionality.

8.4 Adaptable Priority Queues

The functions of the priority queue ADT given in Section 8.1.3 are sufficient for most basic applications of priority queues such as sorting. However, there are situations where additional functions would be useful as shown in the scenarios below that refer to the standby airline passenger application.

- A standby passenger with a pessimistic attitude may become tired of waiting and decide to leave ahead of the boarding time, requesting to be removed from the waiting list. Thus, we would like to remove the entry associated with this passenger from the priority queue. Operation removeMin is not suitable for this purpose, since it only removes the entry with the lowest priority. Instead, we want a new operation that removes an arbitrary entry.
- Another standby passenger finds her gold frequent-flyer card and shows it to the agent. Thus, her priority has to be modified accordingly. To achieve this change of priority, we would like to have a new operation that changes the information associated with a given entry. This might affect the entry's key value (such as frequent-flyer status) or not (such as correcting a misspelled name).

Functions of the Adaptable Priority Queue ADT

The above scenarios motivate the definition of a new ADT for priority queues, which includes functions for modifying or removing specified entries. In order to do this, we need some way of indicating which entry of the queue is to be affected by the operation. Note that we cannot use the entry's key value, because keys are not distinct. Instead, we assume that the priority queue operation insert(e) is augmented so that, after inserting the element e, it returns a reference to the newly created entry, called a ***position*** (recall Section 6.2.1). This position is permanently attached to the entry, so that, even if the location of the entry changes within the priority queue's internal data structure (as is done when performing bubbling operations in a heap), the position remains fixed to this entry. Thus, positions provide us with a means to uniquely specify the entry to which each operation is applied.

We formally define an ***adaptable priority queue*** P to be a priority queue that, in addition to the standard priority queue operations, supports the following enhancements.

> insert(e): Insert the element e into P and return a position referring to its entry.
>
> remove(p): Remove the entry referenced by p from P.
>
> replace(p, e): Replace with e the element associated with the entry referenced by p and return the position of the altered entry.

8.4.1   A List-Based Implementation

In this section, we present a simple implementation of an adaptable priority queue, called AdaptPriorityQueue. Our implementation is a generalization of the sorted-list priority queue implementation given in Section 8.2.

In Code Fragment 8.7, we present the class definition, with the exception of the class Position, which is presented later. The public part of the class is essentially the same as the standard priority queue interface, which was presented in Code Fragment 8.4, together with the new functions remove and replace. Note that the function insert now returns a position.

```
template <typename E, typename C>
class AdaptPriorityQueue {                  // adaptable priority queue
protected:
  typedef std::list<E> ElementList;         // list of elements
public:
  // ...insert Position class definition here
public:
  int size() const;                         // number of elements
  bool empty() const;                       // is the queue empty?
  const E& min() const;                     // minimum element
  Position insert(const E& e);              // insert element
  void removeMin();                         // remove minimum
  void remove(const Position& p);           // remove at position p
  Position replace(const Position& p, const E& e); // replace at position p
private:
  ElementList L;                            // priority queue contents
  C isLess;                                 // less-than comparator
};
```

Code Fragment 8.19: The class definition for an adaptable priority queue.

We next define the class Position, which is nested within the public part of class AdaptPriorityQueue. Its data member is an iterator to the STL list. This list contains the contents of the priority queue. The main public member is a function that returns a "const" reference the underlying element, which is implemented by overloading the "*" operator. This is presented in Code Fragment 8.20.

```
class Position {                            // a position in the queue
private:
  typename ElementList::iterator q;         // a position in the list
public:
  const E& operator*() { return *q; }       // the element at this position
  friend class AdaptPriorityQueue;          // grant access
};
```

Code Fragment 8.20: The class representing a position in AdaptPriorityQueue.

The operation insert is presented in Code Fragment 8.21. It is essentially the same as presented in the standard list priority queue (see Code Fragment 8.9). Since it is declared outside the class, we need to provide the complete template specifications for the function. We search for the first entry p whose key value exceeds ours, and insert e just prior to this entry. We then create a position that refers to the entry just prior to p and return it.

```
template <typename E, typename C>              // insert element
typename AdaptPriorityQueue<E,C>::Position
AdaptPriorityQueue<E,C>::insert(const E& e) {
  typename ElementList::iterator p = L.begin();
  while (p != L.end() && !isLess(e, *p)) ++p;  // find larger element
  L.insert(p, e);                              // insert before p
  Position pos;  pos.q = −−p;
  return pos;                                  // inserted position
}
```

Code Fragment 8.21: The function insert for class AdaptPriorityQueue.

We omit the definitions of the member functions size, empty, min, and remove-Min, since they are the same as in the standard list-based priority queue implementation (see Code Fragments 8.8 and 8.10). Next, in Code Fragment 8.22, we present the implementations of the functions remove and replace. The function remove invokes the erase function of the STL list to remove the entry referred to by the given position.

```
template <typename E, typename C>              // remove at position p
void AdaptPriorityQueue<E,C>::remove(const Position& p)
  { L.erase(p.q); }

template <typename E, typename C>              // replace at position p
typename AdaptPriorityQueue<E,C>::Position
AdaptPriorityQueue<E,C>::replace(const Position& p, const E& e) {
  L.erase(p.q);                                // remove the old entry
  return insert(e);                            // insert replacement
}
```

Code Fragment 8.22: The functions remove and replace for AdaptPriorityQueue.

We have chosen perhaps the simplest way to implement the function replace. We remove the entry to be modified and simply insert the new element e into the priority queue. In general, the key information may have changed, and therefore it may need to be moved to a new location in the sorted list. Under the assumption that key changes are rare, a more clever solution would involve searching forwards or backwards to determine the proper position for the modified entry. While it may not be very efficient, our approach has the virtue of simplicity.

8.4.2 Location-Aware Entries

In our implementation of the adaptable priority queue, AdaptPriorityQueue, presented in the previous section, we exploited a nice property of the list-based priority queue implementation. In particular, once a new entry is added to the sorted list, the element associated with this entry never changes. This means that the positions returned by the insert and replace functions always refer to the same element.

Note, however, that this same approach would fail if we tried to apply it to the heap-based priority queue of Section 8.3.3. The reason is that the heap-based implementation moves the entries around the heap (for example, through up-heap bubbling and down-heap bubbling). When an element e is inserted, we return a reference to the entry p containing e. But if e were to be moved as a result of subsequent operations applied to the priority queue, p does not change. As a result, p might be pointing to a different element of the priority queue. An attempt to apply remove(p) or replace(p, e'), would not be applied to e but instead to some other element.

The solution to this problem involves decoupling positions and entries. In our implementation of AdaptPriorityQueue, each position p is essentially a pointer to a node of the underlying data structure (for this is how an STL iterator is implemented). If we move an entry, we need to also change the associated pointer. In order to deal with moving entries, each time we insert a new element e in the priority queue, in addition to creating a new entry in the data structure, we also allocate memory for an object, called a ***locator***. The locator's job is to store the current position p of element e in the data structure. Each entry of the priority queue needs to know its associated locator l. Thus, rather than just storing the element itself in the priority queue, we store a pair $(e, \&l)$, consisting of the element e and a pointer to its locator. We call this a ***locator-aware entry***. After inserting a new element in the priority queue, we return the associated locator object, which points to this pair.

How does this solve the decoupling problem? First, observe that whenever the user of the priority queue wants to locate the position p of a previously inserted element, it suffices to access the locator that stores this position. Suppose, however, that the entry moves to a different position p' within the data structure. To handle this, we first access the location-aware entry $(e, \&l)$ to access the locator l. We then modify l so that it refers to the new position p'. The user may find the new position by accessing the locator.

The price we pay for this extra generality is fairly small. For each entry, we need to two store two additional pointers (the locator and the locator's address). Each time we move an object in the data structure, we need to modify a constant number of pointers. Therefore, the running time increases by just a constant factor.

8.5 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-8.1 What are the running times of each of the functions of the (standard) priority queue ADT if we implement it by adapting the STL priority_queue?

R-8.2 How long would it take to remove the $\lceil \log n \rceil$ smallest elements from a heap that contains n entries using the removeMin() operation?

R-8.3 Show that, given only the less-than operator (<) and the boolean operators *and* (&&), *or* (||), and *not* (!), it is possible to implement all of the other comparators: >, <=, >=, ==, !=.

R-8.4 Explain how to implement a priority queue based on the composition method (of storing key-element pairs) by adapting a priority queue based on the comparator approach.

R-8.5 Suppose you label each node v of a binary tree T with a key equal to the preorder rank of v. Under what circumstances is T a heap?

R-8.6 Show the output from the following sequence of priority queue ADT operations. The entries are key-element pairs, where sorting is based on the key value: insert$(5, a)$, insert$(4, b)$, insert$(7, i)$, insert$(1, d)$, removeMin(), insert$(3, j)$, insert$(6, c)$, removeMin(), removeMin(), insert$(8, g)$, remove-Min(), insert$(2, h)$, removeMin(), removeMin().

R-8.7 An airport is developing a computer simulation of air-traffic control that handles events such as landings and takeoffs. Each event has a *time-stamp* that denotes the time when the event occurs. The simulation program needs to efficiently perform the following two fundamental operations:

- Insert an event with a given time-stamp (that is, add a future event)
- Extract the event with smallest time-stamp (that is, determine the next event to process)

Which data structure should be used for the above operations? Why?

R-8.8 Although it is correct to use a "reverse" comparator with our priority queue ADT so that we retrieve and remove an element with the maximum key each time, it is confusing to have an element with the maximum key returned by a function named "removeMin." Write a short adapter class that can take any priority queue P and an associated comparator C and implement a priority queue that concentrates on the element with the maximum key, using functions with names like removeMax.

(Hint: Define a new comparator C' in terms of C.)

R-8.9   Illustrate the performance of the selection-sort algorithm on the following input sequence: $(22, 15, 36, 44, 10, 3, 9, 13, 29, 25)$.

R-8.10 Illustrate the performance of the insertion-sort algorithm on the input sequence of the previous problem.

R-8.11 Give an example of a worst-case sequence with n elements for insertion-sort, and show that insertion-sort runs in $\Omega(n^2)$ time on such a sequence.

R-8.12 At which nodes of a heap can an entry with the largest key be stored?

R-8.13 In defining the relation "to the left of" for two nodes of a binary tree (Section 8.3.1), can we use a preorder traversal instead of an inorder traversal? How about a postorder traversal?

R-8.14 Illustrate the performance of the heap-sort algorithm on the following input sequence: $(2, 5, 16, 4, 10, 23, 39, 18, 26, 15)$.

R-8.15 Let T be a complete binary tree such that node v stores the key-entry pairs $(f(v), 0)$, where $f(v)$ is the level number of v. Is tree T a heap? Why or why not?

R-8.16 Explain why the case where the right child of r is internal and the left child is external was not considered in the description of down-heap bubbling.

R-8.17 Is there a heap T storing seven distinct elements such that a preorder traversal of T yields the elements of T in sorted order? How about an inorder traversal? How about a postorder traversal?

R-8.18 Consider the numbering of the nodes of a binary tree defined in Section 7.3.5, and show that the insertion position in a heap with n keys is the node with number $n + 1$.

R-8.19 Let H be a heap storing 15 entries using the vector representation of a complete binary tree. What is the sequence of indices of the vector that are visited in a preorder traversal of H? What about an inorder traversal of H? What about a postorder traversal of H?

R-8.20 Show that the sum $\sum_{i=1}^{n} \log i$, which appears in the analysis of heap-sort, is $\Omega(n \log n)$.

R-8.21 Bill claims that a preorder traversal of a heap will list its keys in nondecreasing order. Draw an example of a heap that proves him wrong.

R-8.22 Hillary claims that a postorder traversal of a heap will list its keys in nonincreasing order. Draw an example of a heap that proves her wrong.

R-8.23 Show all the steps of the algorithm for removing key 16 from the heap of Figure 8.3.

R-8.24 Draw an example of a heap whose keys are all the odd numbers from 1 to 59 (with no repeats), such that the insertion of an entry with key 32 would cause up-heap bubbling to proceed all the way up to a child of the root (replacing that child's key with 32).

R-8.25 Give a pseudo-code description of a nonrecursive in-place heap-sort algorithm.

R-8.26 A group of children want to play a game, called ***Unmonopoly***, where in each turn the player with the most money must give half of his/her money to the player with the least amount of money. What data structure(s) should be used to play this game efficiently? Why?

Creativity

C-8.1 An online computer system for trading stock needs to process orders of the form "buy 100 shares at $\$x$ each" or "sell 100 shares at $\$y$ each." A buy order for $\$x$ can only be processed if there is an existing sell order with price $\$y$ such that $y \leq x$. Likewise, a sell order for $\$y$ can only be processed if there is an existing buy order with price $\$x$ such that $x \geq y$. If a buy or sell order is entered but cannot be processed, it must wait for a future order that allows it to be processed. Describe a scheme that allows for buy and sell orders to be entered in $O(\log n)$ time, independent of whether or not they can be immediately processed.

C-8.2 Extend a solution to the previous problem so that users are allowed to update the prices for their buy or sell orders that have yet to be processed.

C-8.3 Write a comparator for integer objects that determines order based on the number of 1s in each number's binary expansion, so that $i < j$ if the number of 1s in the binary representation of i is less than the number of 1s in the binary representation of j.

C-8.4 Show how to implement the stack ADT using only a priority queue and one additional member variable.

C-8.5 Show how to implement the (standard) queue ADT using only a priority queue and one additional member variable.

C-8.6 Describe, in detail, an implementation of a priority queue based on a sorted array. Show that this implementation achieves $O(1)$ time for operations min and removeMin and $O(n)$ time for operation insert.

C-8.7 Describe an in-place version of the selection-sort algorithm that uses only $O(1)$ space for member variables in addition to an input array itself.

C-8.8 Assuming the input to the sorting problem is given in an array A, describe how to implement the insertion-sort algorithm using only the array A and, at most, six additional (base-type) variables.

C-8.9 Assuming the input to the sorting problem is given in an array A, describe how to implement the heap-sort algorithm using only the array A and, at most, six additional (base-type) variables.

C-8.10 Describe a sequence of n insertions to a heap that requires $\Omega(n \log n)$ time to process.

C-8.11 An alternative method for finding the last node during an insertion in a heap T is to store, in the last node and each external node of T, a pointer to the external node immediately to its right (wrapping to the first node in the next lower level for the rightmost external node). Show how to maintain such a pointer in $O(1)$ time per operation of the priority queue ADT, assuming T is implemented as a linked structure.

C-8.12 We can represent a path from the root to a given node of a binary tree by means of a binary string, where 0 means "go to the left child" and 1 means "go to the right child." For example, the path from the root to the node storing 8 in the heap of Figure 8.3 is represented by the binary string 101. Design an $O(\log n)$-time algorithm for finding the last node of a complete binary tree with n nodes based on the above representation. Show how this algorithm can be used in the implementation of a complete binary tree by means of a linked structure that does not keep a reference to the last node.

C-8.13 Suppose the binary tree T used to implement a heap can be accessed using only the functions of the binary tree ADT. That is, we cannot assume T is implemented as a vector. Given a pointer to the current last node, v, describe an efficient algorithm for finding the insertion point (that is, the new last node) using just the functions of the binary tree interface. Be sure and handle all possible cases as illustrated in Figure 8.12. What is the running time of this function?
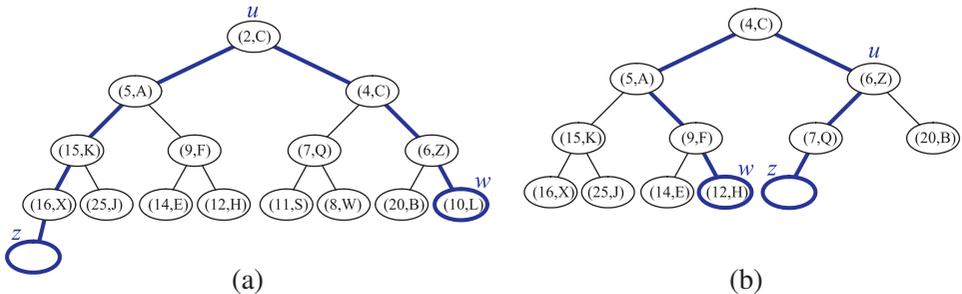
Figure 8.12: Updating the last node in a complete binary tree after operation add or remove. Node w is the last node before operation add or after operation remove. Node z is the last node after operation add or before operation remove.

C-8.14 Given a heap T and a key k, give an algorithm to compute all the entries in T with a key less than or equal to k. For example, given the heap of Figure 8.12(a) and query $k = 7$, the algorithm should report the entries with keys 2, 4, 5, 6, and 7 (but not necessarily in this order). Your algorithm should run in time proportional to the number of entries returned.

C-8.15 Show that, for any n, there is a sequence of insertions in a heap that requires $\Omega(n \log n)$ time to process.

C-8.16 Provide a justification of the time bounds in Table 8.1.

C-8.17 Develop an algorithm that computes the kth smallest element of a set of n distinct integers in $O(n + k \log n)$ time.

C-8.18 Suppose the internal nodes of two binary trees, T_1 and T_2 respectively, hold items that satisfy the heap-order property. Describe a method for combining these two trees into a tree T, whose internal nodes hold the union of the items in T_1 and T_2 and also satisfy the heap-order property. Your algorithms should run in time $O(h_1 + h_2)$ where h_1 and h_2 are the respective heights of T_1 and T_2.

C-8.19 Give an alternative analysis of bottom-up heap construction by showing that, for any positive integer h, $\sum_{i=1}^{h}(i/2^i)$ is $O(1)$.

C-8.20 Let T be a heap storing n keys. Give an efficient algorithm for reporting all the keys in T that are smaller than or equal to a given query key x (which is not necessarily in T). For example, given the heap of Figure 8.3 and query key $x = 7$, the algorithm should report 4, 5, 6, 7. Note that the keys do not need to be reported in sorted order. Ideally, your algorithm should run in $O(k)$ time, where k is the number of keys reported.

C-8.21 Give an alternate description of the in-place heap-sort algorithm that uses a standard comparator instead of a reverse one.

C-8.22 Describe efficient algorithms for performing operations remove(e) on an adaptable priority queue realized by means of an unsorted list with location-aware entries.

C-8.23 Let S be a set of n points in the plane with distinct integer x- and y-coordinates. Let T be a complete binary tree storing the points from S at its external nodes, such that the points are ordered left-to-right by increasing x-coordinates. For each node v in T, let $S(v)$ denote the subset of S consisting of points stored in the subtree rooted at v. For the root r of T, define $top(r)$ to be the point in $S = S(r)$ with maximum y-coordinate. For every other node v, define $top(r)$ to be the point in S with highest y-coordinate in $S(v)$ that is not also the highest y-coordinate in $S(u)$, where u is the parent of v in T (if such a point exists). Such labeling turns T into a **priority search tree**. Describe a linear-time algorithm for turning T into a priority search tree.

Projects

P-8.1 Generalize the Heap data structure of Section 8.3 from a binary tree to a k-ary tree, for an arbitrary $k \geq 2$. Study the relative efficiencies of the

resulting data structure for various values of k, by inserting and removing a large number of randomly generated keys into each data structure.

P-8.2 Give a C++ implementation of a priority queue based on an unsorted list.

P-8.3 Develop a C++ implementation of a priority queue that is based on a heap and supports the locator-based functions.

P-8.4 Implement the in-place heap-sort algorithm. Compare its running time with that of the standard heap-sort that uses an external heap.

P-8.5 Implement a heap-based priority queue that supports the following additional operation in linear time:

    replaceComparator(c): Replace the current comparator with c.

After changing the comparator, the heap will need to be restructured. (Hint: Utilize the bottom-up heap construction algorithm.)

P-8.6 Write a program that can process a sequence of stock buy and sell orders as described in Exercise C-8.1.

P-8.7 One of the main applications of priority queues is in operating systems— for *scheduling jobs* on a CPU. In this project you are to build a program that schedules simulated CPU jobs. Your program should run in a loop, each iteration of which corresponds to a *time slice* for the CPU. Each job is assigned a priority, which is an integer between -20 (highest priority) and 19 (lowest priority), inclusive. From among all jobs waiting to be processed in a time slice, the CPU must work on the job with highest priority. In this simulation, each job will also come with a *length* value, which is an integer between 1 and 100, inclusive, indicating the number of time slices that are needed to process this job. For simplicity, you may assume jobs cannot be interrupted—once it is scheduled on the CPU, a job runs for a number of time slices equal to its length. Your simulator must output the name of the job running on the CPU in each time slice and must process a sequence of commands, one per time slice, each of which is of the form "add job *name* with length n and priority p" or "no new job this slice."

Chapter Notes

Knuth's book on sorting and searching [57] describes the motivation and history for the selection-sort, insertion-sort, and heap-sort algorithms. The heap-sort algorithm is due to Williams [103], and the linear-time heap construction algorithm is due to Floyd [33]. Additional algorithms and analyses for heaps and heap-sort variations can be found in papers by Bentley [12], Carlsson [20], Gonnet and Munro [38], McDiarmid and Reed [70], and Schaffer and Sedgewick [88]. The design pattern of using location-aware entries (also described in [39]), appears to be new.

Chapter

9

Hash Tables, Maps, and Skip Lists

Contents

9.1   Maps

Figure 9.1: A conceptual illustration of the map ADT. Keys (labels) are assigned to values (folders) by a user. The resulting entries (labeled folders) are inserted into the map (file cabinet). The keys can be used later to retrieve or remove values.

A **map** allows us to store elements so they can be located quickly using keys. The motivation for such searches is that each element typically stores additional useful information besides its search key, but the only way to get at that information is to use the search key. Specifically, a map stores key-value pairs (k, v), which we call **entries**, where k is the key and v is its corresponding value. In addition, the map ADT requires that each key be unique, so the association of keys to values defines a mapping. In order to achieve the highest level of generality, we allow both the keys and the values stored in a map to be of any object type. (See Figure 9.1.) In a map storing student records (such as the student's name, address, and course grades), the key might be the student's ID number. In some applications, the key and the value may be the same. For example, if we had a map storing prime numbers, we could use each number itself as both a key and its value.

In either case, we use a **key** as a unique identifier that is assigned by an application or user to an associated value object. Thus, a map is most appropriate in situations where each key is to be viewed as a kind of unique **index** address for its value, that is, an object that serves as a kind of location for that value. For example, if we wish to store student records, we would probably want to use student ID objects as keys (and disallow two students having the same student ID). In other words, the key associated with an object can be viewed as an "address" for that object. Indeed, maps are sometimes referred to as **associative stores** or **associative containers**, because the key associated with an object determines its "location" in the data structure.

Entries and the Composition Pattern

As mentioned above, a map stores key-value pairs, called entries. An entry is actually an example of a more general object-oriented design pattern, the ***composition pattern***, which defines a single object that is composed of other objects. A pair is the simplest composition, because it combines two objects into a single pair object.

To implement this concept, we define a class that stores two objects in its first and second member variables, respectively, and provides functions to access and update these variables. In Code Fragment 9.1, we present such an implementation storing a single key-value pair. We define a class Entry, which is templated based on the key and value types. In addition to a constructor, it provides member functions that return references to the key and value. It also provides functions that allow us to set the key and value members.

```cpp
template <typename K, typename V>
class Entry {                                  // a (key, value) pair
public:                                         // public functions
  Entry(const K& k = K(), const V& v = V())    // constructor
    : _key(k), _value(v) { }
  const K& key() const { return _key; }         // get key
  const V& value() const { return _value; }     // get value
  void setKey(const K& k) { _key = k; }         // set key
  void setValue(const V& v) { _value = v; }     // set value
private:                                         // private data
  K _key;                                        // key
  V _value;                                      // value
};
```

Code Fragment 9.1: A C++ class for an entry storing a key-value pair.

9.1.1 The Map ADT

In this section, we describe a map ADT. Recall that a map is a collection of key-value entries, with each value associated with a distinct key. We assume that a map provides a special pointer object, which permits us to reference entries of the map. Such an object would normally be called a ***position***. As we did in Chapter 6, in order to be more consistent with the C++ Standard Template Library, we define a somewhat more general object called an ***iterator***, which can both reference entries and navigate around the map. Given a map iterator p, the associated entry may be accessed by dereferencing the iterator, namely as $*p$. The individual key and value can be accessed using p->key() and p->value(), respectively.

In order to advance an iterator from its current position to the next, we overload the increment operator. Thus, $++p$ advances the iterator p to the next entry of the

map. We can enumerate all the entries of a map M by initializing p to M.begin() and then repeatedly incrementing p as long as it is not equal to M.end().

In order to indicate that an object is not present in the map, we assume that there exists a special sentinel iterator called end. By convention, this sentinel refers to an imaginary element that lies just beyond the last element of the map.

The map ADT consists of the following:

size(): Return the number of entries in M.

empty(): Return true if M is empty and false otherwise.

find(k): If M contains an entry $e = (k, v)$, with key equal to k, then return an iterator p referring to this entry, and otherwise return the special iterator end.

put(k, v): If M does not have an entry with key equal to k, then add entry (k, v) to M, and otherwise, replace the value field of this entry with v; return an iterator to the inserted/modified entry.

erase(k): Remove from M the entry with key equal to k; an error condition occurs if M has no such entry.

erase(p): Remove from M the entry referenced by iterator p; an error condition occurs if p points to the end sentinel.

begin(): Return an iterator to the first entry of M.

end(): Return an iterator to a position just beyond the end of M.

We have provided two means of removing entries, one given a key and the other given an iterator. The key-based operation should be used only when it is known that the key is present in the map. Otherwise, it is necessary to first check that the key exists using the operation "$p = M$.find(k)," and if so, then apply the operation M.erase(p). The iterator-based removal operation has the advantage that it does not need to repeat the search for the key, and hence is more efficient.

The operation put, may either insert an entry or modify an existing entry. It is designed explicitly in this way, since we require that the keys be unique. Later, in Section 9.5, we consider a different data structure, which allows multiple instances to have the same keys. Note that an iterator remains associated with an entry, even if its value is changed.

Example 9.1: *In the following, we show the effect of a series of operations on an initially empty map storing entries with integer keys and single-character values. In the column "Output," we use the notation $p_i : [(k, v)]$ to mean that the operation returns an iterator denoted by p_i that refers to the entry (k, v). The entries of the map are not listed in any particular order.*

Operation	*Output*	*Map*
empty()	**true**	$\emptyset$
put(5, A)	$p_1 : [(5,A)]$	$\{(5,A)\}$
put(7, B)	$p_2 : [(7,B)]$	$\{(5,A),(7,B)\}$
put(2, C)	$p_3 : [(2,C)]$	$\{(5,A),(7,B),(2,C)\}$
put(2, E)	$p_3 : [(2,E)]$	$\{(5,A),(7,B),(2,E)\}$
find(7)	$p_2 : [(7,B)]$	$\{(5,A),(7,B),(2,E)\}$
find(4)	end	$\{(5,A),(7,B),(2,E)\}$
find(2)	$p_3 : [(2,E)]$	$\{(5,A),(7,B),(2,E)\}$
size()	3	$\{(5,A),(7,B),(2,E)\}$
erase(5)	–	$\{(7,B),(2,E)\}$
erase(p_3)	–	$\{(7,B)\}$
find(2)	end	$\{(7,B)\}$

9.1.2 A C++ Map Interface

Before discussing specific implementations of the map ADT, we first define a C++ interface for a map in Code Fragment 9.2. It is not a complete C++ class, just a declaration of the public functions. The interface is templated by two types, the key type K, and the value type V.

```
template <typename K, typename V>
class Map {                                // map interface
public:
  class Entry;                             // a (key,value) pair
  class Iterator;                          // an iterator (and position)

  int size() const;                        // number of entries in the map
  bool empty() const;                      // is the map empty?
  Iterator find(const K& k) const;         // find entry with key k
  Iterator put(const K& k, const V& v);    // insert/replace pair (k,v)
  void erase(const K& k)                   // remove entry with key k
    throw(NonexistentElement);
  void erase(const Iterator& p);           // erase entry at p
  Iterator begin();                        // iterator to first entry
  Iterator end();                          // iterator to end entry
};
```

Code Fragment 9.2: An informal C++ Map interface (not a complete class).

In addition to its member functions, the interface defines two types, Entry and Iterator. These two classes provide the types for the entry and iterator objects, respectively. Outside the class, these would be accessed with Map<K,V>::Entry and Map<K,V>::Iterator, respectively.

We have not presented an interface for the iterator object, but its definition is similar to the STL iterator. It supports the operator "∗", which returns a reference to the associated entry. The unary increment and decrement operators "++" and "−−" move an iterator forward and backwards, respectively. Also, two iterators can be compared for equality using "==".

A more sophisticated implementation would have also provided for a third type, namely a "const" iterator. Such an iterator provides a function for reading entries without modifying them. (Recall Section 6.1.4.) We omit this type in order to keep our interface relatively simple.

The remainder of the interface follows from our earlier descriptions of the map operations. An error condition occurs if the function erase(k) is called with a key k that is not in the map. This is signaled by throwing an exception of type NonexistentElement. Its definition is similar to other exceptions that we have seen. (See Code Fragment 5.2.)

9.1.3   The STL map Class

The C++ Standard Template Library (STL) provides an implementation of a map simply called map. As with many of the other STL classes we have seen, the STL map is an example of a container, and hence supports access by iterators.

In order to declare an object of type map, it is necessary to first include the definition file called "map." The map is part of the std namespace, and hence it is necessary either to use "std::map" or to provide an appropriate "**using**" statement.

The STL map is templated with two arguments, the key type and the value type. The declaration "map<K,V>" defines a map whose keys are of type K and whose values are of type V. As with the other STL containers, an iterator type is provided both for referencing individual entries and enumerating multiple entries. The map iterator type is "map<K,E>::iterator." The (k, v) entries are stored in a composite object called pair. Given an iterator p, its associated key and value members can be referenced using p->*first* and p->*second*, respectively. (These are equivalent to p->key() and p->value() in our map ADT, but note that there are no parentheses following *first* and *second*.)

As with other iterators we have seen, each map object M defines two special iterators through the functions begin and end, where M.begin() yields an iterator to the first element of the map, and M.end() yields an iterator to an imaginary element just beyond the end of the map. A map iterator p is bidirectional, meaning that we can move forwards and backwards through the map using the increment and decrement operators, $++p$ and $--p$, respectively.

The principal member functions of the STL map are given below. Let M be declared to be an STL map, let k be a key object, and let v be a value object for the class M. Let p be an iterator for M.

size(): Return the number of elements in the map.

empty(): Return true if the map is empty and false otherwise.

find(k): Find the entry with key k and return an iterator to it; if no such key exists return end.

operator$[k]$: Produce a reference to the value of key k; if no such key exists, create a new entry for key k.

insert(pair(k, v)): Insert pair (k, v), returning an iterator to its position.

erase(k): Remove the element with key k.

erase(p): Remove the element referenced by iterator p.

begin(): Return an iterator to the beginning of the map.

end(): Return an iterator just past the end of the map.

Our map ADT is quite similar to the above functions. The insert function is a bit different. In our ADT, it is given two arguments. In the STL map, the argument is a composite object of type pair, whose first and second elements are the key and value, respectively.

The STL map provides a very convenient way to search, insert, and modify entries by overloading the subscript operator ("[ ]"). Given a map M, the assignment "$M[k] = v$" inserts the pair (k, v) if k is not already present, or modifies the value if it is. Thus, the subscript assignment behaves essentially the same as our ADT function put(k, v). Reading the value of $M[k]$ is equivalent to performing find(k) and accessing the value part of the resulting iterator. An example of the use of the STL map is shown in Code Fragment 9.3.

```cpp
map<string, int> myMap;                      // a (string,int) map
map<string, int>::iterator p;                // an iterator to the map
myMap.insert(pair<string, int>("Rob", 28));  // insert ("Rob",28)
myMap["Joe"] = 38;                           // insert("Joe",38)
myMap["Joe"] = 50;                           // change to ("Joe",50)
myMap["Sue"] = 75;                           // insert("Sue",75)
p = myMap.find("Joe");                       // *p = ("Joe",50)
myMap.erase(p);                              // remove ("Joe",50)
myMap.erase("Sue");                          // remove ("Sue",75)
p = myMap.find("Joe");
if (p == myMap.end()) cout << "nonexistent\n"; // outputs: "nonexistent"
for (p = myMap.begin(); p != myMap.end(); ++p) { // print all entries
  cout << "(" << p->first << "," << p->second << ")\n";
}
```

Code Fragment 9.3: Example of the usage of STL map.

As with the other STL containers we have seen, the STL does not check for errors. It is up to the programmer to be sure that no illegal operations are performed.

9.1.4   A Simple List-Based Map Implementation

A simple way of implementing a map is to store its n entries in a list L, implemented as a doubly linked list. Performing the fundamental functions, find(k), put(k,v), and erase(k), involves simple scans down L looking for an entry with key k. Pseudo-code is presented in Code Fragments 9.4. We use the notation $[L.\text{begin}(), L.\text{end}())$ to denote all the positions of list L, from $L.\text{begin}()$ and up to, but not including, $L.\text{end}()$.

Algorithm find(k):
   Input: A key k
   Output: The position of the matching entry of L, or end if there is no key k in L
   for each position $p \in [L.\text{begin}(), L.\text{end}())$ **do**
     if $p.\text{key}() = k$ **then**
       return p
   return end   {there is no entry with key equal to k}

Algorithm put(k,v):
   Input: A key-value pair (k,v)
   Output: The position of the inserted/modified entry
   for each position $p \in [L.\text{begin}(), L.\text{end}())$ **do**
     if $p.\text{key}() = k$ **then**
       $*p \leftarrow (k,v)$
       return p {return the position of the modified entry}
   $p \leftarrow L.\text{insertBack}((k,v))$
   $n \leftarrow n+1$ {increment variable storing number of entries}
   return p {return the position of the inserted entry}

Algorithm erase(k):
   Input: A key k
   Output: None
   for each position $p \in [L.\text{begin}(), L.\text{end}())$ **do**
     if $p.\text{key}() = k$ **then**
       $L.\text{erase}(p)$
       $n \leftarrow n-1$ {decrement variable storing number of entries}

Code Fragment 9.4: Algorithms for find, put, and erase for a map stored in a list L.

This list-based map implementation is simple, but it is only efficient for very small maps. Every one of the fundamental functions takes $O(n)$ time on a map with n entries, because each function involves searching through the entire list in the worst case. Thus, we would like something much faster.

9.2   Hash Tables

The keys associated with values in a map are typically thought of as "addresses" for those values. Examples of such applications include a compiler's symbol table and a registry of environment variables. Both of these structures consist of a collection of symbolic names where each name serves as the "address" for properties about a variable's type and value. One of the most efficient ways to implement a map in such circumstances is to use a ***hash table***. Although, as we will see, the worst-case running time of map operations in an *n*-entry hash table is $O(n)$. A hash table can usually perform these operations in $O(1)$ expected time. In general, a hash table consists of two major components, a ***bucket array*** and a ***hash function***.

9.2.1   Bucket Arrays

A ***bucket array*** for a hash table is an array A of size N, where each cell of A is thought of as a "bucket" (that is, a collection of key-value pairs) and the integer N defines the ***capacity*** of the array. If the keys are integers well distributed in the range $[0, N-1]$, this bucket array is all that is needed. An entry e with key k is simply inserted into the bucket $A[k]$. (See Figure 9.2.)
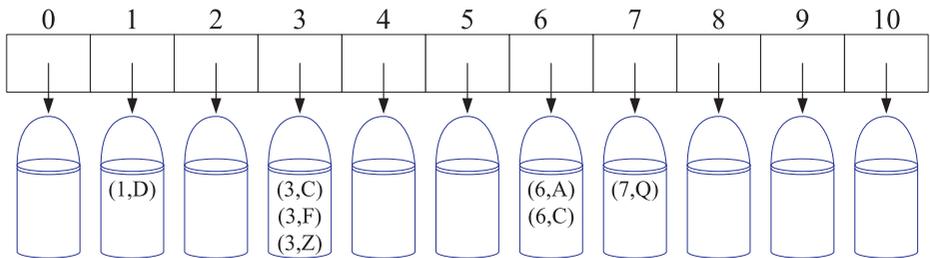
Figure 9.2: A bucket array of size 11 for the entries (1,D), (3,C), (3,F), (3,Z), (6,A), (6,C), and (7,Q).

If our keys are unique integers in the range $[0, N-1]$, then each bucket holds at most one entry. Thus, searches, insertions, and removals in the bucket array take $O(1)$ time. This sounds like a great achievement, but it has two drawbacks. First, the space used is proportional to N. Thus, if N is much larger than the number of entries n actually present in the map, we have a waste of space. The second drawback is that keys are required to be integers in the range $[0, N-1]$, which is often not the case. Because of these two drawbacks, we use the bucket array in conjunction with a "good" mapping from the keys to the integers in the range $[0, N-1]$.

9.2.2   Hash Functions

The second part of a hash table structure is a function, h, called a ***hash function***, that maps each key k in our map to an integer in the range $[0, N-1]$, where N is the capacity of the bucket array for this table. Equipped with such a hash function, h, we can apply the bucket array method to arbitrary keys. The main idea of this approach is to use the hash function value, $h(k)$, as an index into our bucket array, A, instead of the key k (which is most likely inappropriate for use as a bucket array index). That is, we store the entry (k, v) in the bucket $A[h(k)]$.

Of course, if there are two or more keys with the same hash value, then two different entries will be mapped to the same bucket in A. In this case, we say that a ***collision*** has occurred. Clearly, if each bucket of A can store only a single entry, then we cannot associate more than one entry with a single bucket, which is a problem in the case of collisions. To be sure, there are ways of dealing with collisions, which we discuss later, but the best strategy is to try to avoid them in the first place. We say that a hash function is "good" if it maps the keys in our map in such a way as to minimize collisions as much as possible. For practical reasons, we also would like a hash function to be fast and easy to compute.

We view the evaluation of a hash function, $h(k)$, as consisting of two actions—mapping the key k to an integer, called the ***hash code***, and mapping the hash code to an integer within the range of indices ($[0, N-1]$) of a bucket array, called the ***compression function***. (See Figure 9.3.)
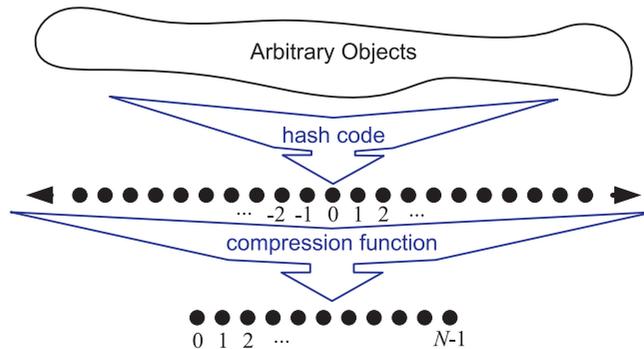
Figure 9.3: The two parts of a hash function: hash code and compression function.

9.2.3   Hash Codes

The first action that a hash function performs is to take an arbitrary key k in our map and assign it an integer value. The integer assigned to a key k is called the ***hash code*** for k. This integer value need not be in the range $[0, N-1]$, and may even be

negative, but we want the set of hash codes assigned to our keys to avoid collisions as much as possible. If the hash codes of our keys cause collisions, then there is no hope for our compression function to avoid them. In addition, to be consistent with all of our keys, the hash code we use for a key k should be the same as the hash code for any key that is equal to k.

Hash Codes in C++

The hash codes described below are based on the assumption that the number of bits of each type is known. This information is provided in the standard include file <limits>. This include file defines a templated class numeric_limits. Given a base type T (such as **char**, **int**, or **float**), the number of bits in a variable of type T is given by "numeric_limits<T>.digits." Let us consider several common data types and some example functions for assigning hash codes to objects of these types.

Converting to an Integer

To begin, we note that, for any data type X that is represented using at most as many bits as our integer hash codes, we can simply take an integer interpretation of its bits as a hash code for X. Thus, for the C++ fundamental types **char**, **short**, and **int**, we can achieve a good hash code simply by casting this type to **int**.

On many machines, the type **long** has a bit representation that is twice as long as type **int**. One possible hash code for a **long** object is to simply cast it down to an integer and then apply the integer hash code. The problem is that such a hash code ignores half of the information present in the original value. If many of the keys in our map only differ in these bits, they will collide using this simple hash code. A better hash code, which takes all the original bits into consideration, sums an integer representation of the high-order bits with an integer representation of the low-order bits.

Indeed, the approach of summing components can be extended to any object x whose binary representation can be viewed as a k-tuple $(x_0, x_1, \ldots, x_{k-1})$ of integers, because we can then form a hash code for x as $\sum_{i=0}^{k-1} x_i$. For example, given any floating-point number, we can sum its mantissa and exponent as long integers, and then apply a hash code for long integers to the result.

Polynomial Hash Codes

The summation hash code, described above, is not a good choice for character strings or other variable-length objects that can be viewed as tuples of the form $(x_0, x_1, \ldots, x_{k-1})$, where the order of the x_i's is significant. For example, consider a hash code for a character string s that sums the ASCII values of the characters

in *s*. Unfortunately, this hash code produces lots of unwanted collisions for common groups of strings. In particular, `"temp01"` and `"temp10"` collide using this function, as do `"stop"`, `"tops"`, `"pots"`, and `"spot"`. A better hash code takes into consideration the positions of the x_i's. An alternative hash code, which does exactly this, chooses a nonzero constant, $a \neq 1$, and uses

$$x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$$

as a hash code value. Mathematically speaking, this is simply a polynomial in *a* that takes the components $(x_0, x_1, \ldots, x_{k-1})$ of an object *x* as its coefficients. This hash code is therefore called a *polynomial hash code*. By Horner's rule (see Exercise C-4.16), this polynomial can be rewritten as

$$x_{k-1} + a(x_{k-2} + a(x_{k-3} + \cdots + a(x_2 + a(x_1 + ax_0)) \cdots)).$$

Intuitively, a polynomial hash code uses multiplication by the constant *a* as a way of "making room" for each component in a tuple of values, while also preserving a characterization of the previous components. Of course, on a typical computer, evaluating a polynomial is done using the finite bit representation for a hash code; hence, the value periodically overflows the bits used for an integer. Since we are more interested in a good spread of the object *x* with respect to other keys, we simply ignore such overflows. Still, we should be mindful that such overflows are occurring and choose the constant *a* so that it has some nonzero, low-order bits, which serve to preserve some of the information content even if we are in an overflow situation.

We have done some experimental studies that suggest that 33, 37, 39, and 41 are good choices for *a* when working with character strings that are English words. In fact, in a list of over 50,000 English words formed as the union of the word lists provided in two variants of Unix, we found that taking *a* to be 33, 37, 39, or 41 produced less than seven collisions in each case! Many implementations of string hashing choose a polynomial hash function, using one of these constants for *a*, as a default hash code. For the sake of speed, however, some implementations only apply the polynomial hash function to a fraction of the characters in long strings.

Cyclic Shift Hash Codes

A variant of the polynomial hash code replaces multiplication by *a* with a cyclic shift of a partial sum by a certain number of bits. Such a function, applied to character strings in C++ could, for example, look like the following. We assume a 32-bit integer word length, and we assume access to a function hashCode(x) for integers. To achieve a 5-bit cyclic shift we form the "bitwise or" (see Section 1.2) of a 5-bit left shift and a 27-bit right shift. As before, we use an unsigned integer so that right shifts fill with zeros.

```
int hashCode(const char* p, int len) {        // hash a character array
  unsigned int h = 0;
  for (int i = 0; i < len; i++) {
    h = (h << 5) | (h >> 27);                  // 5-bit cyclic shift
    h += (unsigned int) p[i];                  // add in next character
  }
  return hashCode(int(h));
}
```

As with the traditional polynomial hash code, using the cyclic-shift hash code requires some fine-tuning. In this case, we must wisely choose the amount to shift by for each new character.

Experimental Results

In Table 9.1, we show the results of some experiments run on a list of just over 25,000 English words, which compare the number of collisions for various shift amounts.

	Collisions			Collisions	
Shift	*Total*	*Max*	*Shift*	*Total*	*Max*
0	23739	86	9	18	2
1	10517	21	10	277	3
2	2254	6	11	453	4
3	448	3	12	43	2
4	89	2	13	13	2
5	4	2	14	135	3
6	6	2	15	1082	6
7	14	2	16	8760	9
8	105	2			

Table 9.1: Comparison of collision behavior for the cyclic shift variant of the polynomial hash code as applied to a list of just over 25,000 English words. The "Total" column records the total number of collisions and the "Max" column records the maximum number of collisions for any one hash code. Note that, with a cyclic shift of 0, this hash code reverts to the one that simply sums all the characters.

These and our previous experiments show that if we choose our constant a or our shift value wisely, then either the polynomial hash code or its cyclic-shift variant are suitable for any object that can be written as a tuple $(x_0, x_1, \ldots, x_{k-1})$,

where the order in tuples matters. In particular, note that using a shift of 5 or 6 is particularly good for English words. Also, note how poorly a simple addition of the values would be with no shifting (that is, for a shift of 0).

Hashing Floating-Point Quantities

On most machines, types **int** and **float** are both 32-bit quantities. Nonetheless, the approach of casting a **float** variable to type **int** would not produce a good hash function, since this would truncate the fractional part of the floating-point value. For the purposes of hashing, we do not really care about the number's value. It is sufficient to treat the number as a sequence of bits. Assuming that a **char** is stored as an 8-bit byte, we could interpret a 32-bit **float** as a four-element character array, and a 64-bit **double** as an eight-element character array. C++ provides an operation called a ***reinterpret cast***, to cast between such unrelated types. This cast treats quantities as a sequence of bits and makes no attempt to intelligently convert the meaning of one quantity to another.

For example, we could design a hash function for a **float** by first reinterpreting it as an array of characters and then applying the character-array hashCode function defined above. We use the operator **sizeof**, which returns the number of bytes in a type.

```
int hashCode(const float& x) {                    // hash a float
  int len = sizeof(x);
  const char* p = reinterpret_cast<const char*>(&x);
  return hashCode(p, len);
}
```

Reinterpret casts are generally not portable operations, since the result depends on the particular machine's encoding of types as a pattern of bits. In our case, portability is not an issue since we are interested only in interpreting the floating point value as a sequence of bits. The only property that we require is that float variables with equal values must have the same bit sequence.

9.2.4   Compression Functions

The hash code for a key k is typically not suitable for immediate use with a bucket array, because the range of possible hash codes for our keys typically exceeds the range of legal indices of our bucket array A. That is, incorrectly using a hash code as an index into our bucket array may result in an error condition, either because the index is negative or it exceeds the capacity of A. Thus, once we have determined an integer hash code for a key object k, there is still the issue of mapping that integer

into the range $[0, N-1]$. This compression step is the second action that a hash function performs.

The Division Method

One simple ***compression function*** to use is

$$h(k) = |k| \bmod N,$$

which is called the ***division method***. Additionally, if we take N to be a prime number, then this hash function helps "spread out" the distribution of hashed values. Indeed, if N is not prime, there is a higher likelihood that patterns in the distribution of keys will be repeated in the distribution of hash codes, thereby causing collisions. For example, if we hash the keys $\{200, 205, 210, 215, 220, \ldots, 600\}$ to a bucket array of size 100 using the division method, then each hash code collides with three others. But if this same set of keys is similarly hashed to a bucket array of size 101, then there are no collisions. If a hash function is chosen well, it should ensure that the probability of two different keys getting hashed to the same bucket is $1/N$. Choosing N to be a prime number is not always enough, however, because if there is a repeated pattern of key values of the form $iN + j$ for several different i's, then there are still collisions.

The MAD Method

A more sophisticated compression function, which helps eliminate repeated patterns in a set of integer keys is the ***multiply add and divide*** (or "MAD") method. In using this method, we define the compression function as

$$h(k) = |ak + b| \bmod N,$$

where N is a prime number, and a and b are nonnegative integers randomly chosen at the time the compression function is determined, so that $a \bmod N \neq 0$. This compression function is chosen in order to eliminate repeated patterns in the set of hash codes and to get us closer to having a "good" hash function, that is, one having the probability that any two different keys collide is $1/N$. This good behavior would be the same as if these keys were "thrown" into A uniformly at random.

With a compression function such as this, that spreads n integers fairly evenly in the range $[0, N-1]$, and a mapping of the keys in our map to integers, we have an effective hash function. Together, such a hash function and a bucket array define the key ingredients of the hash table implementation of the map ADT.

But before we can give the details of how to perform such operations as find, insert, and erase, we must first resolve the issue of how we to handle collisions.

9.2.5   Collision-Handling Schemes

The main idea of a hash table is to take a bucket array, A, and a hash function, h, and use them to implement a map by storing each entry (k, v) in the "bucket" $A[h(k)]$. This simple idea is challenged, however, when we have two distinct keys, k_1 and k_2, such that $h(k_1) = h(k_2)$. The existence of such **collisions** prevents us from simply inserting a new entry (k, v) directly in the bucket $A[h(k)]$. Collisions also complicate our procedure for performing the find(k), put(k, v), and erase(k) operations.

Separate Chaining

A simple and efficient way for dealing with collisions is to have each bucket $A[i]$ store a small map, M_i, implemented using a list, as described in Section 9.1.4, holding entries (k, v) such that $h(k) = i$. That is, each separate M_i chains together the entries that hash to index i in a linked list. This **collision-resolution** rule is known as **separate chaining**. Assuming that we initialize each bucket $A[i]$ to be an empty list-based map, we can easily use the separate-chaining rule to perform the fundamental map operations as shown in Code Fragment 9.5.

Algorithm find(k):
    Output: The position of the matching entry of the map, or end if there is no key
        k in the map
    return $A[h(k)]$.find(k) {delegate the find(k) to the list-based map at $A[h(k)]$}

Algorithm put(k, v):
    $p \leftarrow A[h(k)]$.put(k, v) {delegate the put to the list-based map at $A[h(k)]$}
    $n \leftarrow n + 1$
    return p

Algorithm erase(k):
    Output: None
    $A[h(k)]$.erase(k) {delegate the erase to the list-based map at $A[h(k)]$}
    $n \leftarrow n - 1$

Code Fragment 9.5: The fundamental functions of the map ADT, implemented with a hash table that uses separate chaining to resolve collisions among its n entries.

For each fundamental map operation involving a key k, the separate-chaining approach delegates the handling of this operation to the miniature list-based map stored at $A[h(k)]$. So, put(k, v) scans this list looking for an entry with key equal to k; if it finds one, it replaces its value with v, otherwise, it puts (k, v) at the end of this list. Likewise, find(k) searches through this list until it reaches the end or

finds an entry with key equal to k. And erase(k) performs a similar search but additionally removes an entry after it is found. We can "get away" with this simple list-based approach because the spreading properties of the hash function help keep each bucket's list small. Indeed, a good hash function tries to minimize collisions as much as possible, which implies that most of our buckets are either empty or store just a single entry. In Figure 9.4, we give an illustration of a hash table with separate chaining.
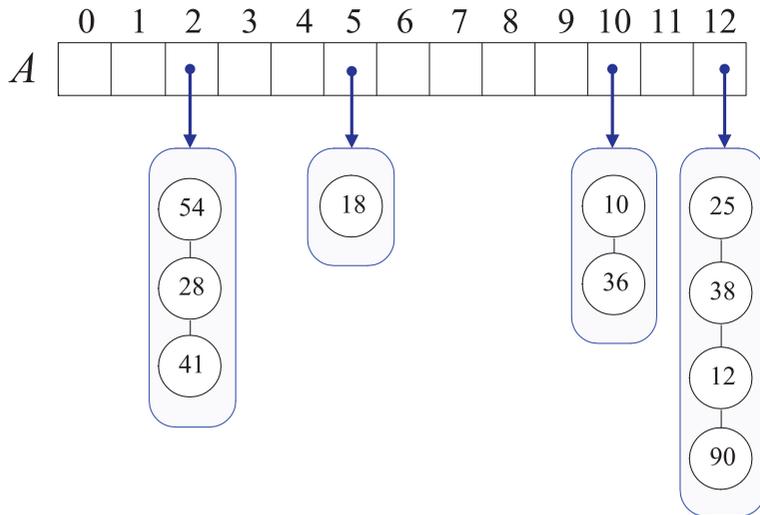
Figure 9.4: A hash table of size 13, storing 10 entries with integer keys, with collisions resolved by separate chaining. The compression function is $h(k) = k \bmod 13$. For simplicity, we do not show the values associated with the keys.

Assuming we use a good hash function to index the n entries of our map in a bucket array of capacity N, we expect each bucket to be of size n/N. This value, called the **load factor** of the hash table (and denoted with λ), should be bounded by a small constant, preferably below 1. Given a good hash function, the expected running time of operations find, put, and erase in a map implemented with a hash table that uses this function is $O(\lceil n/N \rceil)$. Thus, we can implement these operations to run in $O(1)$ expected time provided n is $O(N)$.

Open Addressing

The separate-chaining rule has many nice properties, such as allowing for simple implementations of map operations, but it nevertheless has one slight disadvantage. It requires the use of an auxiliary data structure—a list—to hold entries with colliding keys. We can handle collisions in other ways besides using the separate-

chaining rule, however. In particular, if space is at a premium (for example, if we are writing a program for a small handheld device), then we can use the alternative approach of always storing each entry directly in a bucket, at most one entry per bucket. This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to deal with collisions. There are several variants of this approach, collectively referred to as ***open-addressing*** schemes, which we discuss next. Open addressing requires that the load factor is always at most 1 and that entries are stored directly in the cells of the bucket array itself.

Linear Probing and its Variants

A simple open-addressing method for collision handling is ***linear probing***. In this method, if we try to insert an entry (k, v) into a bucket $A[i]$ that is already occupied (where $i = h(k)$), then we try next at $A[(i+1) \bmod N]$. If $A[(i+1) \bmod N]$ is also occupied, then we try $A[(i+2) \bmod N]$, and so on, until we find an empty bucket that can accept the new entry. Once this bucket is located, we simply insert the entry there. Of course, this collision-resolution strategy requires that we change the implementation of the get(k, v) operation. In particular, to perform such a search, followed by either a replacement or insertion, we must examine consecutive buckets, starting from $A[h(k)]$, until we either find an entry with key equal to k or we find an empty bucket. (See Figure 9.5.) The name "linear probing" comes from the fact that accessing a cell of the bucket array can be viewed as a "probe."

must probe 4 times
before finding empty slot

New element with
key = 15 to be inserted

0	1	2	3	4	5	6	7	8	9	10
		13		26	5	37	16			21
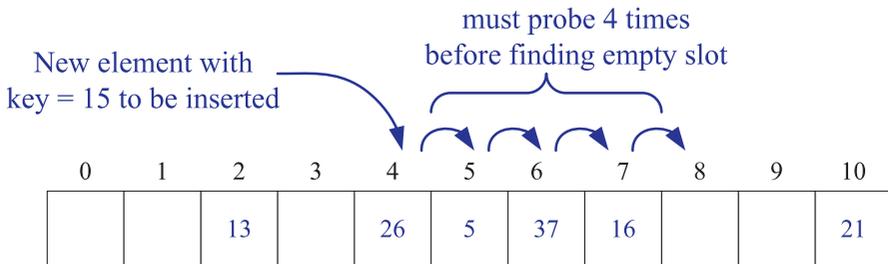
Figure 9.5: An insertion into a hash table using linear probing to resolve collisions. Here we use the compression function $h(k) = k \bmod 11$.

   To implement erase(k), we might, at first, think we need to do a considerable amount of shifting of entries to make it look as though the entry with key k was never inserted, which would be very complicated. A typical way to get around this difficulty is to replace a deleted entry with a special "available" marker object. With this special marker possibly occupying buckets in our hash table, we modify our search algorithm for erase(k) or find(k) so that the search for a key k skips over cells

containing the available marker and continue probing until reaching the desired entry or an empty bucket (or returning back to where we started). Additionally, our algorithm for put(k, v) should remember an available cell encountered during the search for k, since this is a valid place to put a new entry (k, v). Thus, linear probing saves space, but it complicates removals.

Even with the use of the available marker object, linear probing suffers from an additional disadvantage. It tends to cluster the entries of the map into contiguous runs, which may even overlap (particularly if more than half of the cells in the hash table are occupied). Such contiguous runs of occupied hash cells causes searches to slow down considerably.

Quadratic Probing

Another open-addressing strategy, known as ***quadratic probing***, involves iteratively trying the buckets $A[(i + f(j)) \bmod N]$, for $j = 0, 1, 2, \ldots$, where $f(j) = j^2$, until finding an empty bucket. As with linear probing, the quadratic-probing strategy complicates the removal operation, but it does avoid the kinds of clustering patterns that occur with linear probing. Nevertheless, it creates its own kind of clustering, called ***secondary clustering***, where the set of filled array cells "bounces" around the array in a fixed pattern. If N is not chosen as a prime, then the quadratic-probing strategy may not find an empty bucket in A even if one exists. In fact, even if N is prime, this strategy may not find an empty slot if the bucket array is at least half full. We explore the cause of this type of clustering in an exercise (Exercise C-9.9).

Double Hashing

Another open-addressing strategy that does not cause clustering of the kind produced by linear probing or by quadratic probing is the ***double-hashing*** strategy. In this approach, we choose a secondary hash function, h', and if h maps some key k to a bucket $A[i]$, with $i = h(k)$, that is already occupied, then we iteratively try the buckets $A[(i + f(j)) \bmod N]$ next, for $j = 1, 2, 3, \ldots$, where $f(j) = j \cdot h'(k)$. In this scheme, the secondary hash function is not allowed to evaluate to zero; a common choice is $h'(k) = q - (k \bmod q)$, for some prime number $q < N$. Also, N should be a prime. Moreover, we should choose a secondary hash function that attempts to minimize clustering as much as possible.

These ***open-addressing*** schemes save some space over the separate-chaining method, but they are not necessarily faster. In experimental and theoretical analyses, the chaining method is either competitive or faster than the other methods, depending on the load factor of the bucket array. So, if memory space is not a major issue, the collision-handling method of choice seems to be separate chaining.

9.2.6 Load Factors and Rehashing

In all of the hash-table schemes described above, the load factor, $\lambda = n/N$, should be kept below 1. Experiments and average-case analyses suggest that we should maintain $\lambda < 0.5$ for the open-addressing schemes and we should maintain $\lambda < 0.9$ for separate chaining.

As we explore in Exercise C-9.9, some open-addressing schemes can start to fail when $\lambda \geq 0.5$. Although the details of the average-case analysis of hashing are beyond the scope of this book, its probabilistic basis is quite intuitive. If our hash function is good, then we expect the hash function values to be uniformly distributed in the range $[0, N-1]$. Thus, to store n items in our map, the expected number of keys in a bucket would be $\lceil n/N \rceil$ at most, which is $O(1)$ if n is $O(N)$.

With open addressing, as the load factor λ grows beyond 0.5 and starts approaching 1, clusters of items in the bucket array start to grow as well. These clusters cause the probing strategies to "bounce around" the bucket array for a considerable amount of time before they can finish. At the limit, when λ is close to 1, all map operations have linear expected running times, since, in this case, we expect to encounter a linear number of occupied buckets before finding one of the few remaining empty cells.

Rehashing into a New Table

Keeping the load factor below a certain threshold is vital for open-addressing schemes and is also of concern to the separate-chaining method. If the load factor of a hash table goes significantly above a specified threshold, then it is common to require that the table be resized (to regain the specified load factor) and all the objects inserted into this new resized table. Indeed, if we let our hash table become full, some implementations may crash. When rehashing to a new table, a good requirement is having the new array's size be at least double the previous size. Once we have allocated this new bucket array, we must define a new hash function to go with it (possibly computing new parameters, as in the MAD method). Given this new hash function, we then reinsert every item from the old array into the new array using this new hash function. This process is known as *rehashing*.

Even with periodic rehashing, a hash table is an efficient means of implementing an unordered map. Indeed, if we always double the size of the table with each rehashing operation, then we can amortize the cost of rehashing all the elements in the table against the time used to insert them in the first place. The analysis of this rehashing process is similar to that used to analyze vector growth. (See Section 6.1.3.) Each rehashing generally scatters the elements throughout the new bucket array. Thus, a hash table is a practical and effective implementation for an unordered map.

9.2.7 A C++ Hash Table Implementation

In Code Fragments 9.6 through 9.13, we present a C++ implementation of the map ADT, called HashMap, which is based on hashing with separate chaining. The class is templated with the key type K, the value type V, and the hash comparator type H. The hash comparator defines a function, $hash(k)$, which maps a key into an integer index. As with less-than comparators (see Section 8.1.2), a hash comparator class does this by overriding the "()" operator.

We present the general class structure in Code Fragment 9.6. The definition begins with the public types required by the map interface, the entry type Entry, and the iterator type Iterator. This is followed by the declarations of the public member functions. We then give the private member data, which consists of the number of entries n, the hash comparator function $hash$, and the bucket array B. We have omitted two sections, which are filled in later. The first is a declaration of some utility types and functions and the second is the declaration of the map's iterator class.

```cpp
template <typename K, typename V, typename H>
class HashMap {
public:                                       // public types
  typedef Entry<const K,V> Entry;             // a (key,value) pair
  class Iterator;                             // a iterator/position
public:                                       // public functions
  HashMap(int capacity = 100);                // constructor
  int size() const;                           // number of entries
  bool empty() const;                         // is the map empty?
  Iterator find(const K& k);                  // find entry with key k
  Iterator put(const K& k, const V& v);       // insert/replace (k,v)
  void erase(const K& k);                      // remove entry with key k
  void erase(const Iterator& p);              // erase entry at p
  Iterator begin();                           // iterator to first entry
  Iterator end();                             // iterator to end entry
protected:                                     // protected types
  typedef std::list<Entry> Bucket;            // a bucket of entries
  typedef std::vector<Bucket> BktArray;       // a bucket array
  // ...insert HashMap utilities here
private:
  int n;                                      // number of entries
  H hash;                                     // the hash comparator
  BktArray B;                                 // bucket array
public:                                       // public types
  // ...insert Iterator class declaration here
};
```

Code Fragment 9.6: The class HashMap, which implements the map ADT.

We have defined the key part of Entry to be "const K," rather than "K." This prevents a user from inadvertently modifying a key. The class makes use of two major data types. The first is an STL list of entries, called a Bucket, each storing a single bucket. The other is an STL vector of buckets, called BktArray.

Before describing the main elements of the class, we introduce a few local (protected) utilities in Code Fragment 9.7. We declare three helper functions, finder, inserter, and eraser, which, respectively, handle the low-level details of finding, inserting, and removing entries. For convenience, we define two iterator types, one called BItor for iterating over the buckets of the bucket array, and one called EItor, for iterating over the entries of a bucket. We also give two utility functions, nextBkt and endOfBkt, which are used to iterate through the entries of a single bucket.

```
Iterator finder(const K& k);                    // find utility
Iterator inserter(const Iterator& p, const Entry& e);  // insert utility
void eraser(const Iterator& p);                 // remove utility
typedef typename BktArray::iterator BItor;      // bucket iterator
typedef typename Bucket::iterator EItor;        // entry iterator
static void nextEntry(Iterator& p)              // bucket's next entry
  { ++p.ent; }
static bool endOfBkt(const Iterator& p)         // end of bucket?
  { return p.ent == p.bkt−>end(); }
```

Code Fragment 9.7: Declarations of utilities to be inserted into HashMap.

We present the class Iterator in Code Fragment 9.8. An iterator needs to store enough information about the position of an entry to allow it to navigate. The members *ent*, *bkt*, and *ba* store, respectively, an iterator to the current entry, the bucket containing this entry, and the bucket array containing the bucket. The first two are of types EItor and BItor, respectively, and the third is a pointer. Our implementation is minimal. In addition to a constructor, we provide operators for dereferencing ("*"), testing equality ("=="), and advancing through the map ("++").

```
class Iterator {                                // an iterator (& position)
private:
  EItor ent;                                    // which entry
  BItor bkt;                                    // which bucket
  const BktArray* ba;                           // which bucket array
public:
  Iterator(const BktArray& a, const BItor& b, const EItor& q = EItor())
    : ent(q), bkt(b), ba(&a) { }
  Entry& operator*() const;                     // get entry
  bool operator==(const Iterator& p) const;     // are iterators equal?
  Iterator& operator++();                       // advance to next entry
  friend class HashMap;                         // give HashMap access
};
```

Code Fragment 9.8: Declaration of the Iterator class for HashMap.

Iterator Dereferencing and Condensed Function Definitions

Let us now present the definitions of the class member functions for our map's Iterator class. In Code Fragment 9.9, we present an implementation of the dereferencing operator. The function body itself is very simple and involves returning a reference to the corresponding entry. However, the rules of C++ syntax demand an extraordinary number of template qualifiers. First, we need to qualify the function itself as being a member of HashMap's iterator class, which we do with the qualifier HashMap<K,V,H>::Iterator. Second, we need to qualify the function's return type as being HashMap's entry class, which we do with the qualifier HashMap<K,V,H>::Entry. On top of this, we must recall from Section 8.2.1 that, since we are using a template parameter to define a type, we need to include the keyword **typename**.

```
template <typename K, typename V, typename H> // get entry
typename HashMap<K,V,H>::Entry&
HashMap<K,V,H>::Iterator::operator*() const
  { return *ent; }
```

Code Fragment 9.9: The Iterator dereferencing operator (complete form).

In order to make our function definitions more readable, we adopt a notational convention in some of our future code fragments of specifying the scoping qualifier for the code fragment in italic blue font. We omit this qualifier from the code fragment, and we also omit the template statement and the **typename** specifications. Adding these back is a simple mechanical exercise. Although this is not valid C++ syntax, it conveys the important content in a much more succinct manner. An example of the same dereferencing operator is shown in Code Fragment 9.10.

Caution

```
/* HashMap⟨K,V,H⟩ :: */                              // get entry
  Entry& Iterator::operator*() const
    { return *ent; }
```

Code Fragment 9.10: The same dereferencing operator of Code Fragment 9.9 in condensed form.

Definitions of the Other Iterator Member Functions

Let us next consider the Iterator operator "**operator** $==(p)$," which tests whether this iterator is equal to iterator p. We first check that they belong to the same bucket array and the same bucket within this array. If not, the iterators certainly differ. Otherwise, we check whether they both refer to the end of the bucket array. (Since we have established that the buckets are equal, it suffices to test just one of them.) If so, they are both equal to HashMap::end(). If not, we check whether they both

refer to the same entry of the bucket. This is implemented in Code Fragment 9.11.

```
/* HashMap⟨K,V,H⟩ :: */                              // are iterators equal?
  bool Iterator::operator==(const Iterator& p) const {
    if (ba != p.ba || bkt != p.bkt) return false;    // ba or bkt differ?
    else if (bkt == ba->end()) return true;          // both at the end?
    else return (ent == p.ent);                      // else use entry to decide
  }
```

Code Fragment 9.11: The Iterator operators for equality testing and increment.

Next, let us consider the Iterator increment operator, shown in Code Fragment 9.12. The objective is to advance the iterator to the next valid entry. Typically, this involves advancing to the next entry within the current bucket. But, if we fall off the end of this bucket, we must advance to the first element of the next nonempty bucket. To do this, we first advance to the next bucket entry by applying the STL increment operator on the entry iterator *ent*. We then use the utility function endOfBkt to determine whether we have arrived at the end of this bucket. If so, we search for the next nonempty bucket. To do this, we repeatedly increment *bkt* and check whether we have fallen off the end of the bucket array. If so, this is the end of the map and we are done. Otherwise, we check whether the bucket is empty. When we first find a nonempty bucket, we move *ent* to the first entry of this bucket.

```
/* HashMap⟨K,V,H⟩ :: */                              // advance to next entry
  Iterator& Iterator::operator++() {
    ++ent;                                           // next entry in bucket
    if (endOfBkt(*this)) {                           // at end of bucket?
      ++bkt;                                         // go to next bucket
      while (bkt != ba->end() && bkt->empty())       // find nonempty bucket
        ++bkt;
      if (bkt == ba->end()) return *this;            // end of bucket array?
      ent = bkt->begin();                            // first nonempty entry
    }
    return *this;                                    // return self
  }
```

Code Fragment 9.12: The Iterator operators for equality testing and increment.

Definitions of the HashMap Member Functions

Before discussing the main functions of class HashMap, let us present the functions begin and end. These are given in Code Fragment 9.13. The function end is the simpler of the two. It involves generating an iterator whose bucket component is the end of the bucket array. We do not bother to specify a value for the entry part of the

iterator. The reason is that our iterator equality test (shown in Code Fragment 9.11) does not bother to compare the entry iterator values if the bucket iterators are at the end of the bucket array.

```
/* HashMap⟨K,V,H⟩ :: */                              // iterator to end
  Iterator end()
    { return Iterator(B, B.end()); }
```

```
/* HashMap⟨K,V,H⟩ :: */                              // iterator to front
  Iterator begin() {
    if (empty()) return end();                       // emtpty - return end
    BItor bkt = B.begin();                           // else search for an entry
    while (bkt->empty()) ++bkt;                       // find nonempty bucket
    return Iterator(B, bkt, bkt->begin());           // return first of bucket
  }
```

Code Fragment 9.13: The functions of HashMap returning iterators to the beginning and end of the map.

The function begin, shown in the bottom part of Code Fragment 9.13, is more complex, since we need to search for a nonempty bucket. We first check whether the map is empty. If so, we simply return the map's end. Otherwise, starting at the beginning of the bucket array, we search for a nonempty bucket. (We know we will succeed in finding one.) Once we find it, we return an iterator that points to the first entry of this bucket.

Now that we have presented the iterator-related functions, we are ready to present the functions for class HashMap. We begin with the constructor and simple container functions. The constructor is given the bucket array's capacity and creates a vector of this size. The member n tracks the number of entries. These are given in Code Fragment 9.14.

```
/* HashMap⟨K,V,H⟩ :: */                              // constructor
  HashMap(int capacity) : n(0), B(capacity) { }
```

```
/* HashMap⟨K,V,H⟩ :: */                              // number of entries
  int size() const { return n; }
```

```
/* HashMap⟨K,V,H⟩ :: */                              // is the map empty?
  bool empty() const { return size() == 0; }
```

Code Fragment 9.14: The constructor and standard functions for HashMap.

Next, we present the functions related to finding keys in the top part of Code Fragment 9.15. Most of the work is done by the utility function finder. It first applies the hash function associated with the given hash comparator to the key k. It converts this to an index into the bucket array by taking the hash value modulo

the array size. To obtain an iterator to the desired bucket, we add this index to the beginning iterator of the bucket array. (We are using the fact mentioned in Section 6.1.4 that STL vectors provide a random access iterator, so addition is allowed.) Let *bkt* be an iterator to this bucket. We create an iterator *p*, which is initialized to the beginning of this bucket. We then perform a search for an entry whose key matches *k* or until we fall off the end of the list. In either case, we return the final value of the iterator as the search result.

```
/* HashMap⟨K,V,H⟩ :: */                        // find utility
  Iterator finder(const K& k) {
    int i = hash(k) % B.size();                // get hash index i
    BItor bkt = B.begin() + i;                 // the ith bucket
    Iterator p(B, bkt, bkt->begin());          // start of ith bucket
    while (!endOfBkt(p) && (*p).key() != k)     // search for k
      nextEntry(p);
    return p;                                  // return final position
  }

/* HashMap⟨K,V,H⟩ :: */                        // find key
  Iterator find(const K& k) {
    Iterator p = finder(k);                    // look for k
    if (endOfBkt(p))                           // didn't find it?
      return end();                            // return end iterator
    else
      return p;                                // return its position
  }
```

Code Fragment 9.15: The functions of HashMap related to finding keys.

The public member function find is shown in the bottom part of Code Fragment 9.15. It invokes the finder utility. If the entry component is at the end of the bucket, we know that the key was not found, so we return the special iterator end() to the end of the map. (In this way, all unsuccessful searches produce the same result.) This is shown in Code Fragment 9.15.

The insertion utility, inserter, is shown in the top part of Code Fragment 9.16. This utility is given the desired position at which to insert the new entry. It invokes the STL list insert function to perform the insertion. It also increments the count of the number of entries in the map and returns an iterator to the inserted position.

The public insert function, put, first applies finder to determine whether any entry with this key exists in the map. We first determine whether it was not found by testing whether the iterator as fallen off the end of the bucket. If so, we insert it at the end of this bucket. Otherwise, we modify the existing value of this entry. Later, in Section 9.5.2, we present an alternative approach, which inserts a new entry, even when a duplicate key is discovered.

```
/* HashMap⟨K,V,H⟩ :: */                          // insert utility
  Iterator inserter(const Iterator& p, const Entry& e) {
    EItor ins = p.bkt−>insert(p.ent, e);          // insert before p
    n++;                                          // one more entry
    return Iterator(B, p.bkt, ins);               // return this position
  }

/* HashMap⟨K,V,H⟩ :: */                          // insert/replace (v,k)
  Iterator put(const K& k, const V& v) {
    Iterator p = finder(k);                       // search for k
    if (endOfBkt(p)) {                            // k not found?
      return inserter(p, Entry(k, v));            // insert at end of bucket
    }
    else {                                        // found it?
      p.ent−>setValue(v);                         // replace value with v
      return p;                                   // return this position
    }
  }
```

Code Fragment 9.16: The functions of HashMap for inserting and replacing entries.

The removal functions are also quite straightforward and are given in Code Fragment 9.17. The main utility is the function eraser, which removes an entry at a given position by invoking the STL list erase function. It also decrements the number of entries. The iterator-based removal function simply invokes eraser. The key-based removal function first applies the finder utility to look up the key. If it is not found, that is, if the returned position is the end of the bucket, an exception is thrown. Otherwise, the eraser utility is invoked to remove the entry.

```
/* HashMap⟨K,V,H⟩ :: */                          // remove utility
  void eraser(const Iterator& p) {
    p.bkt−>erase(p.ent);                          // remove entry from bucket
    n−−;                                          // one fewer entry
  }

/* HashMap⟨K,V,H⟩ :: */                          // remove entry at p
  void erase(const Iterator& p)
    { eraser(p); }

/* HashMap⟨K,V,H⟩ :: */                          // remove entry with key k
  void erase(const K& k) {
    Iterator p = finder(k);                       // find k
    if (endOfBkt(p))                              // not found?
      throw NonexistentElement("Erase of nonexistent"); // ...error
    eraser(p);                                    // remove it
  }
```

Code Fragment 9.17: The functions of HashMap involved with removing entries.

9.3   Ordered Maps

In some applications, simply looking up values based on associated keys is not enough. We often also want to keep the entries in a map sorted according to some total order and be able to look up keys and values based on this ordering. That is, in an *ordered map*, we want to perform the usual map operations, but also maintain an order relation for the keys in our map and use this order in some of the map functions. We can use a comparator to provide the order relation among keys, allowing us to define an ordered map relative to this comparator, which can be provided to the ordered map as an argument to its constructor.

When the entries of a map are stored in order, we can provide efficient implementations for additional functions in the map ADT. As with the standard map ADT, in order to indicate that an object is not present, the class provides a special sentinel iterator called end. The ordered map includes all the functions of the standard map ADT plus the following:

firstEntry(k):	Return an iterator to the entry with smallest key value; if the map is empty, it returns end.
lastEntry(k):	Return an iterator to the entry with largest key value; if the map is empty, it returns end.
ceilingEntry(k):	Return an iterator to the entry with the least key value greater than or equal to k; if there is no such entry, it returns end.
floorEntry(k):	Return an iterator to the entry with the greatest key value less than or equal to k; if there is no such entry, it returns end.
lowerEntry(k):	Return an iterator to the entry with the greatest key value less than k; if there is no such entry, it returns end.
higherEntry(k):	Return an iterator to the entry with the least key value greater than k; if there is no such entry, it returns end.

Implementing an Ordered Map

The ordered nature of the operations given above for the ordered map ADT makes the use of an unordered list or a hash table inappropriate, because neither of these data structures maintains any ordering information for the keys in the map. Indeed, hash tables achieve their best search speeds when their keys are distributed almost at random. Thus, we should consider an alternative implementation when dealing with ordered maps. We discuss one such implementation next, and we discuss other implementations in Section 9.4 and Chapter 10.

9.3.1 Ordered Search Tables and Binary Search

If the keys in a map come from a total order, we can store the map's entries in a vector L in increasing order of the keys. (See Figure 9.6.) We specify that L is a vector, rather than a node list, because the ordering of the keys in the vector L allows for faster searching than would be possible had L been, say, implemented with a linked list. Admittedly, a hash table has good expected running time for searching. But its worst-case time for searching is no better than a linked list, and in some applications, such as in real-time processing, we need to guarantee a worst-case searching bound. The fast algorithm for searching in an ordered vector, which we discuss in this subsection, has a good worst-case guarantee on its running time. So it might be preferred over a hash table in certain applications. We refer to this ordered vector implementation of a map as an ***ordered search table***.

0	1	2	3	4	5	6	7	8	9	10
4	6	9	12	15	16	18	28	34		

Figure 9.6: Realization of a map by means of an ordered search table. We show only the keys for this map in order to highlight their ordering.

The space requirement of an ordered search table is $O(n)$, which is similar to the list-based map implementation (Section 9.1.4), assuming we grow and shrink the array supporting the vector L to keep the size of this array proportional to the number of entries in L. Unlike an unordered list, however, performing updates in a search table takes a considerable amount of time. In particular, performing the insert(k,v) operation in a search table requires $O(n)$ time in the worst case, since we need to shift up all the entries in the vector with key greater than k to make room for the new entry (k,v). A similar observation applies to the operation erase(k), since it takes $O(n)$ time in the worst case to shift all the entries in the vector with key greater than k to close the "hole" left by the removed entry (or entries). The search table implementation is therefore inferior to the linked list implementation in terms of the worst-case running times of the map update operations. Nevertheless, we can perform the find function much faster in a search table.

Binary Search

A significant advantage of using an ordered vector L to implement a map with n entries is that accessing an element of L by its ***index*** takes $O(1)$ time. We recall, from Section 6.1, that the index of an element in a vector is the number of elements preceding it. Thus, the first element in L has index 0, and the last element has index $n-1$. In this subsection, we give a classic algorithm, ***binary search***, to locate an entry in an ordered search table. We show how this method can be used

to quickly perform the find function of the map ADT, but a similar method can be used for each of the ordered-map functions, ceilingEntry, floorEntry, lowerEntry, and higherEntry.

The elements stored in L are the entries of a map, and since L is ordered, the entry at index i has a key no smaller than the keys of the entries at indices $0, \ldots, i-1$, and no larger than the keys of the entries at indices $i+1, \ldots, n-1$. This observation allows us to quickly "home in" on a search key k using a variant of the children's game "high-low." We call an entry of our map a ***candidate*** if, at the current stage of the search, we cannot rule out that this entry has key equal to k. The algorithm maintains two parameters, *low* and *high*, such that all the candidate entries have index at least *low* and at most *high* in L. Initially, $low = 0$ and $high = n-1$. We then compare k to the key of the median candidate e, that is, the entry e with index

$$mid = \lfloor (low + high)/2 \rfloor.$$

We consider three cases:
- If $k = e.\text{key}()$, then we have found the entry we were looking for, and the search terminates successfully returning e
- If $k < e.\text{key}()$, then we recur on the first half of the vector, that is, on the range of indices from *low* to $mid - 1$
- If $k > e.\text{key}()$, we recur on the range of indices from $mid + 1$ to *high*

This search method is called ***binary search***, and is given in pseudo-code in Code Fragment 9.18. Operation find(k) on an n-entry map implemented with an ordered vector L consists of calling BinarySearch$(L, k, 0, n-1)$.

Algorithm BinarySearch$(L, k, low, high)$:

    Input: An ordered vector L storing n entries and integers *low* and *high*

    Output: An entry of L with key equal to k and index between *low* and *high*, if such an entry exists, and otherwise the special sentinel end

    if *low* > *high* **then**

      return end

    else

      $mid \leftarrow \lfloor (low + high)/2 \rfloor$

      $e \leftarrow L.\text{at}(mid)$

      if $k = e.\text{key}()$ **then**

        return e

      else if $k < e.\text{key}()$ **then**

        return BinarySearch$(L, k, low, mid - 1)$

      else

        return BinarySearch$(L, k, mid + 1, high)$

      Code Fragment 9.18: Binary search in an ordered vector.

We illustrate the binary search algorithm in Figure 9.7.
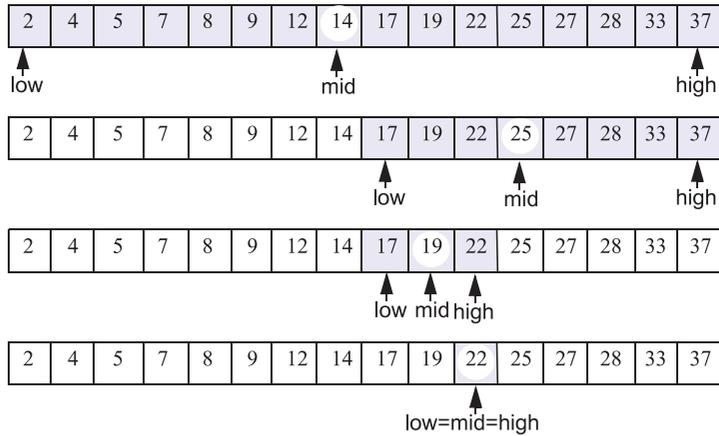
Figure 9.7: Example of a binary search to perform operation find(22), in a map with integer keys, implemented with an ordered vector. For simplicity, we show the keys, not the whole entries.

Considering the running time of binary search, we observe that a constant number of primitive operations are executed at each recursive call of function Binary-Search. Hence, the running time is proportional to the number of recursive calls performed. A crucial fact is that with each recursive call the number of candidate entries still to be searched in the vector L is given by the value

$$high - low + 1.$$

Moreover, the number of remaining candidates is reduced by at least one half with each recursive call. Specifically, from the definition of mid, the number of remaining candidates is either

$$(mid - 1) - low + 1 = \left\lfloor \frac{low + high}{2} \right\rfloor - low \leq \frac{high - low + 1}{2}$$

or

$$high - (mid + 1) + 1 = high - \left\lfloor \frac{low + high}{2} \right\rfloor \leq \frac{high - low + 1}{2}.$$

Initially, the number of candidate entries is n; after the first call to BinarySearch, it is at most $n/2$; after the second call, it is at most $n/4$; and so on. In general, after the ith call to BinarySearch, the number of candidate entries remaining is at most $n/2^i$. In the worst case (unsuccessful search), the recursive calls stop when there are no more candidate entries. Hence, the maximum number of recursive calls performed, is the smallest integer m, such that

$$n/2^m < 1.$$

In other words (recalling that we omit a logarithm's base when it is 2), $m > \log n$. Thus, we have

$$m = \lfloor \log n \rfloor + 1,$$

which implies that binary search runs in $O(\log n)$ time.

Thus, we can use an ordered search table to perform fast searches in an ordered map, but using such a table for lots of map updates would take a considerable amount of time. For this reason, the primary applications for search tables are in situations where we expect few updates but many searches. Such a situation could arise, for example, in an ordered list of English words we use to order entries in an encyclopedia or help file.

Comparing Map Implementations

Note that we can use an ordered search table to implement the map ADT even if we don't want to use the additional functions of the ordered map ADT. Table 9.2 compares the running times of the functions of a (standard) map realized by either an unordered list, a hash table, or an ordered search table. Note that an unordered list allows for fast insertions but slow searches and removals, whereas a search table allows for fast searches but slow insertions and removals. Incidentally, although we don't explicitly discuss it, we note that a sorted list implemented with a doubly linked list would be slow in performing almost all the map operations. (See Exercise R-9.5.) Nevertheless, the list-like data structure we discuss in the next section can perform the functions of the ordered map ADT quite efficiently.

Method	*List*	*Hash Table*	*Search Table*
size, empty	$O(1)$	$O(1)$	$O(1)$
find	$O(n)$	$O(1)$ exp., $O(n)$ worst-case	$O(\log n)$
insert	$O(1)$	$O(1)$	$O(n)$
erase	$O(n)$	$O(1)$ exp., $O(n)$ worst-case	$O(n)$

Table 9.2: Comparison of the running times of the functions of a map realized by means of an unordered list, a hash table, or an ordered search table. We let n denote the number of entries in the map and we let N denote the capacity of the bucket array in the hash table implementation. The space requirement of all the implementations is $O(n)$, assuming that the arrays supporting the hash-table and search-table implementations are maintained such that their capacity is proportional to the number of entries in the map.

9.3.2 Two Applications of Ordered Maps

As we have mentioned in the preceding sections, unordered and ordered maps have many applications. In this section, we explore some specific applications of ordered maps.

Flight Databases

There are several web sites on the Internet that allow users to perform queries on flight databases to find flights between various cities, typically with the intent of buying a ticket. To make a query, a user specifies origin and destination cities, a departure date, and a departure time. To support such queries, we can model the flight database as a map, where keys are Flight objects that contain fields corresponding to these four parameters. That is, a key is a ***tuple***

$$k = (origin, \ destination, \ date, \ time).$$

Additional information about a flight, such as the flight number, the number of seats still available in first (F) and coach (Y) class, the flight duration, and the fare, can be stored in the value object.

Finding a requested flight is not simply a matter of finding a key in the map matching the requested query, however. The main difficulty is that, although a user typically wants to exactly match the origin and destination cities, as well as the departure date, he or she will probably be content with any departure time that is close to his or her requested departure time. We can handle such a query, of course, by ordering our keys lexicographically. Thus, given a user query key k, we could, for instance, call ceilingEntry(k) to return the flight between the desired cities on the desired date, with departure time at the desired time or after. A similar use of floorEntry(k) would give us the flight with departure time at the desired time or before. Given these entries, we could then use the higherEntry or lowerEntry functions to find flights with the next close-by departure times that are respectively higher or lower than the desired time, k. Therefore, an efficient implementation for an ordered map would be a good way to satisfy such queries. For example, calling ceilingEntry(k) on a query key $k = $ (ORD, PVD, 05May, 09:30), followed by the respective calls to higherEntry, might result in the following sequence of entries:

```
( (ORD, PVD, 05May, 09:53),   (AA 1840, F5, Y15, 02:05, $251) )
( (ORD, PVD, 05May, 13:29),   (AA 600, F2, Y0, 02:16, $713) )
( (ORD, PVD, 05May, 17:39),   (AA 416, F3, Y9, 02:09, $365) )
( (ORD, PVD, 05May, 19:50),   (AA 1828, F9, Y25, 02:13, $186) )
```

Maxima Sets

Life is full of trade-offs. We often have to trade off a desired performance measure against a corresponding cost. Suppose, for the sake of an example, we are interested in maintaining a database rating automobiles by their maximum speeds and their cost. We would like to allow someone with a certain amount to spend to query our database to find the fastest car they can possibly afford.

We can model such a trade-off problem as this by using a key-value pair to model the two parameters that we are trading off, which in this case would be the pair (*cost*, *speed*) for each car. Notice that some cars are strictly better than other cars using this measure. For example, a car with cost-speed pair (20,000, 100) is strictly better than a car with cost-speed pair (30,000, 90). At the same time, there are some cars that are not strictly dominated by another car. For example, a car with cost-speed pair (20,000, 100) may be better or worse than a car with cost-speed pair (30,000, 120), depending on how much money we have to spend. (See Figure 9.8.)
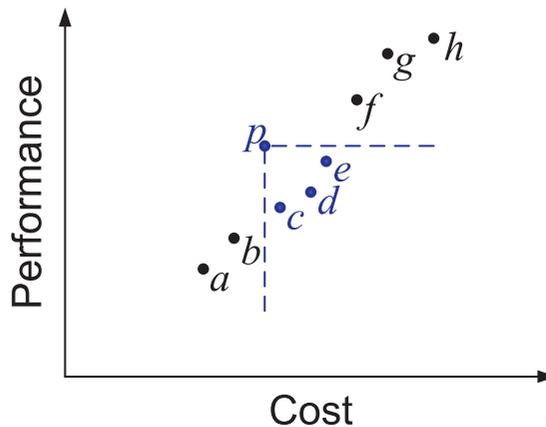
Figure 9.8: The cost-performance trade-off with key-value pairs represented by points in the plane. Notice that point p is strictly better than points c, d, and e, but may be better or worse than points a, b, f, g, and h, depending on the price we are willing to pay. Thus, if we were to add p to our set, we could remove the points c, d, and e, but not the others.

Formally, we say a price-performance pair (a, b) ***dominates*** a pair (c, d) if $a < c$ and $b > d$. A pair (a, b) is called a ***maximum*** pair if it is not dominated by any other pairs. We are interested in maintaining the set of maxima of a collection C of price-performance pairs. That is, we would like to add new pairs to this collection (for example, when a new car is introduced), and we would like to query this collection for a given dollar amount, d, to find the fastest car that costs no more than d dollars.

Maintaining a Maxima Set with an Ordered Map

We can store the set of maxima pairs in an ordered map, M, ordered by cost, so that the cost is the key field and performance (speed) is the value field. We can then implement operations $\mathsf{add}(c,p)$, which adds a new cost-performance pair (c,p), and $\mathsf{best}(c)$, which returns the best pair with cost at most c, as shown in Code Fragments 9.19 and 9.20.

Algorithm $\mathsf{best}(c)$:
  Input: A cost c
  Output: The cost-performance pair in M with largest cost less than or equal to
   c, or the special sentinel end, if there is no such pair
   return $M.\mathsf{floorEntry}(c)$

Code Fragment 9.19: The $\mathsf{best}()$ function, used in a class maintaining a set of maxima implemented with an ordered map M.

Algorithm $\mathsf{add}(c,p)$:
  Input: A cost-performance pair (c,p)
  Output: None (but M will have (c,p) added to the set of cost-performance
   pairs)
   $e \leftarrow M.\mathsf{floorEntry}(c)$ {the greatest pair with cost at most c}
   if $e \neq$ end **then**
    if $e.\mathsf{value}() > p$ **then**
     return   {(c,p) is dominated, so don't insert it in M}
   $e \leftarrow M.\mathsf{ceilingEntry}(c)$ {next pair with cost at least c}
   {Remove all the pairs that are dominated by (c,p)}
   while $e \neq$ end **and** $e.\mathsf{value}() < p$ **do**
    $M.\mathsf{erase}(e.\mathsf{key}())$ {this pair is dominated by (c,p)}
    $e \leftarrow M.\mathsf{higherEntry}(e.\mathsf{key}())$ {the next pair after e}
   $M.\mathsf{put}(c,p)$ {Add the pair (c,p), which is not dominated}

Code Fragment 9.20: The $\mathsf{add}(c,p)$ function used in a class for maintaining a set of maxima implemented with an ordered map M.

Unfortunately, if we implement M using any of the data structures described above, it results in a poor running time for the above algorithm. If, on the other hand, we implement M using a skip list, which we describe next, then we can perform $\mathsf{best}(c)$ queries in $O(\log n)$ expected time and $\mathsf{add}(c,p)$ updates in $O((1 + r)\log n)$ expected time, where r is the number of points removed.

9.4   Skip Lists

An interesting data structure for efficiently realizing the ordered map ADT is the *skip list*. This data structure makes random choices in arranging the entries in such a way that search and update times are $O(\log n)$ *on average*, where n is the number of entries in the dictionary. Interestingly, the notion of average time complexity used here does not depend on the probability distribution of the keys in the input. Instead, it depends on the use of a random-number generator in the implementation of the insertions to help decide where to place the new entry. The running time is averaged over all possible outcomes of the random numbers used when inserting entries.

Because they are used extensively in computer games, cryptography, and computer simulations, functions that generate numbers that can be viewed as random numbers are built into most modern computers. Some functions, called *pseudo-random number generators*, generate random-like numbers, starting with an initial *seed*. Other functions use hardware devices to extract "true" random numbers from nature. In any case, we assume that our computer has access to numbers that are sufficiently random for our analysis.

The main advantage of using *randomization* in data structure and algorithm design is that the structures and functions that result are usually simple and efficient. We can devise a simple randomized data structure, called the skip list, which has the same logarithmic time bounds for searching as is achieved by the binary searching algorithm. Nevertheless, the bounds are *expected* for the skip list, while they are *worst-case* bounds for binary searching in a lookup table. On the other hand, skip lists are much faster than lookup tables for map updates.

A *skip list* S for a map M consists of a series of lists $\{S_0, S_1, \ldots, S_h\}$. Each list S_i stores a subset of the entries of M sorted by increasing keys plus entries with two special keys, denoted $-\infty$ and $+\infty$, where $-\infty$ is smaller than every possible key that can be inserted in M and $+\infty$ is larger than every possible key that can be inserted in M. In addition, the lists in S satisfy the following:

- List S_0 contains every entry of the map M (plus the special entries with keys $-\infty$ and $+\infty$)
- For $i = 1, \ldots, h-1$, list S_i contains (in addition to $-\infty$ and $+\infty$) a randomly generated subset of the entries in list S_{i-1}
- List S_h contains only $-\infty$ and $+\infty$.

An example of a skip list is shown in Figure 9.9. It is customary to visualize a skip list S with list S_0 at the bottom and lists $S_1, \ldots, S_h$ above it. Also, we refer to h as the *height* of skip list S.
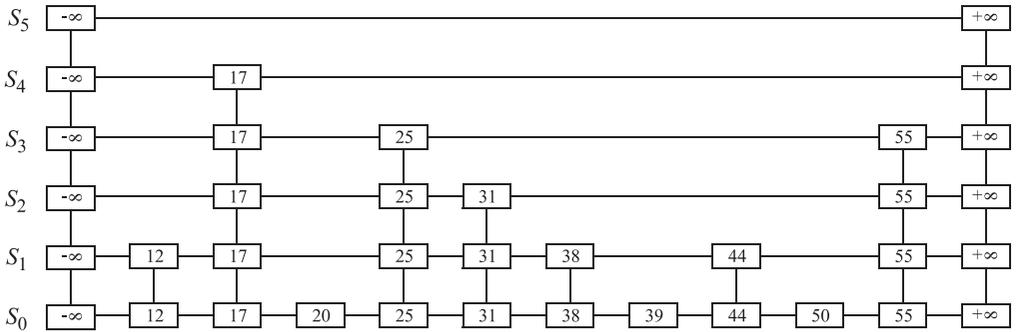
Figure 9.9: Example of a skip list storing 10 entries. For simplicity, we show only the keys of the entries.

Intuitively, the lists are set up so that S_{i+1} contains more or less every other entry in S_i. As can be seen in the details of the insertion method, the entries in S_{i+1} are chosen at random from the entries in S_i by picking each entry from S_i to also be in S_{i+1} with probability $1/2$. That is, in essence, we "flip a coin" for each entry in S_i and place that entry in S_{i+1} if the coin comes up "heads." Thus, we expect S_1 to have about $n/2$ entries, S_2 to have about $n/4$ entries, and, in general, S_i to have about $n/2^i$ entries. In other words, we expect the height h of S to be about $\log n$. The halving of the number of entries from one list to the next is not enforced as an explicit property of skip lists, however. Instead, randomization is used.

Using the position abstraction used for lists and trees, we view a skip list as a two-dimensional collection of positions arranged horizontally into *levels* and vertically into *towers*. Each level is a list S_i and each tower contains positions storing the same entry across consecutive lists. The positions in a skip list can be traversed using the following operations:

after(p): Return the position following p on the same level.

before(p): Return the position preceding p on the same level.

below(p): Return the position below p in the same tower.

above(p): Return the position above p in the same tower.

We conventionally assume that the above operations return a null position if the position requested does not exist. Without going into the details, we note that we can easily implement a skip list by means of a linked structure such that the above traversal functions each take $O(1)$ time, given a skip-list position p. Such a linked structure is essentially a collection of h doubly linked lists aligned at towers, which are also doubly linked lists.

9.4.1   Search and Update Operations in a Skip List

The skip list structure allows for simple map search and update algorithms. In fact, all of the skip list search and update algorithms are based on an elegant SkipSearch function that takes a key k and finds the position p of the entry e in list S_0 such that e has the largest key (which is possibly $-\infty$) less than or equal to k.

Searching in a Skip List

Suppose we are given a search key k. We begin the SkipSearch function by setting a position variable p to the top-most, left position in the skip list S, called the ***start position*** of S. That is, the start position is the position of S_h storing the special entry with key $-\infty$. We then perform the following steps (see Figure 9.10), where key(p) denotes the key of the entry at position p:

1. If S.below(p) is *null*, then the search terminates—we are ***at the bottom*** and have located the largest entry in S with key less than or equal to the search key k. Otherwise, we ***drop down*** to the next lower level in the present tower by setting $p \leftarrow S$.below(p).
2. Starting at position p, we move p forward until it is at the right-most position on the present level such that key$(p) \leq k$. We call this the ***scan forward*** step. Note that such a position always exists, since each level contains the keys $+\infty$ and $-\infty$. In fact, after we perform the scan forward for this level, p may remain where it started. In any case, we then repeat the previous step.
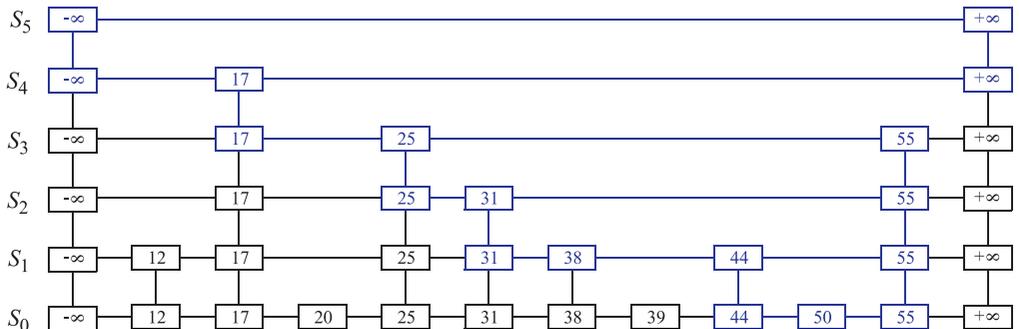
Figure 9.10: Example of a search in a skip list. The positions visited when searching for key 50 are highlighted in blue.

We give a pseudo-code description of the skip-list search algorithm, SkipSearch, in Code Fragment 9.21. Given this function, it is now easy to implement the operation find(k)—we simply perform $p \leftarrow$ SkipSearch(k) and test whether or not key$(p) = k$. If these two keys are equal, we return p; otherwise, we return *null*.

Algorithm SkipSearch(k):

    Input: A search key k

    Output: Position p in the bottom list S_0 such that the entry at p has the largest
    key less than or equal to k

    $p \leftarrow s$

    while below$(p) \neq null$ **do**

        $p \leftarrow$ below(p) {drop down}

        while $k \geq$ key(after(p)) **do**

            $p \leftarrow$ after(p) {scan forward}

    return p.

Code Fragment 9.21: Search in a skip list S. Variable s holds the start position of S.

As it turns out, the expected running time of algorithm SkipSearch on a skip list with n entries is $O(\log n)$. We postpone the justification of this fact, however, until after we discuss the implementation of the update functions for skip lists.

Insertion in a Skip List

The insertion algorithm for skip lists uses randomization to decide the height of the tower for the new entry. We begin the insertion of a new entry (k, v) by performing a SkipSearch(k) operation. This gives us the position p of the bottom-level entry with the largest key less than or equal to k (note that p may hold the special entry with key $-\infty$). We then insert (k, v) immediately after position p. After inserting the new entry at the bottom level, we "flip" a coin. If the flip comes up tails, then we stop here. Else (the flip comes up heads), we backtrack to the previous (next higher) level and insert (k, v) in this level at the appropriate position. We again flip a coin; if it comes up heads, we go to the next higher level and repeat. Thus, we continue to insert the new entry (k, v) in lists until we finally get a flip that comes up tails. We link together all the references to the new entry (k, v) created in this process to create the tower for the new entry. A coin flip can be simulated with C++'s built-in, pseudo-random number generator by testing whether a random integer is even or odd.

We give the insertion algorithm for a skip list S in Code Fragment 9.22 and we illustrate it in Figure 9.11. The algorithm uses function insertAfterAbove$(p, q, (k, v))$ that inserts a position storing the entry (k, v) after position p (on the same level as p) and above position q, returning the position r of the new entry (and setting internal references so that after, before, above, and below functions work correctly for p, q, and r). The expected running time of the insertion algorithm on a skip list with n entries is $O(\log n)$, which we show in Section 9.4.2.

Algorithm SkipInsert(k,v):

    Input: Key k and value v

    Output: Topmost position of the entry inserted in the skip list

    $p \leftarrow$ SkipSearch(k)

    $q \leftarrow null$

    $e \leftarrow (k,v)$

    $i \leftarrow -1$

    repeat

      $i \leftarrow i+1$

      if $i \geq h$ **then**

        $h \leftarrow h+1$ {add a new level to the skip list}

        $t \leftarrow$ after(s)

        $s \leftarrow$ insertAfterAbove($null,s,(-\infty,null)$)

        insertAfterAbove($s,t,(+\infty,null)$)

      while above(p) $= null$ **do**

        $p \leftarrow$ before(p)   {scan backward}

      $p \leftarrow$ above(p)   {jump up to higher level}

      $q \leftarrow$ insertAfterAbove(p,q,e) {add a position to the tower of the new entry}

    until coinFlip() $=$ tails

    $n \leftarrow n+1$

    return q

Code Fragment 9.22: Insertion in a skip list. Method coinFlip() returns "heads" or "tails," each with probability $1/2$. Variables n, h, and s hold the number of entries, the height, and the start node of the skip list.
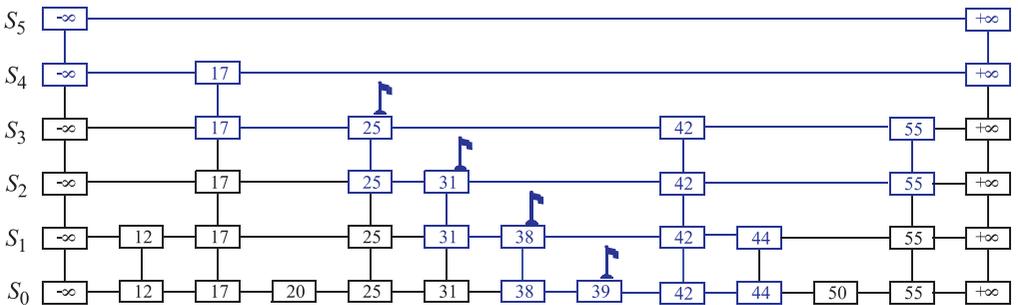
Figure 9.11: Insertion of an entry with key 42 into the skip list of Figure 9.9. We assume that the random "coin flips" for the new entry came up heads three times in a row, followed by tails. The positions visited are highlighted in blue. The positions inserted to hold the new entry are drawn with thick lines, and the positions preceding them are flagged.

Removal in a Skip List

Like the search and insertion algorithms, the removal algorithm for a skip list is quite simple. In fact, it is even easier than the insertion algorithm. That is, to perform an erase(k) operation, we begin by executing function SkipSearch(k). If the position p stores an entry with key different from k, we return *null*. Otherwise, we remove p and all the positions above p, which are easily accessed by using above operations to climb up the tower of this entry in S starting at position p. The removal algorithm is illustrated in Figure 9.12 and a detailed description of it is left as an exercise (Exercise R-9.17). As we show in the next subsection, operation erase in a skip list with n entries has $O(\log n)$ expected running time.

Before we give this analysis, however, there are some minor improvements to the skip list data structure we would like to discuss. First, we don't actually need to store references to entries at the levels of the skip list above the bottom level, because all that is needed at these levels are references to keys. Second, we don't actually need the above function. In fact, we don't need the before function either. We can perform entry insertion and removal in strictly a top-down, scan-forward fashion, thus saving space for "up" and "prev" references. We explore the details of this optimization in Exercise C-9.10. Neither of these optimizations improve the asymptotic performance of skip lists by more than a constant factor, but these improvements can, nevertheless, be meaningful in practice. In fact, experimental evidence suggests that optimized skip lists are faster in practice than AVL trees and other balanced search trees, which are discussed in Chapter 10.

The expected running time of the removal algorithm is $O(\log n)$, which we show in Section 9.4.2.
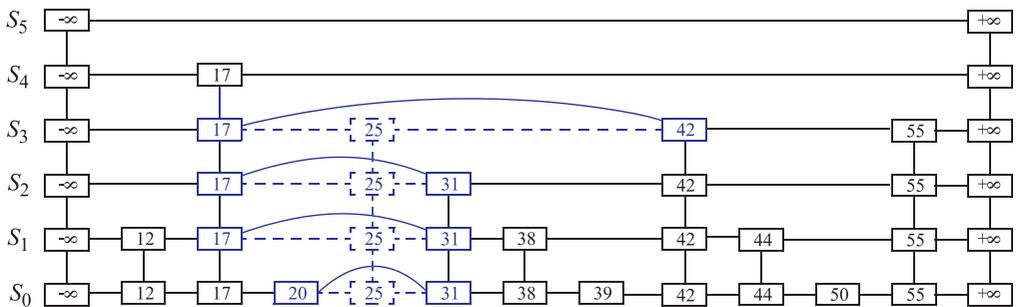
Figure 9.12: Removal of the entry with key 25 from the skip list of Figure 9.11. The positions visited after the search for the position of S_0 holding the entry are highlighted in blue. The positions removed are drawn with dashed lines.

Maintaining the Top-most Level

A skip list S must maintain a reference to the start position (the top-most, left position in S) as a member variable, and must have a policy for any insertion that wishes to continue inserting a new entry past the top level of S. There are two possible courses of action we can take, both of which have their merits.

One possibility is to restrict the top level, h, to be kept at some fixed value that is a function of n, the number of entries currently in the map (from the analysis we see that $h = \max\{10, 2\lceil \log n \rceil\}$ is a reasonable choice, and picking $h = 3\lceil \log n \rceil$ is even safer). Implementing this choice means that we must modify the insertion algorithm to stop inserting a new position once we reach the top-most level (unless $\lceil \log n \rceil < \lceil \log(n+1) \rceil$, in which case we can now go at least one more level, since the bound on the height is increasing).

The other possibility is to let an insertion continue inserting a new position as long as heads keeps getting returned from the random number generator. This is the approach taken in Algorithm SkipInsert of Code Fragment 9.22. As we show in the analysis of skip lists, the probability that an insertion will go to a level that is more than $O(\log n)$ is very low, so this design choice should also work.

Either choice still results in the expected $O(\log n)$ time to perform search, insertion, and removal, however, which we show in the next section.

9.4.2 A Probabilistic Analysis of Skip Lists ⋆

As we have shown above, skip lists provide a simple implementation of an ordered map. In terms of worst-case performance, however, skip lists are not a superior data structure. In fact, if we don't officially prevent an insertion from continuing significantly past the current highest level, then the insertion algorithm can go into what is almost an infinite loop (it is not actually an infinite loop, however, since the probability of having a fair coin repeatedly come up heads forever is 0). Moreover, we cannot infinitely add positions to a list without eventually running out of memory. In any case, if we terminate position insertion at the highest level h, then the ***worst-case*** running time for performing the find, insert, and erase operations in a skip list S with n entries and height h is $O(n+h)$. This worst-case performance occurs when the tower of every entry reaches level $h-1$, where h is the height of S. However, this event has very low probability. Judging from this worst case, we might conclude that the skip list structure is strictly inferior to the other map implementations discussed earlier in this chapter. But this would not be a fair analysis because this worst-case behavior is a gross overestimate.

Bounding the Height of a Skip List

Because the insertion step involves randomization, a more accurate analysis of skip lists involves a bit of probability. At first, this might seem like a major undertaking, since a complete and thorough probabilistic analysis could require deep mathematics (and, indeed, there are several such deep analyses that have appeared in the research literature related to data structures). Fortunately, such an analysis is not necessary to understand the expected asymptotic behavior of skip lists. The informal and intuitive probabilistic analysis we give below uses only basic concepts of probability theory.

Let us begin by determining the expected value of the height h of a skip list S with n entries (assuming that we do not terminate insertions early). The probability that a given entry has a tower of height $i \geq 1$ is equal to the probability of getting i consecutive heads when flipping a coin, that is, this probability is $1/2^i$. Hence, the probability P_i that level i has at least one position is at most

$$P_i \leq \frac{n}{2^i},$$

because the probability that any one of n different events occurs is at most the sum of the probabilities that each occurs.

The probability that the height h of S is larger than i is equal to the probability that level i has at least one position, that is, it is no more than P_i. This means that h is larger than, say, $3 \log n$ with probability at most

$$
\begin{aligned}
P_{3 \log n} &\leq \frac{n}{2^{3 \log n}} \\
&= \frac{n}{n^3} = \frac{1}{n^2}.
\end{aligned}
$$

For example, if $n = 1000$, this probability is a one-in-a-million long shot. More generally, given a constant $c > 1$, h is larger than $c \log n$ with probability at most $1/n^{c-1}$. That is, the probability that h is smaller than $c \log n$ is at least $1 - 1/n^{c-1}$. Thus, with high probability, the height h of S is $O(\log n)$.

Analyzing Search Time in a Skip List

Next, consider the running time of a search in skip list S, and recall that such a search involves two nested **while** loops. The inner loop performs a scan forward on a level of S as long as the next key is no greater than the search key k, and the outer loop drops down to the next level and repeats the scan forward iteration. Since the height h of S is $O(\log n)$ with high probability, the number of drop-down steps is $O(\log n)$ with high probability.

So we have yet to bound the number of scan-forward steps we make. Let n_i be the number of keys examined while scanning forward at level i. Observe that, after the key at the starting position, each additional key examined in a scan-forward at level i cannot also belong to level $i + 1$. If any of these keys were on the previous level, we would have encountered them in the previous scan-forward step. Thus, the probability that any key is counted in n_i is $1/2$. Therefore, the expected value of n_i is exactly equal to the expected number of times we must flip a fair coin before it comes up heads. This expected value is 2.

Hence, the expected amount of time spent scanning forward at any level i is $O(1)$. Since S has $O(\log n)$ levels with high probability, a search in S takes expected time $O(\log n)$. By a similar analysis, we can show that the expected running time of an insertion or a removal is $O(\log n)$.

Space Usage in a Skip List

Finally, let us turn to the space requirement of a skip list S with n entries. As we observed above, the expected number of positions at level i is $n/2^i$, which means that the expected total number of positions in S is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i}.$$

Using Proposition 4.5 on geometric summations, we have

$$\sum_{i=0}^{h} \frac{1}{2^i} = \frac{\left(\frac{1}{2}\right)^{h+1} - 1}{\frac{1}{2} - 1} = 2 \cdot \left(1 - \frac{1}{2^{h+1}}\right) < 2 \quad \text{for all } h \geq 0.$$

Hence, the expected space requirement of S is $O(n)$.

Table 9.3 summarizes the performance of an ordered map realized by a skip list.

Operation	Time
size, empty	$O(1)$
firstEntry, lastEntry	$O(1)$
find, insert, erase	$O(\log n)$ (expected)
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(\log n)$ (expected)

Table 9.3: Performance of an ordered map implemented with a skip list. We use n to denote the number of entries in the dictionary at the time the operation is performed. The expected space requirement is $O(n)$.

9.5 Dictionaries

Like a map, a dictionary stores key-value pairs (k, v), which we call ***entries***, where k is the key and v is the value. Similarly, a dictionary allows for keys and values to be of any object type. But, whereas a map insists that entries have unique keys, a dictionary allows for multiple entries to have the same key, much like an English dictionary, which allows for multiple definitions for the same word.

The ability to store multiple entries with the same key has several applications. For example, we might want to store records for computer science authors indexed by their first and last names. Since there are a few cases of different authors with the same first and last name, there will naturally be some instances where we have to deal with different entries having equal keys. Likewise, a multi-user computer game involving players visiting various rooms in a large castle might need a mapping from rooms to players. It is natural in this application to allow users to be in the same room simultaneously, however, to engage in battles. Thus, this game would naturally be another application where it would be useful to allow for multiple entries with equal keys.

9.5.1 The Dictionary ADT

The ***dictionary*** ADT is quite similar to the map ADT, which was presented in Section 9.1. The principal differences involve the issue of multiple values sharing a common key. As with the map ADT, we assume that there is an object, called Iterator, that provides a way to reference entries of the dictionary. There is a special sentinel value, end, which is used to indicate a nonexistent entry. The iterator may be incremented from entry to entry, making it possible to enumerate entries from the collection.

As an ADT, a (unordered) dictionary D supports the following functions:

size(): Return the number of entries in D.

empty(): Return true if D is empty and false otherwise.

find(k): If D contains an entry with key equal to k, then return an iterator p referring any such entry, else return the special iterator end.

findAll(k): Return a pair of iterators (b, e), such that all the entries with key value k lie in the range from b up to, but not including, e.

insert(k, v): Insert an entry with key k and value v into D, returning an iterator referring to the newly created entry.

erase(k): Remove from D an arbitrary entry with key equal to k; an error condition occurs if D has no such entry.

erase(p): Remove from D the entry referenced by iterator p; an error condition occurs if p points to the end sentinel.

begin(): Return an iterator to the first entry of D.

end(): Return an iterator to a position just beyond the end of D.

Note that operation find(k) returns an ***arbitrary*** entry, whose key is equal to k, and erase(k) removes an arbitrary entry with key value k. In order to remove a specific entry among those having the same key, it would be necessary to remember the iterator value p returned by insert(k,v), and then use the operation erase(p).

Example 9.2: *In the following, we show a series of operations on an initially empty dictionary storing entries with integer keys and character values. In the column "Output," we use the notation $p_i : [(k,v)]$ to mean that the operation returns an iterator denoted by p_i that refers to the entry (k,v).*

Although the entries are not necessarily stored in any particular order, in order to implement the operation findAll, *we assume that items with the same keys are stored contiguously. (Alternatively, the operation* findAll *would need to return a smarter form of iterator that returns keys of equal value.)*

Operation	*Output*	*Dictionary*
empty()	**true**	$\emptyset$
insert($5,A$)	$p_1 : [(5,A)]$	$\{(5,A)\}$
insert($7,B$)	$p_2 : [(7,B)]$	$\{(5,A),(7,B)\}$
insert($2,C$)	$p_3 : [(2,C)]$	$\{(5,A),(7,B),(2,C)\}$
insert($8,D$)	$p_4 : [(8,D)]$	$\{(5,A),(7,B),(2,C),(8,D)\}$
insert($2,E$)	$p_5 : [(2,E)]$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(7)	$p_2 : [(7,B)]$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(4)	end	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(2)	$p_3 : [(2,C)]$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
findAll(2)	(p_3,p_4)	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
size()	5	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
erase(5)	–	$\{(7,B),(2,C),(2,E),(8,D)\}$
erase(p_3)	–	$\{(7,B),(2,E),(8,D)\}$
find(2)	$p_5 : [(2,E)]$	$\{(7,B),(2,E),(8,D)\}$

The operation findAll(2) *returns the iterator pair* (p_3,p_4), *referring to the entries* $(2,C)$ *and* $(8,D)$. *Assuming that the entries are stored in the order listed above, iterating from* p_3 *up to, but not including,* p_4, *would enumerate the entries* $\{(2,C),(2,E)\}$.

9.5.2   A C++ Dictionary Implementation

In this Section, we describe a C++ implementation of the dictionary ADT. Our implementation, called HashDict, is a subclass of the HashMap class, from Section 9.2.7. The map ADT already includes most of the functions of the dictionary ADT. Our HashDict class implements the new function insert, which inserts a key-value pair, and the function findAll, which generates an iterator range for all the values equal to a given key. All the other functions are inherited from HashMap.

In order to support the return type of findAll, we define a nested class called Range. It is presented in Code Fragment 9.23. This simple class stores a pair of objects of type Iterator, a constructor, and two member functions for accessing each of them. This definition will be nested inside the public portion of the HashMap class definition.

```
class Range {                                  // an iterator range
private:
  Iterator _begin;                             // front of range
  Iterator _end;                               // end of range
public:
  Range(const Iterator& b, const Iterator& e)  // constructor
    : _begin(b), _end(e) { }
  Iterator& begin() { return _begin; }         // get beginning
  Iterator& end() { return _end; }             // get end
};
```

Code Fragment 9.23: Definition of the Range class to be added to HashMap.

The HashDict class definition is presented in Code Fragment 9.24. As indicated in the first line of the declaration, this is a subclass of HashMap. The class begins with type definitions for the Iterator and Entry types from the base class. This is followed by the code for class Range from Code Fragment 9.23, and the public function declarations.

```
template <typename K, typename V, typename H>
class HashDict : public HashMap<K,V,H> {
public:                                        // public functions
  typedef typename HashMap<K,V,H>::Iterator Iterator;
  typedef typename HashMap<K,V,H>::Entry Entry;
  // ...insert Range class declaration here
public:                                        // public functions
  HashDict(int capacity = 100);                // constructor
  Range findAll(const K& k);                   // find all entries with k
  Iterator insert(const K& k, const V& v);     // insert pair (k,v)
};
```

Code Fragment 9.24: The class HashDict, which implements the dictionary ADT.

Observe that, when referring to the parent class, HashMap, we need to specify its template parameters. To avoid the need for continually repeating these parameters, we have provided type definitions for the iterator and entry classes. Because most of the dictionary ADT functions are already provided by HashMap, we need only provide a constructor and the missing dictionary ADT functions.

The constructor definition is presented in Code Fragment 9.25. It simply invokes the constructor for the base class. Note that we employ the condensed function notation that we introduced in Section 9.2.7.

```
/* HashDict⟨K,V,H⟩ :: */                                    // constructor
  HashDict(int capacity) : HashMap<K,V,H>(capacity) { }
```

Code Fragment 9.25: The class HashDict constructor.

In Code Fragment 9.26, we present an implementation of the function insert. It first locates the key by invoking the finder utility (see Code Fragment 9.15). Recall that this utility returns an iterator to an entry containing this key, if found, and otherwise it returns an iterator to the end of the bucket. In either case, we insert the new entry immediately prior to this location by invoking the inserter utility. (See Code Fragment 9.16.) An iterator referencing the resulting location is returned.

```
/* HashDict⟨K,V,H⟩ :: */                                    // insert pair (k,v)
  Iterator insert(const K& k, const V& v) {
    Iterator p = finder(k);                                 // find key
    Iterator q = inserter(p, Entry(k, v));                  // insert it here
    return q;                                               // return its position
  }
```

Code Fragment 9.26: An implementation of the dictionary function insert.

We exploit a property of how insert works. Whenever a new entry (k,v) is inserted, if the structure already contains another entry (k,v') with the same key, the finder utility function returns an iterator to the first such occurrence. The inserter utility then inserts the new entry just prior to this. It follows that all the entries having the same key are stored in a sequence of ***contiguous*** positions, all within the same bucket. (In fact, they appear in the reverse of their insertion order.) This means that, in order to produce an iterator range (b,e) for the call findAll(k), it suffices to set b to the first entry of this sequence and set e to the entry immediately following the last one.

Our implementation of findAll is given in Code Fragment 9.27. We first invoke the finder function to locate the key. If the finder returns a position at the end of some bucket, we know that the key is not present, and we return the empty iterator (end, end). Otherwise, recall from Code Fragment 9.15 that finder returns the first entry with the given key value. We store this in the entry iterator b. We then traverse

the bucket until either coming to the bucket's end or encountering an entry with a key of different value. Let p be this iterator value. We return the iterator range (b, p).

```
/* HashDict⟨K,V,H⟩ :: */                          // find all entries with k
  Range findAll(const K& k) {
    Iterator b = finder(k);                        // look up k
    Iterator p = b;
    while (!endOfBkt(p) && (*p).key() == (*b).key()) { // find next unequal key
      ++p;
    }
    return Range(b, p);                            // return range of positions
  }
```

Code Fragment 9.27: An implementation of the dictionary function findAll.

9.5.3 Implementations with Location-Aware Entries

As with the map ADT, there are several possible ways we can implement the dictionary ADT, including an unordered list, a hash table, an ordered search table, or a skip list. As we did for adaptable priority queues (Section 8.4.2), we can also use location-aware entries to speed up the running time for some operations in a dictionary. In removing a location-aware entry e, for instance, we could simply go directly to the place in our data structure where we are storing e and remove it. We could implement a location-aware entry, for example, by augmenting our entry class with a private *location* variable and protected functions location() and setLocation(p), which return and set this variable respectively. We would then require that the *location* variable for an entry e would always refer to e's position or index in the data structure. We would, of course, have to update this variable any time we moved an entry, as follows.

- **Unordered list**: In an unordered list, L, implementing a dictionary, we can maintain the *location* variable of each entry e to point to e's position in the underlying linked list for L. This choice allows us to perform erase(e) as L.erase(e.location()), which would run in $O(1)$ time.
- **Hash table with separate chaining**: Consider a hash table, with bucket array A and hash function h, that uses separate chaining for handling collisions. We use the *location* variable of each entry e to point to e's position in the list L implementing the list $A[h(k)]$. This choice allows us to perform an erase(e) as L.erase(e.location()), which would run in constant expected time.
- **Ordered search table**: In an ordered table, T, implementing a dictionary, we should maintain the *location* variable of each entry e to be e's index in T. This choice would allow us to perform erase(e) as T.erase(e.location()).

(Recall that location() now returns an integer.) This approach would run fast if entry *e* was stored near the end of *T*.

- *Skip list*: In a skip list, *S*, implementing a dictionary, we should maintain the *location* variable of each entry *e* to point to *e*'s position in the bottom level of *S*. This choice would allow us to skip the search step in our algorithm for performing erase(*e*) in a skip list.

We summarize the performance of entry removal in a dictionary with location-aware entries in Table 9.4.

List	Hash Table	Search Table	Skip List
$O(1)$	$O(1)$ (expected)	$O(n)$	$O(\log n)$ (expected)

Table 9.4: Performance of the erase function in dictionaries implemented with location-aware entries. We use *n* to denote the number of entries in the dictionary.

9.6 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-9.1 Which of the hash table collision-handling schemes could tolerate a load factor above 1 and which could not?

R-9.2 What is the worst-case running time for inserting n key-value entries into an initially empty map M that is implemented with a list?

R-9.3 What is the worst-case asymptotic running time for performing n (correct) erase() operations on a map, implemented with an ordered search table, that initially contains $2n$ entries?

R-9.4 Describe how to use a skip-list map to implement the dictionary ADT, allowing the user to insert different entries with equal keys.

R-9.5 Describe how an ordered list implemented as a doubly linked list could be used to implement the map ADT.

R-9.6 What would be a good hash code for a vehicle identification that is a string of numbers and letters of the form "9X9XX99X9XX999999," where a "9" represents a digit and an "X" represents a letter?

R-9.7 Draw the 11-entry hash table that results from using the hash function, $h(i) = (3i+5) \bmod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.

R-9.8 What is the result of the previous exercise, assuming collisions are handled by linear probing?

R-9.9 Show the result of Exercise R-9.7, assuming collisions are handled by quadratic probing, up to the point where the method fails.

R-9.10 What is the result of Exercise R-9.7 when collisions are handled by double hashing using the secondary hash function $h'(k) = 7 - (k \bmod 7)$?

R-9.11 Give a pseudo-code description of an insertion into a hash table that uses quadratic probing to resolve collisions, assuming we also use the trick of replacing deleted items with a special "deactivated item" object.

R-9.12 Describe a set of operations for an ordered dictionary ADT that would correspond to the functions of the ordered map ADT. Be sure to define the meaning of the functions so that they can deal with the possibility of different entries with equal keys.

R-9.13 Show the result of rehashing the hash table shown in Figure 9.4, into a table of size 19, using the new hash function $h(k) = 3k \bmod 17$.

R-9.14 Explain why a hash table is not suited to implement the ordered dictionary ADT.

R-9.15 What is the worst-case running time for inserting n items into an initially empty hash table, where collisions are resolved by chaining? What is the best case?

R-9.16 Draw an example skip list that results from performing the following series of operations on the skip list shown in Figure 9.12: erase(38), insert(48,x), insert(24,y), erase(55). Record your coin flips, as well.

R-9.17 Give a pseudo-code description of the erase operation in a skip list.

R-9.18 What is the expected running time of the functions for maintaining a maxima set if we insert n pairs such that each pair has lower cost and performance than the one before it? What is contained in the ordered dictionary at the end of this series of operations? What if each pair had a lower cost and higher performance than the one before it?

R-9.19 Argue why location-aware entries are not really needed for a dictionary implemented with a good hash table.

Creativity

C-9.1 Describe how you could perform each of the additional functions of the ordered map ADT using a skip list.

C-9.2 Describe how to use a skip list to implement the vector ADT, so that index-based insertions and removals both run in $O(\log n)$ expected time.

C-9.3 Suppose we are given two ordered dictionaries S and T, each with n items, and that S and T are implemented by means of array-based ordered sequences. Describe an $O(\log^2 n)$-time algorithm for finding the kth smallest key in the union of the keys from S and T (assuming no duplicates).

C-9.4 Give an $O(\log n)$-time solution for the previous problem.

C-9.5 Design a variation of binary search for performing findAll(k) in an ordered dictionary implemented with an ordered array, and show that it runs in time $O(\log n + s)$, where n is the number of elements in the dictionary and s is the size of the iterator returned.

C-9.6 The hash table dictionary implementation requires that we find a prime number between a number M and a number $2M$. Implement a function for finding such a prime by using the *sieve algorithm*. In this algorithm, we allocate a $2M$ cell Boolean array A, such that cell i is associated with the integer i. We then initialize the array cells to all be "true" and we "mark off" all the cells that are multiples of 2, 3, 5, 7, and so on. This process can stop after it reaches a number larger than $\sqrt{2M}$.
(Hint: Consider a bootstrapping method for computing the primes up to $\sqrt{2M}$.)

C-9.7 Describe how to perform a removal from a hash table that uses linear probing to resolve collisions where we do not use a special marker to represent deleted elements. That is, we must rearrange the contents so that it appears that the removed entry was never inserted in the first place.

C-9.8 Given a collection C of n cost-performance pairs (c, p), describe an algorithm for finding the maxima pairs of C in $O(n \log n)$ time.

C-9.9 The quadratic probing strategy has a clustering problem that relates to the way it looks for open slots when a collision occurs. Namely, when a collision occurs at bucket $h(k)$, we check $A[(h(k) + f(j)) \bmod N]$, for $f(j) = j^2$, using $j = 1, 2, \ldots, N - 1$.

  a. Show that $f(j) \bmod N$ will assume at most $(N + 1)/2$ distinct values, for N prime, as j ranges from 1 to $N - 1$. As a part of this justification, note that $f(R) = f(N - R)$ for all R.

  b. A better strategy is to choose a prime N, such that N is congruent to 3 modulo 4 and then to check the buckets $A[(h(k) \pm j^2) \bmod N]$ as j ranges from 1 to $(N - 1)/2$, alternating between addition and subtraction. Show that this alternate type of quadratic probing is guaranteed to check every bucket in A.

C-9.10 Show that the functions above(p) and before(p) are not actually needed to efficiently implement a dictionary using a skip list. That is, we can implement entry insertion and removal in a skip list using a strictly top-down, scan-forward approach, without ever using the above or before functions.

(Hint: In the insertion algorithm, first repeatedly flip the coin to determine the level where you should start inserting the new entry.)

C-9.11 Suppose that each row of an $n \times n$ array A consists of 1's and 0's such that, in any row of A, all the 1's come before any 0's in that row. Assuming A is already in memory, describe a method running in $O(n \log n)$ time (not $O(n^2)$ time!) for counting the number of 1's in A.

C-9.12 Describe an efficient ordered dictionary structure for storing n elements that have an associated set of $k < n$ keys that come from a total order. That is, the set of keys is smaller than the number of elements. Your structure should perform all the ordered dictionary operations in $O(\log k + s)$ expected time, where s is the number of elements returned.

C-9.13 Describe an efficient dictionary structure for storing n entries whose $r < n$ keys have distinct hash codes. Your structure should perform operation findAll in $O(1 + s)$ expected time, where s is the number of entries returned, and the remaining operations of the dictionary ADT in $O(1)$ expected time.

C-9.14 Describe an efficient data structure for implementing the *bag* ADT, which supports a function add(e), for adding an element e to the bag, and a function remove, which removes an arbitrary element in the bag. Show that both of these functions can be done in $O(1)$ time.

C-9.15 Describe how to modify the skip list data structure to support the function median(), which returns the position of the element in the "bottom" list S_0 at index $\lfloor n/2 \rfloor$. Show that your implementation of this function runs in $O(\log n)$ expected time.

Projects

P-9.1 Write a spell-checker class that stores a set of words, W, in a hash table and implements a function, spellCheck(s), which performs a *spell check* on the string s with respect to the set of words, W. If s is in W, then the call to spellCheck(s) returns an iterable collection that contains only s, since it is assumed to be spelled correctly in this case. Otherwise, if s is not in W, then the call to spellCheck(s) returns a list of every word in W that could be a correct spelling of s. Your program should be able to handle all the common ways that s might be a misspelling of a word in W, including swapping adjacent characters in a word, inserting a single character inbetween two adjacent characters in a word, deleting a single character from a word, and replacing a character in a word with another character. For an extra challenge, consider phonetic substitutions as well.

P-9.2 Write an implementation of the dictionary ADT using a linked list.

P-9.3 Write an implementation of the map ADT using a vector.

P-9.4 Implement a class that implements a version of an ordered dictionary ADT using a skip list. Be sure to carefully define and implement dictionary versions of corresponding functions of the ordered map ADT.

P-9.5 Implement the map ADT with a hash table with separate-chaining collision handling (do not adapt any of the STL classes).

P-9.6 Implement the ordered map ADT using a skip list.

P-9.7 Extend the previous project by providing a graphical animation of the skip list operations. Visualize how entries move up the skip list during insertions and are linked out of the skip list during removals. Also, in a search operation, visualize the scan-forward and drop-down actions.

P-9.8 Implement a dictionary that supports location-aware entries by means of an ordered list.

P-9.9 Perform a comparative analysis that studies the collision rates for various hash codes for character strings, such as various polynomial hash codes for different values of the parameter a. Use a hash table to determine

collisions, but only count collisions where different strings map to the same hash code (not if they map to the same location in this hash table). Test these hash codes on text files found on the Internet.

P-9.10 Perform a comparative analysis as in the previous exercise but for 10-digit telephone numbers instead of character strings.

P-9.11 Design a C++ class that implements the skip-list data structure. Use this class to create implementations of both the map and dictionary ADTs, including location-aware functions for the dictionary.

Chapter Notes

Hashing is a well studied technique. The reader interested in further study is encouraged to explore the book by Knuth [60], as well as the book by Vitter and Chen [100]. Interestingly, binary search was first published in 1946, but was not published in a fully correct form until 1962. For further discussions on lessons learned, please see papers by Bentley [11] and Levisse [64]. Skip lists were introduced by Pugh [86]. Our analysis of skip lists is a simplification of a presentation given by Motwani and Raghavan [80]. For a more in-depth analysis of skip lists, please see the various research papers on skip lists that have appeared in the data structures literature [54, 82, 83]. Exercise C-9.9 was contributed by James Lee.

This page intentionally left blank
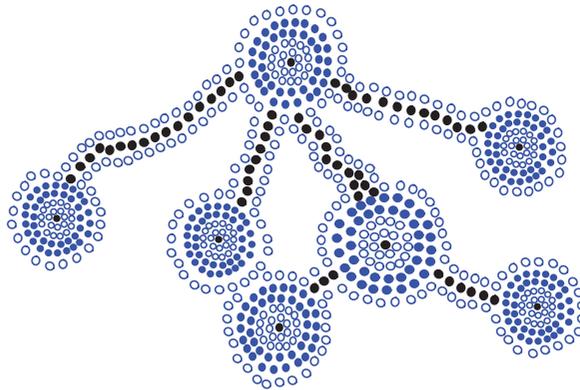
Contents

10.1   Binary Search Trees

All of the structures that we discuss in this chapter are ***search trees***, that is, tree data structures that can be used to implement ordered maps and ordered dictionaries. Recall from Chapter 9 that a map is a collection of key-value entries, with each value associated with a distinct key. A dictionary differs in that multiple values may share the same key value. Our presentation focuses mostly on maps, but we consider both data structures in this chapter.

We assume that maps and dictionaries provide a special pointer object, called an ***iterator***, which permits us to reference and enumerate the entries of the structure. In order to indicate that an object is not present, there exists a special sentinel iterator called end. By convention, this sentinel refers to an imaginary element that lies just beyond the last element of the structure.

Let M be a map. In addition to the standard container operations (size, empty, begin, and end) the map ADT (Section 9.1) includes the following:

$\text{find}(k)$: If M contains an entry $e = (k, v)$, with key equal to k, then return an iterator p referring to this entry, and otherwise return the special iterator end.

$\text{put}(k, v)$: If M does not have an entry with key equal to k, then add entry (k, v) to M, and otherwise, replace the value field of this entry with v; return an iterator to the inserted/modified entry.

$\text{erase}(k)$: Remove from M the entry with key equal to k; an error condition occurs if M has no such entry.

$\text{erase}(p)$: Remove from M the entry referenced by iterator p; an error condition occurs if p points to the end sentinel.

$\text{begin}()$: Return an iterator to the first entry of M.

$\text{end}()$: Return an iterator to a position just beyond the end of M.

The dictionary ADT (Section 9.5) provides the additional operations $\text{insert}(k, v)$, which inserts the entry (k, v), and $\text{findAll}(k)$, which returns an iterator range (b, e) of all entries whose key value is k.

Given an iterator p, the associated entry may be accessed using $*p$. The individual key and value can be accessed using $p\text{->key}()$ and $p\text{->value}()$, respectively. We assume that the key elements are drawn from a total order, which is defined by overloading the C++ relational less-than operator ("<"). Given an iterator p to some entry, it may be advanced to the next entry in this order using the increment operator ("++p").

The ordered map and dictionary ADTs also include some additional functions for finding predecessor and successor entries with respect to a given key, but their

performance is similar to that of find. So, we focus on find as the primary search operation in this chapter.

Binary Search Trees and Ordered Maps

Binary trees are an excellent data structure for storing the entries of a map, assuming we have an order relation defined on the keys. As mentioned previously (Section 7.3.6), a ***binary search tree*** is a binary tree T such that each internal node v of T stores an entry (k,x) such that:

- Keys stored at nodes in the left subtree of v are less than or equal to k
- Keys stored at nodes in the right subtree of v are greater than or equal to k.

An example of a search tree storing integer keys is shown in Figure 10.1.
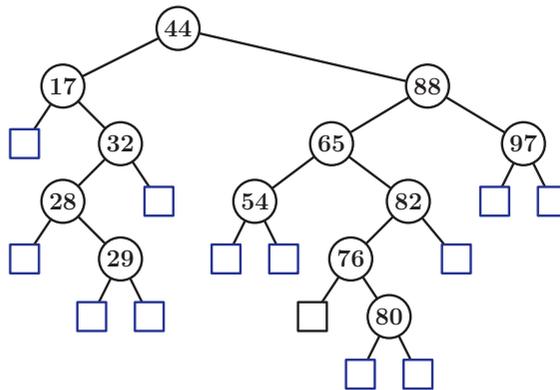
Figure 10.1: A binary search tree T representing a map with integer keys.

As we show below, the keys stored at the nodes of T provide a way of performing a search by making comparisons at a series of internal nodes. The search can stop at the current node v or continue at v's left or right child. Thus, we take the view here that binary search trees are nonempty proper binary trees. That is, we store entries only at the internal nodes of a binary search tree, and the external nodes serve as "placeholders." This approach simplifies several of our search and update algorithms. Incidentally, we could allow for improper binary search trees, which have better space usage, but at the expense of more complicated search and update functions.

Independent of whether we view binary search trees as proper or not, the important property of a binary search tree is the realization of an ordered map (or dictionary). That is, a binary search tree should hierarchically represent an ordering of its keys, using relationships between parent and children. Specifically, an inorder traversal (Section 7.3.6) of the nodes of a binary search tree T should visit the keys in nondecreasing order. Incrementing an iterator through a map visits the entries in this same order.

10.1.1   Searching

To perform operation find(k) in a map M that is represented with a binary search tree T, we view the tree T as a decision tree (recall Figure 7.10). In this case, the question asked at each internal node v of T is whether the search key k is less than, equal to, or greater than the key stored at node v, denoted with key(v). If the answer is "smaller," then the search continues in the left subtree. If the answer is "equal," then the search terminates successfully. If the answer is "greater," then the search continues in the right subtree. Finally, if we reach an external node, then the search terminates unsuccessfully. (See Figure 10.2.)
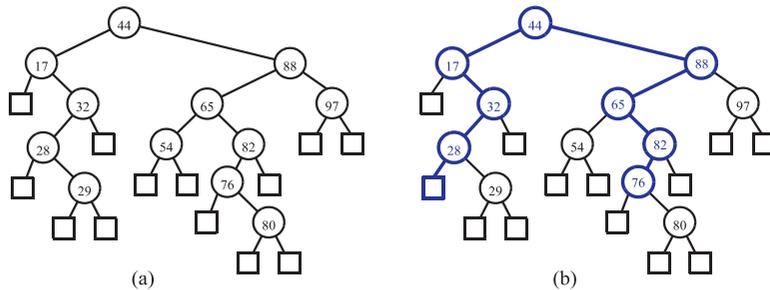
Figure 10.2: (a) A binary search tree T representing a map with integer keys; (b) nodes of T visited when executing operations find(76) (successful) and find(25) (unsuccessful) on M. For simplicity, we show only the keys of the entries.

We describe this approach in detail in Code Fragment 10.1. Given a search key k and a node v of T, this function, TreeSearch, returns a node (position) w of the subtree $T(v)$ of T rooted at v, such that one of the following occurs:

- w is an internal node and w's entry has key equal to k
- w is an external node representing k's proper place in an inorder traversal of $T(v)$, but k is not a key contained in $T(v)$

Thus, function find(k) can be performed by calling TreeSearch$(k, T.\text{root}())$. Let w be the node of T returned by this call. If w is an internal node, then we return w's entry; otherwise, we return *null*.

Algorithm TreeSearch(k, v):
    if $T.\text{isExternal}(v)$ **then**
       return v
    if $k < \text{key}(v)$ **then**
       return TreeSearch$(k, T.\text{left}(v))$
    else if $k > \text{key}(v)$ **then**
       return TreeSearch$(k, T.\text{right}(v))$
    return v \{we know $k = \text{key}(v)$\}

        Code Fragment 10.1: Recursive search in a binary search tree.

Analysis of Binary Tree Searching

The analysis of the worst-case running time of searching in a binary search tree T is simple. Algorithm TreeSearch is recursive and executes a constant number of primitive operations for each recursive call. Each recursive call of TreeSearch is made on a child of the previous node. That is, TreeSearch is called on the nodes of a path of T that starts at the root and goes down one level at a time. Thus, the number of such nodes is bounded by $h+1$, where h is the height of T. In other words, since we spend $O(1)$ time per node encountered in the search, function find on map M runs in $O(h)$ time, where h is the height of the binary search tree T used to implement M. (See Figure 10.3.)
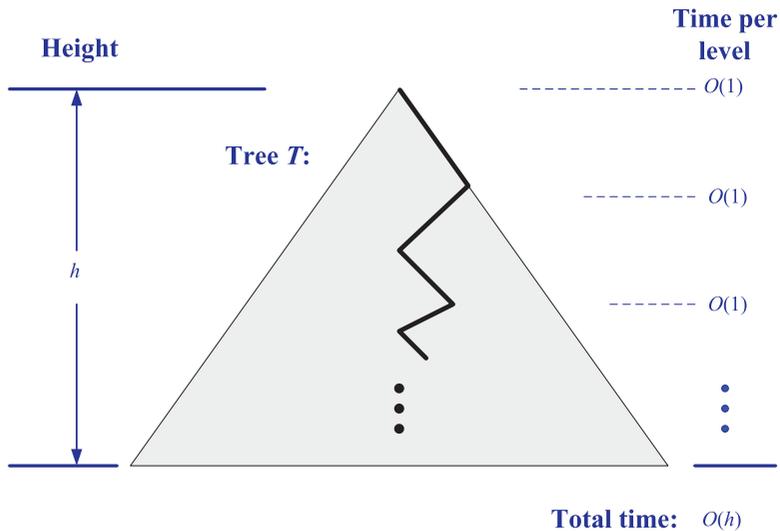
Figure 10.3: The running time of searching in a binary search tree. We use a standard visualization shortcut of viewing a binary search tree as a big triangle and a path from the root as a zig-zag line.

We can also show that a variation of the above algorithm performs operation findAll(k) of the dictionary ADT in time $O(h+s)$, where s is the number of entries returned. However, this function is slightly more complicated, and the details are left as an exercise (Exercise C-10.2).

Admittedly, the height h of T can be as large as the number of entries, n, but we expect that it is usually much smaller. Indeed, we show how to maintain an upper bound of $O(\log n)$ on the height of a search tree T in Section 10.2. Before we describe such a scheme, however, let us describe implementations for map update functions.

10.1.2   Update Operations

Binary search trees allow implementations of the insert and erase operations using algorithms that are fairly straightforward, but not trivial.

Insertion

Let us assume a proper binary tree T supports the following update operation:

insertAtExternal(v, e): Insert the element e at the external node v, and expand v to be internal, having new (empty) external node children; an error occurs if v is an internal node.

Given this function, we perform insert(k, x) for a dictionary implemented with a binary search tree T by calling TreeInsert$(k, x, T.\text{root}())$, which is given in Code Fragment 10.2.

Algorithm TreeInsert(k, x, v):
  Input: A search key k, an associated value, x, and a node v of T
  Output: A new node w in the subtree $T(v)$ that stores the entry (k, x)
  $w \leftarrow$ TreeSearch(k, v)
  if $T.\text{isInternal}(w)$ **then**
    return TreeInsert$(k, x, T.\text{left}(w))$ {going to the right would be correct too}
  $T.\text{insertAtExternal}(w, (k, x))$ {this is an appropriate place to put (k, x)}
  return w

Code Fragment 10.2: Recursive algorithm for insertion in a binary search tree.

This algorithm traces a path from T's root to an external node, which is expanded into a new internal node accommodating the new entry. An example of insertion into a binary search tree is shown in Figure 10.4.

(a)   (b)
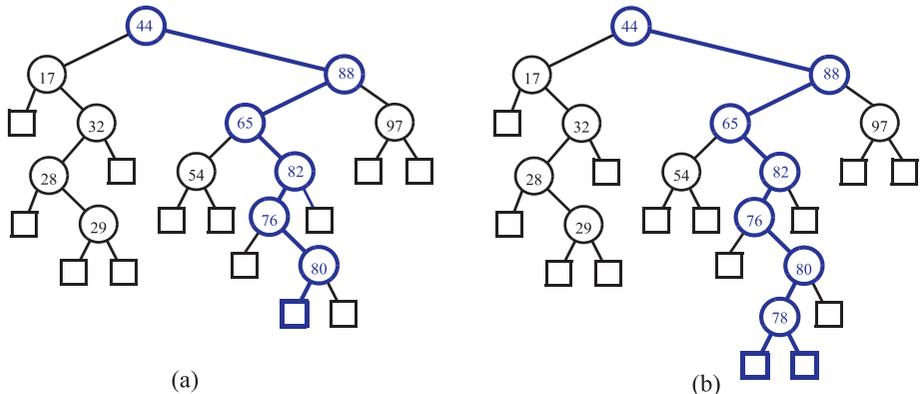
Figure 10.4: Insertion of an entry with key 78 into the search tree of Figure 10.1: (a) finding the position to insert; (b) the resulting tree.

Removal

The implementation of the erase(k) operation on a map M implemented with a binary search tree T is a bit more complex, since we do not wish to create any "holes" in the tree T. We assume, in this case, that a proper binary tree supports the following additional update operation:

removeAboveExternal(v): Remove an external node v and its parent, replacing v's parent with v's sibling; an error occurs if v is not external.

Given this operation, we begin our implementation of operation erase(k) of the map ADT by calling TreeSearch(k, T.root()) on T to find a node of T storing an entry with key equal to k. If TreeSearch returns an external node, then there is no entry with key k in map M, and an error condition is signaled. If, instead, TreeSearch returns an internal node w, then w stores an entry we wish to remove, and we distinguish two cases:

- If one of the children of node w is an external node, say node z, we simply remove w and z from T by means of operation removeAboveExternal(z) on T. This operation restructures T by replacing w with the sibling of z, removing both w and z from T. (See Figure 10.5.)
- If both children of node w are internal nodes, we cannot simply remove the node w from T, since this would create a "hole" in T. Instead, we proceed as follows (see Figure 10.6):

  - We find the first internal node y that follows w in an inorder traversal of T. Node y is the left-most internal node in the right subtree of w, and is found by going first to the right child of w and then down T from there, following the left children. Also, the left child x of y is the external node that immediately follows node w in the inorder traversal of T.
  - We move the entry of y into w. This action has the effect of removing the former entry stored at w.
  - We remove nodes x and y from T by calling removeAboveExternal(x) on T. This action replaces y with x's sibling, and removes both x and y from T.

As with searching and insertion, this removal algorithm traverses a path from the root to an external node, possibly moving an entry between two nodes of this path, and then performs a removeAboveExternal operation at that external node.

The position-based variant of removal is the same, except that we can skip the initial step of invoking TreeSearch(k, T.root()) to locate the node containing the key.
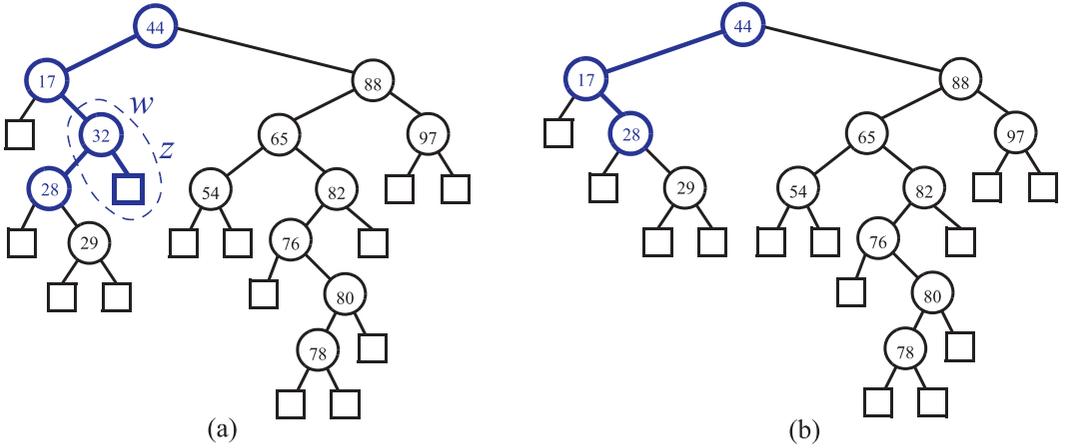
Figure 10.5: Removal from the binary search tree of Figure 10.4b, where the entry to remove (with key 32) is stored at a node (*w*) with an external child: (a) before the removal; (b) after the removal.
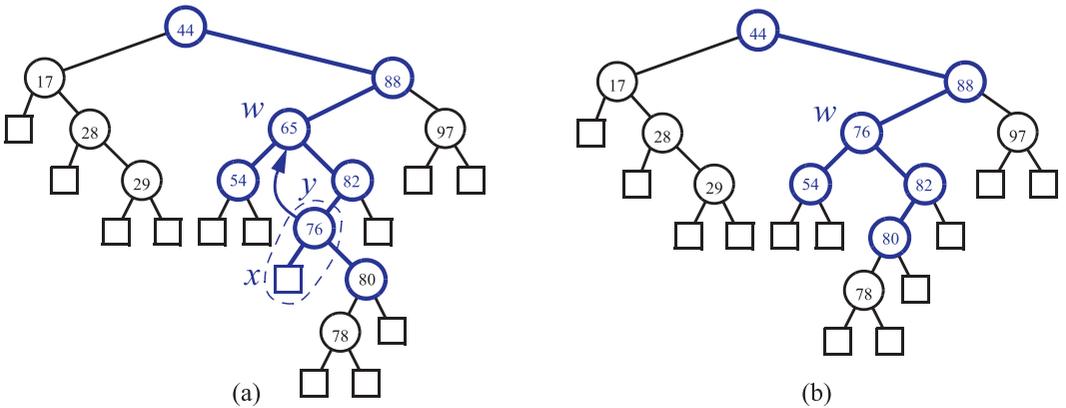
Figure 10.6: Removal from the binary search tree of Figure 10.4b, where the entry to remove (with key 65) is stored at a node (*w*) whose children are both internal: (a) before the removal; (b) after the removal.

Performance of a Binary Search Tree

The analysis of the search, insertion, and removal algorithms are similar. We spend $O(1)$ time at each node visited, and, in the worst case, the number of nodes visited is proportional to the height h of T. Thus, in a map M implemented with a binary search tree T, the find, insert, and erase functions run in $O(h)$ time, where h is the height of T. Thus, a binary search tree T is an efficient implementation of a map with n entries only if the height of T is small. In the best case, T has height $h = \lceil \log(n+1) \rceil$, which yields logarithmic-time performance for all the map operations. In the worst case, however, T has height n, in which case it would look and feel like an ordered list implementation of a map. Such a worst-case configuration arises, for example, if we insert a series of entries with keys in increasing or decreasing order. (See Figure 10.7.)
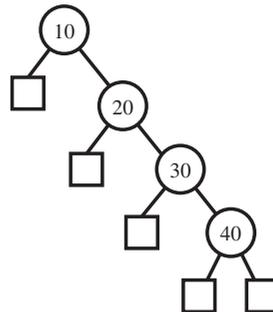
Figure 10.7: Example of a binary search tree with linear height, obtained by inserting entries with keys in increasing order.

The performance of a map implemented with a binary search tree is summarized in the following proposition and in Table 10.1.

Proposition 10.1: *A binary search tree T with height h for n key-value entries uses $O(n)$ space and executes the map ADT operations with the following running times. Operations* size *and* empty *each take $O(1)$ time. Operations* find, insert, *and* erase *each take $O(h)$ time.*

Operation	Time
size, empty	$O(1)$
find, insert, erase	$O(h)$

Table 10.1: Running times of the main functions of a map realized by a binary search tree. We denote the current height of the tree with h. The space usage is $O(n)$, where n is the number of entries stored in the map.

Note that the running time of search and update operations in a binary search tree varies dramatically depending on the tree's height. We can nevertheless take

comfort that, on average, a binary search tree with n keys generated from a random series of insertions and removals of keys has expected height $O(\log n)$. Such a statement requires careful mathematical language to precisely define what we mean by a random series of insertions and removals, and sophisticated probability theory to prove; hence, its justification is beyond the scope of this book. Nevertheless, keep in mind the poor worst-case performance and take care in using standard binary search trees in applications where updates are not random. There are, after all, applications where it is essential to have a map with fast worst-case search and update times. The data structures presented in the next sections address this need.

10.1.3   C++ Implementation of a Binary Search Tree

In this section, we present a C++ implementation of the dictionary ADT based on a binary search tree, which we call Search Tree. Recall that a dictionary differs from a map in that it allows multiple copies of the same key to be inserted. For simplicity, we have not implemented the findAll function.

To keep the number of template parameters small, rather than templating our class on the key and value types, we have chosen instead to template our binary search tree on just the entry type denoted E. To obtain access to the key and value types, we assume that the entry class defines two public types defining them. Given an entry object of type E, we may access these types E::Key and E::Value. Otherwise, our entry class is essentially the same as the entry class given in Code Fragment 9.1. It is presented in Code Fragment 10.3.

```cpp
template <typename K, typename V>
class Entry {                                       // a (key, value) pair
public:                                             // public types
    typedef K Key;                                  // key type
    typedef V Value;                                // value type
public:                                             // public functions
    Entry(const K& k = K(), const V& v = V())       // constructor
        : _key(k), _value(v) { }
    const K& key() const { return _key; }           // get key (read only)
    const V& value() const { return _value; }       // get value (read only)
    void setKey(const K& k) { _key = k; }           // set key
    void setValue(const V& v) { _value = v; }       // set value
private:                                            // private data
    K _key;                                         // key
    V _value;                                       // value
};
```

Code Fragment 10.3: A C++ class for a key-value entry.

In Code Fragment 10.4, we present the main parts of the class definition for our binary search tree. We begin by defining the publicly accessible types for the entry, key, value, and the class iterator. This is followed by a declaration of the public member functions. We define two local types, BinaryTree and TPos, which represent a binary search tree and position within this binary tree, respectively. We also declare a number of local utility functions to help in finding, inserting, and erasing entries. The member data consists of a binary tree and the number of entries in the tree.

```
template <typename E>
class SearchTree {                              // a binary search tree
public:                                         // public types
  typedef typename E::Key K;                    // a key
  typedef typename E::Value V;                  // a value
  class Iterator;                               // an iterator/position
public:                                         // public functions
  SearchTree();                                 // constructor
  int size() const;                             // number of entries
  bool empty() const;                           // is the tree empty?
  Iterator find(const K& k);                    // find entry with key k
  Iterator insert(const K& k, const V& x);      // insert (k,x)
  void erase(const K& k) throw(NonexistentElement); // remove key k entry
  void erase(const Iterator& p);                // remove entry at p
  Iterator begin();                             // iterator to first entry
  Iterator end();                               // iterator to end entry
protected:                                      // local utilities
  typedef BinaryTree<E> BinaryTree;             // linked binary tree
  typedef typename BinaryTree::Position TPos;   // position in the tree
  TPos root() const;                            // get virtual root
  TPos finder(const K& k, const TPos& v);       // find utility
  TPos inserter(const K& k, const V& x);        // insert utility
  TPos eraser(TPos& v);                         // erase utility
  TPos restructure(const TPos& v)               // restructure
      throw(BoundaryViolation);
private:                                        // member data
  BinaryTree T;                                 // the binary tree
  int n;                                        // number of entries
public:
  // ...insert Iterator class declaration here
};
```

Code Fragment 10.4: Class SearchTree, which implements a binary search tree.

We have omitted the definition of the iterator class for our binary search tree. This is presented in Code Fragment 10.5. An iterator consists of a single position in the tree. We overload the dereferencing operator ("∗") to provide both read-

only and read-write access to the node referenced by the iterator. We also provide an operator for checking the equality of two iterators. This is useful for checking whether an iterator is equal to end.

```
class Iterator {                                        // an iterator/position
private:
  TPos v;                                               // which entry
public:
  Iterator(const TPos& vv) : v(vv) { }                  // constructor
  const E& operator*() const { return *v; } // get entry (read only)
  E& operator*() { return *v; }                         // get entry (read/write)
  bool operator==(const Iterator& p) const              // are iterators equal?
    { return v == p.v; }
  Iterator& operator++();                               // inorder successor
  friend class SearchTree;                              // give search tree access
};
```

Code Fragment 10.5: Declaration of the Iterator class, which is part of SearchTree.

Code Fragment 10.6 presents the definition of the iterator's increment operator, which advances the iterator from a given position of the tree to its inorder successor. Only internal nodes are visited, since external nodes do not contain entries. If the node *v* has a right child, the inorder successor is the leftmost internal node of its right subtree. Otherwise, *v* must be the largest key in the left subtree of some node *w*. To find *w*, we walk up the tree through successive ancestors. As long as we are the right child of the current ancestor, we continue to move upwards. When this is no longer true, the parent is the desired node *w*. Note that we employ the condensed function notation, which we introduced in Section 9.2.7, where the messy scoping qualifiers involving SearchTree have been omitted.

```
/* SearchTree⟨E⟩ :: */                                 // inorder successor
  Iterator& Iterator::operator++() {
    TPos w = v.right();
    if (w.isInternal()) {                               // have right subtree?
      do { v = w; w = w.left(); }                       // move down left chain
      while (w.isInternal());
    }
    else {
      w = v.parent();                                   // get parent
      while (v == w.right())                            // move up right chain
        { v = w; w = w.parent(); }
      v = w;                                            // and first link to left
    }
    return *this;
  }
```

Code Fragment 10.6: The increment operator ("++") for Iterator.

The implementation of the increment operator appears to contain an obvious bug. If the iterator points to the rightmost node of the entire tree, then the above function would loop until arriving at the root, which has no parent. The rightmost node of the tree has no successor, so the iterator *should* return the value end.

There is a simple and elegant way to achieve the desired behavior. We add a special sentinel node to our tree, called the ***super root***, which is created when the initial tree is constructed. The root of the binary search tree, which we call the ***virtual root***, is made the left child of the super root. We define end to be an iterator that returns the position of the super root. Observe that, if we attempt to increment an iterator that points to the rightmost node of the tree, the function given in Code Fragment 10.6 moves up the right chain until reaching the virtual root, and then stops at its parent, the super root, since the virtual root is its left child. Therefore, it returns an iterator pointing to the super root, which is equivalent to end. This is exactly the behavior we desire.

To implement this strategy, we define the constructor to create the super root. We also define a function root, which returns the virtual root's position, that is, the left child of the super root. These functions are given in Code Fragment 10.7.

```
/* SearchTree⟨E⟩ :: */                              // constructor
  SearchTree() : T(), n(0)
    { T.addRoot(); T.expandExternal(T.root()); }    // create the super root

/* SearchTree⟨E⟩ :: */                              // get virtual root
  TPos root() const
    { return T.root().left(); }                     // left child of super root
```

Code Fragment 10.7: The constructor and the utility function root. The constructor creates the super root, and root returns the virtual root of the binary search tree.

Next, in Code Fragment 10.8, we define the functions begin and end. The function begin returns the first node according to an inorder traversal, which is the leftmost internal node. The function end returns the position of the super root.

```
/* SearchTree⟨E⟩ :: */                              // iterator to first entry
  Iterator begin() {
    TPos v = root();                                // start at virtual root
    while (v.isInternal()) v = v.left();            // find leftmost node
    return Iterator(v.parent());
  }

/* SearchTree⟨E⟩ :: */                              // iterator to end entry
  Iterator end()
    { return Iterator(T.root()); }                  // return the super root
```

Code Fragment 10.8: The begin and end functions of class SearchTree. The function end returns a pointer to the super root.

We are now ready to present implementations of the principal class functions, for finding, inserting, and removing entries. We begin by presenting the function find(k) in Code Fragment 10.9. It invokes the recursive utility function finder starting at the root. This utility function is based on the algorithm given in Code Fragment 10.1. The code has been structured so that only the less-than operator needs to be defined on keys.

```
/* SearchTree⟨E⟩ :: */                            // find utility
  TPos finder(const K& k, const TPos& v) {
    if (v.isExternal()) return v;                  // key not found
    if (k < v−>key()) return finder(k, v.left());  // search left subtree
    else if (v−>key() < k) return finder(k, v.right()); // search right subtree
    else return v;                                 // found it here
  }

/* SearchTree⟨E⟩ :: */                            // find entry with key k
  Iterator find(const K& k) {
    TPos v = finder(k, root());                    // search from virtual root
    if (v.isInternal()) return Iterator(v);        // found it
    else return end();                             // didn't find it
  }
```

Code Fragment 10.9: The functions of SearchTree related to finding keys.

The insertion functions are shown in Code Fragment 10.10. The inserter utility does all the work. First, it searches for the key. If found, we continue to search until reaching an external node. (Recall that we allow duplicate keys.) We then create a node, copy the entry information into this node, and update the entry count. The insert function simply invokes the inserter utility, and converts the resulting node position into an iterator.

```
/* SearchTree⟨E⟩ :: */                            // insert utility
  TPos inserter(const K& k, const V& x) {
    TPos v = finder(k, root());                    // search from virtual root
    while (v.isInternal())                         // key already exists?
      v = finder(k, v.right());                    // look further
    T.expandExternal(v);                           // add new internal node
    v−>setKey(k); v−>setValue(x);                  // set entry
    n++;                                           // one more entry
    return v;                                      // return insert position
  }

/* SearchTree⟨E⟩ :: */                            // insert (k,x)
  Iterator insert(const K& k, const V& x)
    { TPos v = inserter(k, x); return Iterator(v); }
```

Code Fragment 10.10: The functions of SearchTree for inserting entries.

Finally, we present the removal functions in Code Fragment 10.11. We implement the approach presented in Section 10.1.2. If the node has an external child, we set w to point to this child. Otherwise, we let w be the leftmost external node in v's right subtree. Let u be w's parent. We copy u's entry contents to v. In all cases, we then remove the external node w and its parent through the use of the binary tree functions removeAboveExternal.

```
/* SearchTree⟨E⟩ :: */                              // remove utility
  TPos eraser(TPos& v) {
    TPos w;
    if (v.left().isExternal()) w = v.left();        // remove from left
    else if (v.right().isExternal()) w = v.right(); // remove from right
    else {                                          // both internal?
      w = v.right();                                // go to right subtree
      do { w = w.left(); } while (w.isInternal());  // get leftmost node
      TPos u = w.parent();
      v−>setKey(u−>key()); v−>setValue(u−>value()); // copy w's parent to v
    }
    n−−;                                            // one less entry
    return T.removeAboveExternal(w);                // remove w and parent
  }

/* SearchTree⟨E⟩ :: */                              // remove key k entry
  void erase(const K& k) throw(NonexistentElement) {
    TPos v = finder(k, root());                     // search from virtual root
    if (v.isExternal())                             // not found?
      throw NonexistentElement("Erase of nonexistent");
    eraser(v);                                      // remove it
  }

/* SearchTree⟨E⟩ :: */                              // erase entry at p
  void erase(const Iterator& p)
    { eraser(p.v); }
```

Code Fragment 10.11: The functions of SearchTree involved with removing entries.

When updating node entries (in inserter and eraser), we explicitly change only the key and value (using setKey and setValue). You might wonder, what else is there to change? Later in this chapter, we present data structures that are based on modifying the Entry class. It is important that only the key and value data are altered when copying nodes for these structures.

Our implementation has focused on the main elements of the binary search tree implementation. There are a few more things that could have been included. It is a straightforward exercise to implement the dictionary operation findAll. It would also be worthwhile to implement the decrement operator ("−−"), which moves an iterator to its inorder predecessor.

10.2   AVL Trees

In the previous section, we discussed what should be an efficient map data struc-
ture, but the worst-case performance it achieves for the various operations is linear
time, which is no better than the performance of list- and array-based map imple-
mentations (such as the unordered lists and search tables discussed in Chapter 9).
In this section, we describe a simple way of correcting this problem in order to
achieve logarithmic time for all the fundamental map operations.

Definition of an AVL Tree

The simple correction is to add a rule to the binary search tree definition that main-
tains a logarithmic height for the tree. The rule we consider in this section is the
following **height-balance property**, which characterizes the structure of a binary
search tree T in terms of the heights of its internal nodes (recall from Section 7.2.1
that the height of a node v in a tree is the length of the longest path from v to an
external node):

Height-Balance Property: For every internal node v of T, the heights of the chil-
dren of v differ by at most 1.

Any binary search tree T that satisfies the height-balance property is said to be an
AVL tree, named after the initials of its inventors, Adel'son-Vel'skii and Landis.
An example of an AVL tree is shown in Figure 10.8.
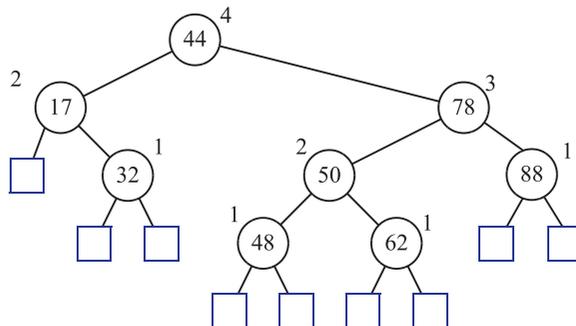
Figure 10.8: An example of an AVL tree. The keys of the entries are shown inside
the nodes, and the heights of the nodes are shown next to the nodes.

An immediate consequence of the height-balance property is that a subtree of an
AVL tree is itself an AVL tree. The height-balance property has also the important
consequence of keeping the height small, as shown in the following proposition.

Proposition 10.2: *The height of an AVL tree storing n entries is $O(\log n)$.*

Justification: Instead of trying to find an upper bound on the height of an AVL tree directly, it turns out to be easier to work on the "inverse problem" of finding a lower bound on the minimum number of internal nodes $n(h)$ of an AVL tree with height h. We show that $n(h)$ grows at least exponentially. From this, it is an easy step to derive that the height of an AVL tree storing n entries is $O(\log n)$.

To start with, notice that $n(1) = 1$ and $n(2) = 2$, because an AVL tree of height 1 must have at least one internal node and an AVL tree of height 2 must have at least two internal nodes. Now, for $h \geq 3$, an AVL tree with height h and the minimum number of nodes is such that both its subtrees are AVL trees with the minimum number of nodes: one with height $h - 1$ and the other with height $h - 2$. Taking the root into account, we obtain the following formula that relates $n(h)$ to $n(h - 1)$ and $n(h - 2)$, for $h \geq 3$:

$$n(h) = 1 + n(h-1) + n(h-2). \tag{10.1}$$

At this point, the reader familiar with the properties of Fibonacci progressions (Section 2.2.3 and Exercise C-4.17) already sees that $n(h)$ is a function exponential in h. For the rest of the readers, we will proceed with our reasoning.

Formula 10.1 implies that $n(h)$ is a strictly increasing function of h. Thus, we know that $n(h-1) > n(h-2)$. Replacing $n(h-1)$ with $n(h-2)$ in Formula 10.1 and dropping the 1, we get, for $h \geq 3$,

$$n(h) > 2 \cdot n(h-2). \tag{10.2}$$

Formula 10.2 indicates that $n(h)$ at least doubles each time h increases by 2, which intuitively means that $n(h)$ grows exponentially. To show this fact in a formal way, we apply Formula 10.2 repeatedly, yielding the following series of inequalities:

$$
\begin{aligned}
n(h) &> 2 \cdot n(h-2) \\
&> 4 \cdot n(h-4) \\
&> 8 \cdot n(h-6) \\
&\;\;\vdots \\
&> 2^i \cdot n(h-2i). \tag{10.3}
\end{aligned}
$$

That is, $n(h) > 2^i \cdot n(h-2i)$, for any integer i, such that $h - 2i \geq 1$. Since we already know the values of $n(1)$ and $n(2)$, we pick i so that $h - 2i$ is equal to either 1 or 2. That is, we pick

$$i = \left\lceil \frac{h}{2} \right\rceil - 1.$$

By substituting the above value of i in formula 10.3, we obtain, for $h \geq 3$,

$$
\begin{aligned}
n(h) \; &> \; 2^{\lceil \frac{h}{2} \rceil - 1} \cdot n \left(h - 2 \left\lceil \frac{h}{2} \right\rceil + 2 \right) \\
&\geq \; 2^{\lceil \frac{h}{2} \rceil - 1} n(1) \\
&\geq \; 2^{\frac{h}{2} - 1}.
\end{aligned} \tag{10.4}
$$

By taking logarithms of both sides of formula 10.4, we obtain

$$
\log n(h) \; > \; \frac{h}{2} - 1,
$$

from which we get

$$
h \; < \; 2 \log n(h) + 2, \tag{10.5}
$$

which implies that an AVL tree storing n entries has height at most $2 \log n + 2$. ∎

By Proposition 10.2 and the analysis of binary search trees given in Section 10.1, the operation find, in a map implemented with an AVL tree, runs in time $O(\log n)$, where n is the number of entries in the map. Of course, we still have to show how to maintain the height-balance property after an insertion or removal.

10.2.1   Update Operations

The insertion and removal operations for AVL trees are similar to those for binary search trees, but with AVL trees we must perform additional computations.

Insertion

An insertion in an AVL tree T begins as in an insert operation described in Section 10.1.2 for a (simple) binary search tree. Recall that this operation always inserts the new entry at a node w in T that was previously an external node, and it makes w become an internal node with operation insertAtExternal. That is, it adds two external node children to w. This action may violate the height-balance property, however, for some nodes increase their heights by one. In particular, node w, and possibly some of its ancestors, increase their heights by one. Therefore, let us describe how to restructure T to restore its height balance.

Given a binary search tree T, we say that an internal node v of T is *balanced* if the absolute value of the difference between the heights of the children of v is at most 1, and we say that it is *unbalanced* otherwise. Thus, the height-balance property characterizing AVL trees is equivalent to saying that every internal node is balanced.

Suppose that T satisfies the height-balance property, and hence is an AVL tree, prior to our inserting the new entry. As we have mentioned, after performing the

operation insertAtExternal on T, the heights of some nodes of T, including w, increase. All such nodes are on the path of T from w to the root of T, and these are the only nodes of T that may have just become unbalanced. (See Figure 10.9(a).) Of course, if this happens, then T is no longer an AVL tree; hence, we need a mechanism to fix the "unbalance" that we have just caused.

(a)   (b)
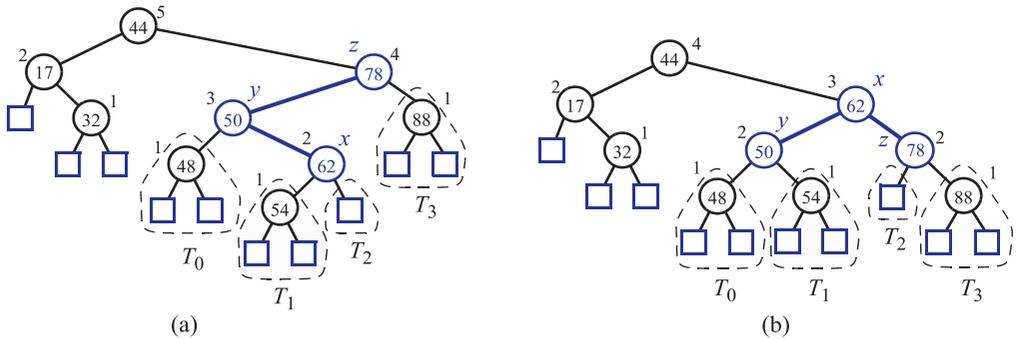
Figure 10.9: An example insertion of an entry with key 54 in the AVL tree of Figure 10.8: (a) after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced; (b) a trinode restructuring restores the height-balance property. We show the heights of nodes next to them, and we identify the nodes x, y, and z participating in the trinode restructuring.

We restore the balance of the nodes in the binary search tree T by a simple "search-and-repair" strategy. In particular, let z be the first node we encounter in going up from w toward the root of T such that z is unbalanced. (See Figure 10.9(a).) Also, let y denote the child of z with higher height (and note that node y must be an ancestor of w). Finally, let x be the child of y with higher height (there cannot be a tie and node x must be an ancestor of w). Also, node x is a grandchild of z and could be equal to w. Since z became unbalanced because of an insertion in the subtree rooted at its child y, the height of y is 2 greater than its sibling.

We now rebalance the subtree rooted at z by calling the ***trinode restructuring*** function, restructure(x), given in Code Fragment 10.12 and illustrated in Figures 10.9 and 10.10. A trinode restructuring temporarily renames the nodes x, y, and z as a, b, and c, so that a precedes b and b precedes c in an inorder traversal of T. There are four possible ways of mapping x, y, and z to a, b, and c, as shown in Figure 10.10, which are unified into one case by our relabeling. The trinode restructuring then replaces z with the node called b, makes the children of this node be a and c, and makes the children of a and c be the four previous children of x, y, and z (other than x and y) while maintaining the inorder relationships of all the nodes in T.

Algorithm restructure(x):

    Input: A node x of a binary search tree T that has both a parent y and a grand-parent z

    Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving nodes x, y, and z

  1: Let (a, b, c) be a left-to-right (inorder) listing of the nodes x, y, and z, and let (T_0, T_1, T_2, T_3) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.

  2: Replace the subtree rooted at z with a new subtree rooted at b.

  3: Let a be the left child of b and let T_0 and T_1 be the left and right subtrees of a, respectively.

  4: Let c be the right child of b and let T_2 and T_3 be the left and right subtrees of c, respectively.

Code Fragment 10.12: The trinode restructuring operation in a binary search tree.

The modification of a tree T caused by a trinode restructuring operation is often called a ***rotation***, because of the geometric way we can visualize the way it changes T. If $b = y$, the trinode restructuring method is called a ***single rotation***, for it can be visualized as "rotating" y over z. (See Figure 10.10(a) and (b).) Otherwise, if $b = x$, the trinode restructuring operation is called a ***double rotation***, for it can be visualized as first "rotating" x over y and then over z. (See Figure 10.10(c) and (d), and Figure 10.9.) Some computer researchers treat these two kinds of rotations as separate methods, each with two symmetric types. We have chosen, however, to unify these four types of rotations into a single trinode restructuring operation. No matter how we view it, though, the trinode restructuring method modifies parent-child relationships of $O(1)$ nodes in T, while preserving the inorder traversal ordering of all the nodes in T.

In addition to its order-preserving property, a trinode restructuring changes the heights of several nodes in T, so as to restore balance. Recall that we execute the function restructure(x) because z, the grandparent of x, is unbalanced. Moreover, this unbalance is due to one of the children of x now having too large a height relative to the height of z's other child. As a result of a rotation, we move up the "tall" child of x while pushing down the "short" child of z. Thus, after performing restructure(x), all the nodes in the subtree now rooted at the node we called b are balanced. (See Figure 10.10.) Thus, we restore the height-balance property ***locally*** at the nodes x, y, and z. In addition, since after performing the new entry insertion the subtree rooted at b replaces the one formerly rooted at z, which was taller by one unit, all the ancestors of z that were formerly unbalanced become balanced. (See Figure 10.9.) (The justification of this fact is left as Exercise C-10.14.) Therefore, this one restructuring also restores the height-balance property ***globally***.
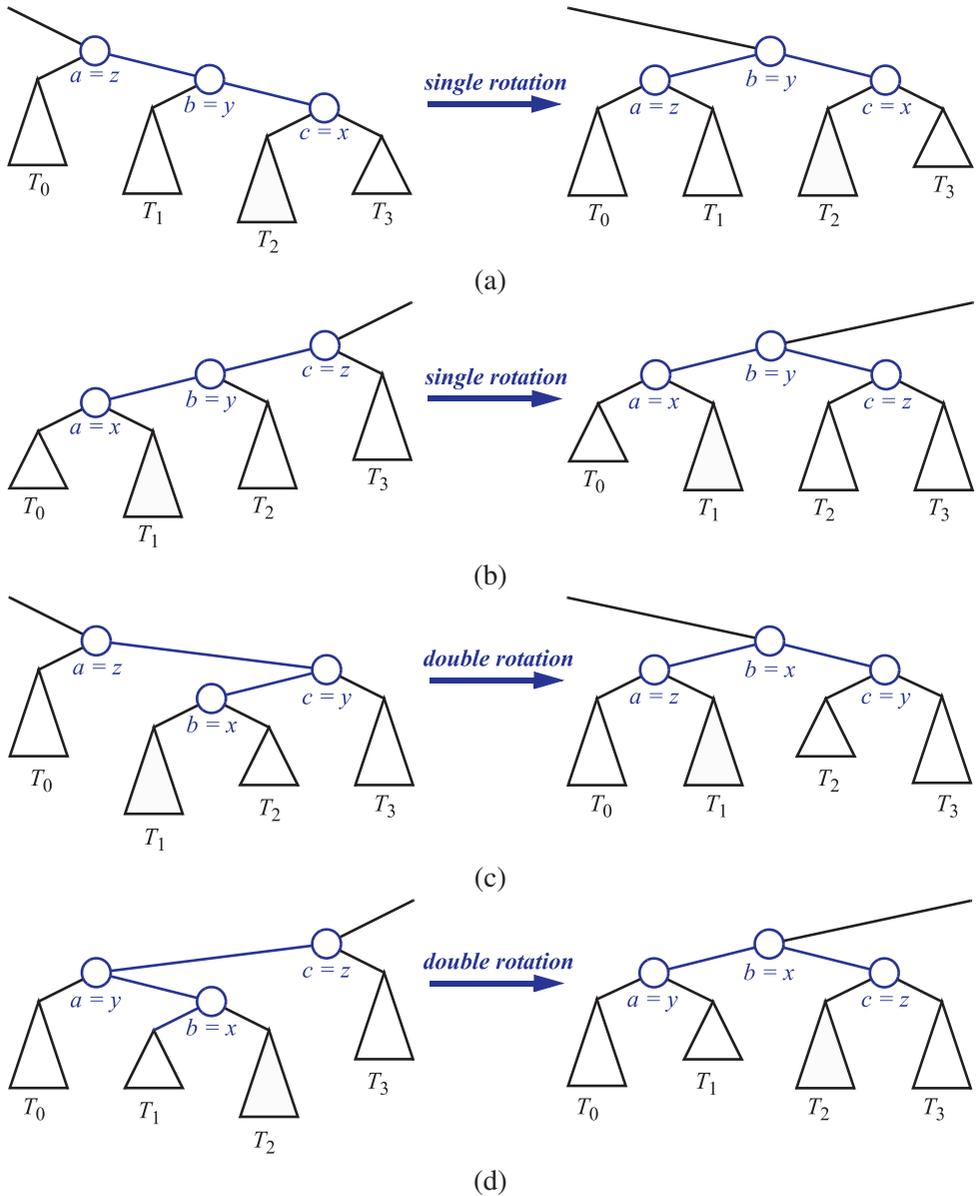
Figure 10.10: Schematic illustration of a trinode restructuring operation (Code Fragment 10.12): (a) and (b) a single rotation; (c) and (d) a double rotation.

Removal

As was the case for the insert map operation, we begin the implementation of the erase map operation on an AVL tree T by using the algorithm for performing this operation on a regular binary search tree. The added difficulty in using this approach with an AVL tree is that it may violate the height-balance property. In particular, after removing an internal node with operation removeAboveExternal and elevating one of its children into its place, there may be an unbalanced node in T on the path from the parent w of the previously removed node to the root of T. (See Figure 10.11(a).) In fact, there can be one such unbalanced node at most. (The justification of this fact is left as Exercise C-10.13.)
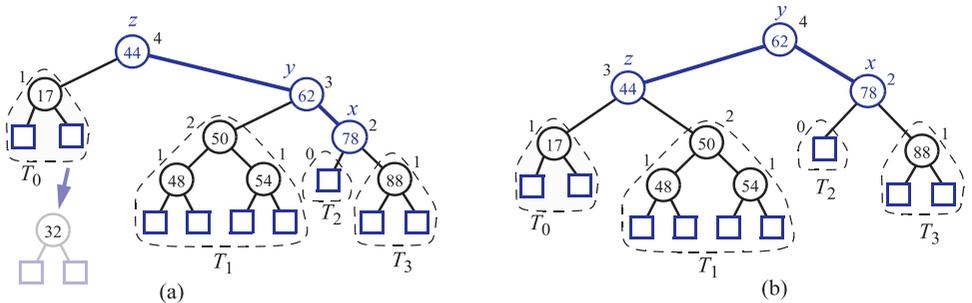
Figure 10.11: Removal of the entry with key 32 from the AVL tree of Figure 10.8: (a) after removing the node storing key 32, the root becomes unbalanced; (b) a (single) rotation restores the height-balance property.

As with insertion, we use trinode restructuring to restore balance in the tree T. In particular, let z be the first unbalanced node encountered going up from w toward the root of T. Also, let y be the child of z with larger height (note that node y is the child of z that is not an ancestor of w), and let x be the child of y defined as follows: if one of the children of y is taller than the other, let x be the taller child of y; else (both children of y have the same height), let x be the child of y on the same side as y (that is, if y is a left child, let x be the left child of y, else let x be the right child of y). In any case, we then perform a restructure(x) operation, which restores the height-balance property *locally*, at the subtree that was formerly rooted at z and is now rooted at the node we temporarily called b. (See Figure 10.11(b).)

Unfortunately, this trinode restructuring may reduce the height of the subtree rooted at b by 1, which may cause an ancestor of b to become unbalanced. So, after rebalancing z, we continue walking up T looking for unbalanced nodes. If we find another, we perform a restructure operation to restore its balance, and continue marching up T looking for more, all the way to the root. Still, since the height of T is $O(\log n)$, where n is the number of entries, by Proposition 10.2, $O(\log n)$ trinode restructurings are sufficient to restore the height-balance property.

Performance of AVL Trees

We summarize the analysis of the performance of an AVL tree T as follows. Operations find, insert, and erase visit the nodes along a root-to-leaf path of T, plus, possibly, their siblings, and spend $O(1)$ time per node. Thus, since the height of T is $O(\log n)$ by Proposition 10.2, each of the above operations takes $O(\log n)$ time. In Table 10.2, we summarize the performance of a map implemented with an AVL tree. We illustrate this performance in Figure 10.12.

Operation	**Time***Time*
size, empty	$O(1)$
find, insert, erase	$O(\log n)$

Table 10.2: Performance of an n-entry map realized by an AVL tree. The space usage is $O(n)$.
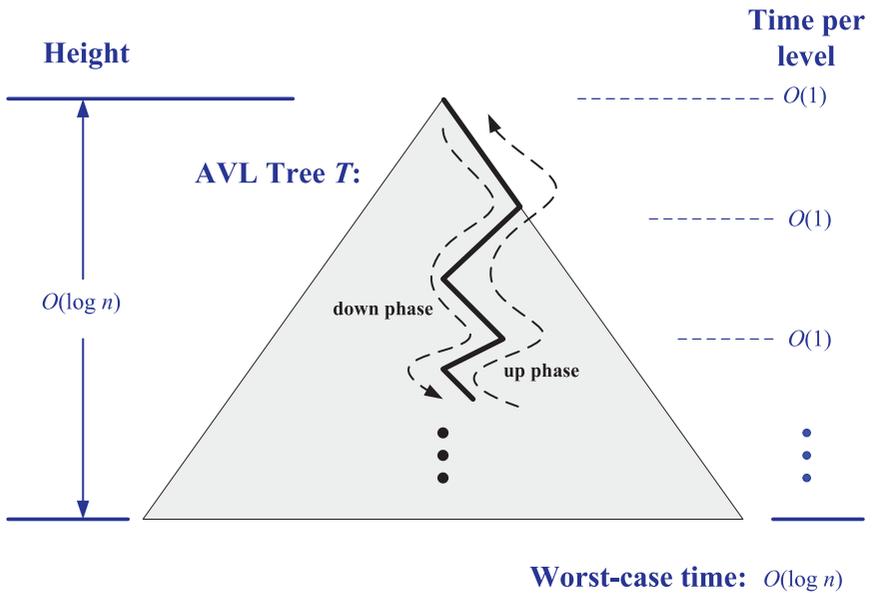
Figure 10.12: Illustrating the running time of searches and updates in an AVL tree. The time performance is $O(1)$ per level, broken into a down phase, which typically involves searching, and an up phase, which typically involves updating height values and performing local trinode restructurings (rotations).

10.2.2   C++ Implementation of an AVL Tree

Let us now turn to the implementation details and analysis of using an AVL tree T with n internal nodes to implement an ordered dictionary of n entries. The insertion and removal algorithms for T require that we are able to perform trinode restructurings and determine the difference between the heights of two sibling nodes. Regarding restructurings, we now need to make sure our underlying implementation of a binary search tree includes the method restructure(x), which performs a trinode restructuring operation (Code Fragment 10.12). (We do not provide an implementation of this function, but it is a straightforward addition to the linked binary tree class given in Section 7.3.4.) It is easy to see that a restructure operation can be performed in $O(1)$ time if T is implemented with a linked structure. We assume that the SearchTree class includes this function.

Regarding height information, we have chosen to store the height of each internal node, v, explicitly in each node. Alternatively, we could have stored the **balance factor** of v at v, which is defined as the height of the left child of v minus the height of the right child of v. Thus, the balance factor of v is always equal to -1, 0, or 1, except during an insertion or removal, when it may become **temporarily** equal to -2 or $+2$. During the execution of an insertion or removal, the heights and balance factors of $O(\log n)$ nodes are affected and can be maintained in $O(\log n)$ time.

In order to store the height information, we derive a subclass, called AVLEntry, from the standard entry class given earlier in Code Fragment 10.3. It is templated with the base entry type, from which it inherits the key and value members. It defines a member variable *ht*, which stores the height of the subtree rooted at the associated node. It provides member functions for accessing and setting this value. These functions are protected, so that a user cannot access them, but AVLTree can.

```
template <typename E>
class AVLEntry : public E {                    // an AVL entry
private:
  int ht;                                      // node height
protected:                                     // local types
  typedef typename E::Key K;                   // key type
  typedef typename E::Value V;                 // value type
  int height() const { return ht; }            // get height
  void setHeight(int h) { ht = h; }            // set height
public:                                        // public functions
  AVLEntry(const K& k = K(), const V& v = V()) // constructor
    : E(k,v), ht(0) { }
  friend class AVLTree<E>;                      // allow AVLTree access
};
```

Code Fragment 10.13: An enhanced key-value entry for class AVLTree, containing the height of the associated node.

In Code Fragment 10.14, we present the class definition for AVLTree. This class is derived from the class SearchTree, but using our enhanced AVLEntry in order to maintain height information for the nodes of the tree. The class defines a number of typedef shortcuts for referring to entities such as keys, values, and tree positions. The class declares all the standard dictionary public member functions. At the end, it also defines a number of protected utility functions, which are used in maintaining the AVL tree balance properties.

```
template <typename E>                           // an AVL tree
class AVLTree : public SearchTree< AVLEntry<E> > {
public:                                         // public types
  typedef AVLEntry<E> AVLEntry;                 // an entry
  typedef typename SearchTree<AVLEntry>::Iterator Iterator; // an iterator
protected:                                      // local types
  typedef typename AVLEntry::Key K;             // a key
  typedef typename AVLEntry::Value V;           // a value
  typedef SearchTree<AVLEntry> ST;              // a search tree
  typedef typename ST::TPos TPos;               // a tree position
public:                                         // public functions
  AVLTree();                                    // constructor
  Iterator insert(const K& k, const V& x);      // insert (k,x)
  void erase(const K& k) throw(NonexistentElement); // remove key k entry
  void erase(const Iterator& p);                // remove entry at p
protected:                                      // utility functions
  int height(const TPos& v) const;             // node height utility
  void setHeight(TPos v);                       // set height utility
  bool isBalanced(const TPos& v) const;         // is v balanced?
  TPos tallGrandchild(const TPos& v) const;     // get tallest grandchild
  void rebalance(const TPos& v);                // rebalance utility
};
```

Code Fragment 10.14: Class AVLTree, an AVL tree implementation of a dictionary.

Next, in Code Fragment 10.15, we present the constructor and height utility function. The constructor simply invokes the constructor for the binary search tree, which creates a tree having no entries. The function height returns the height of a node, by extracting the height information from the AVLEntry. We employ the condensed function notation that we introduced in Section 9.2.7.

```
/* AVLTree⟨E⟩ :: */                             // constructor
  AVLTree() : ST() { }


/* AVLTree⟨E⟩ :: */                             // node height utility
  int height(const TPos& v) const
    { return (v.isExternal() ? 0 : v−>height()); }
```

Code Fragment 10.15: The constructor for class AVLTree and a utility for extracting heights.

In Code Fragment 10.16, we present a few utility functions needed for maintaining the tree's balance. The function setHeight sets the height information for a node as one more than the maximum of the heights of its two children. The function isBalanced determines whether a node satisfies the AVL balance condition, by checking that the height difference between its children is at most 1. Finally, the function tallGrandchild determines the tallest grandchild of a node. Recall that this procedure is needed by the removal operation to determine the node to which the restructuring operation will be applied.

```
/* AVLTree⟨E⟩ :: */                                    // set height utility
  void setHeight(TPos v) {
    int hl = height(v.left());
    int hr = height(v.right());
    v->setHeight(1 + std::max(hl, hr));                // max of left & right
  }

/* AVLTree⟨E⟩ :: */                                    // is v balanced?
  bool isBalanced(const TPos& v) const {
    int bal = height(v.left()) − height(v.right());
    return ((−1 <= bal) && (bal <= 1));
  }

/* AVLTree⟨E⟩ :: */                                    // get tallest grandchild
  TPos tallGrandchild(const TPos& z) const {
    TPos zl = z.left();
    TPos zr = z.right();
    if (height(zl) >= height(zr))                      // left child taller
      if (height(zl.left()) >= height(zl.right()))
        return zl.left();
      else
        return zl.right();
    else                                               // right child taller
      if (height(zr.right()) >= height(zr.left()))
        return zr.right();
      else
        return zr.left();
  }
```

Code Fragment 10.16: Some utility functions used for maintaining balance in the AVL tree.

Next, we present the principal function for rebalancing the AVL tree after an insertion or removal. The procedure starts at the node *v* affected by the operation. It then walks up the tree to the root level. On visiting each node *z*, it updates *z*'s height information (which may have changed due to the update operation) and

checks whether *z* is balanced. If not, it finds *z*'s tallest grandchild, and applies the restructuring operation to this node. Since heights may have changed as a result, it updates the height information for *z*'s children and itself.

```
/* AVLTree⟨E⟩ :: */                                    // rebalancing utility
  void rebalance(const TPos& v) {
    TPos z = v;
    while (!(z == ST::root())) {                        // rebalance up to root
      z = z.parent();
      setHeight(z);                                     // compute new height
      if (!isBalanced(z)) {                             // restructuring needed
        TPos x = tallGrandchild(z);
        z = restructure(x);                             // trinode restructure
        setHeight(z.left());                            // update heights
        setHeight(z.right());
        setHeight(z);
      }
    }
  }
```

Code Fragment 10.17: Rebalancing the tree after an update operation.

Finally, in Code Fragment 10.18, we present the functions for inserting and erasing keys. (We have omitted the iterator-based erase function, since it is very simple.) Each invokes the associated utility function (inserter or eraser, respectively) from the base class SearchTree. Each then invokes rebalance to restore balance to the tree.

```
/* AVLTree⟨E⟩ :: */                                    // insert (k,x)
  Iterator insert(const K& k, const V& x) {
    TPos v = inserter(k, x);                            // insert in base tree
    setHeight(v);                                       // compute its height
    rebalance(v);                                       // rebalance if needed
    return Iterator(v);
  }

/* AVLTree⟨E⟩ :: */                                    // remove key k entry
  void erase(const K& k) throw(NonexistentElement) {
    TPos v = finder(k, ST::root());                     // find in base tree
    if (Iterator(v) == ST::end())                       // not found?
      throw NonexistentElement("Erase of nonexistent");
    TPos w = eraser(v);                                 // remove it
    rebalance(w);                                       // rebalance if needed
  }
```

Code Fragment 10.18: The insertion and erasure functions.

10.3 Splay Trees

Another way we can implement the fundamental map operations is to use a balanced search tree data structure known as a *splay tree*. This structure is conceptually quite different from the other balanced search trees we discuss in this chapter, for a splay tree does not use any explicit rules to enforce its balance. Instead, it applies a certain move-to-root operation, called *splaying*, after every access, in order to keep the search tree balanced in an amortized sense. The splaying operation is performed at the bottom-most node x reached during an insertion, deletion, or even a search. The surprising thing about splaying is that it allows us to guarantee an amortized running time for insertions, deletions, and searches, that is logarithmic. The structure of a *splay tree* is simply a binary search tree T. In fact, there are no additional height, balance, or color labels that we associate with the nodes of this tree.

10.3.1 Splaying

Given an internal node x of a binary search tree T, we *splay* x by moving x to the root of T through a sequence of restructurings. The particular restructurings we perform are important, for it is not sufficient to move x to the root of T by just any sequence of restructurings. The specific operation we perform to move x up depends upon the relative positions of x, its parent y, and (if it exists) x's grandparent z. There are three cases that we consider.

zig-zig: The node x and its parent y are both left children or both right children. (See Figure 10.13.) We replace z by x, making y a child of x and z a child of y, while maintaining the inorder relationships of the nodes in T.

(a)   (b)
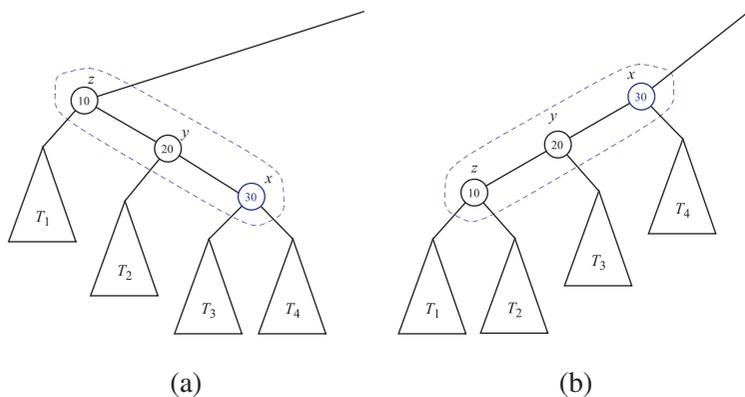
Figure 10.13: Zig-zig: (a) before; (b) after. There is another symmetric configuration where x and y are left children.

***zig-zag*:** One of x and y is a left child and the other is a right child. (See Figure 10.14.) In this case, we replace z by x and make x have y and z as its children, while maintaining the inorder relationships of the nodes in T.

(a)   (b)
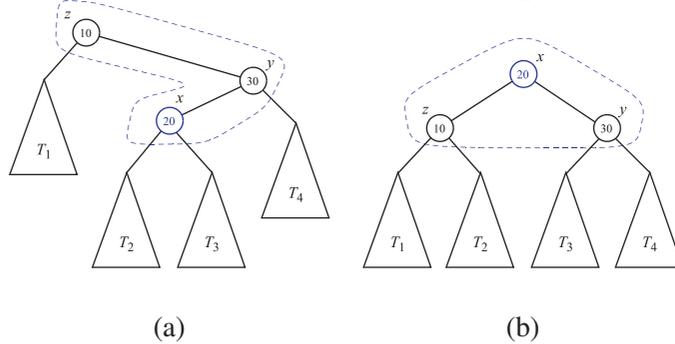
Figure 10.14: Zig-zag: (a) before; (b) after. There is another symmetric configuration where x is a right child and y is a left child.

***zig*:** x does not have a grandparent (or we are not considering x's grandparent for some reason). (See Figure 10.15.) In this case, we rotate x over y, making x's children be the node y and one of x's former children w, in order to maintain the relative inorder relationships of the nodes in T.

(a)   (b)
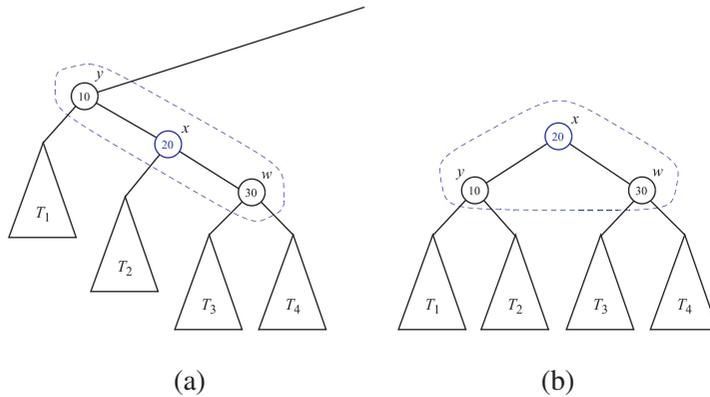
Figure 10.15: Zig: (a) before; (b) after. There is another symmetric configuration where x and w are left children.

We perform a zig-zig or a zig-zag when x has a grandparent, and we perform a zig when x has a parent but not a grandparent. A ***splaying*** step consists of repeating these restructurings at x until x becomes the root of T. Note that this is not the same as a sequence of simple rotations that brings x to the root. An example of the splaying of a node is shown in Figures 10.16 and 10.17.
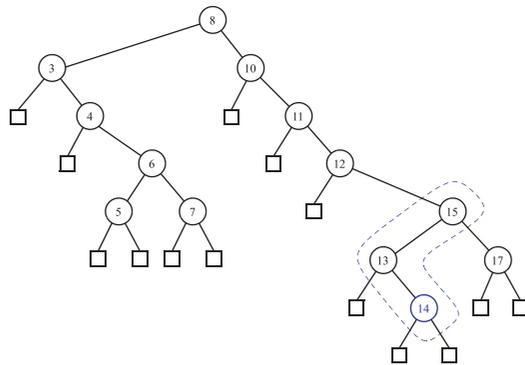
(a)
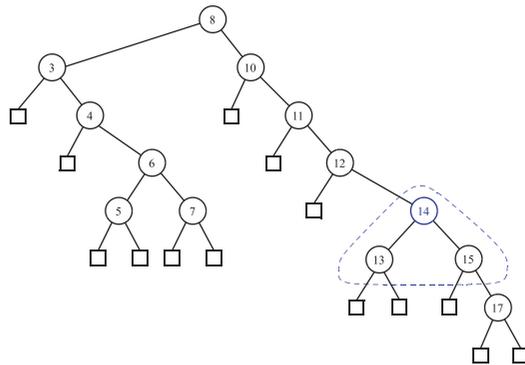
(b)
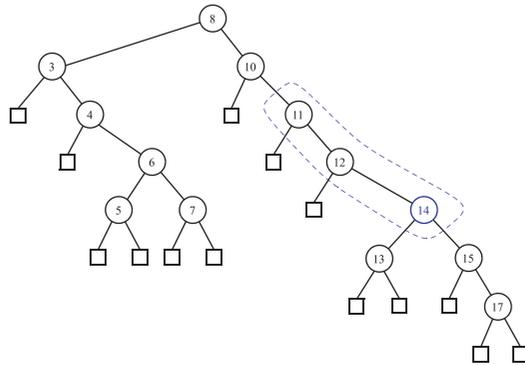
(c)

Figure 10.16: Example of splaying a node: (a) splaying the node storing 14 starts with a zig-zag; (b) after the zig-zag; (c) the next step is a zig-zig. (Continues in Figure 10.17.)
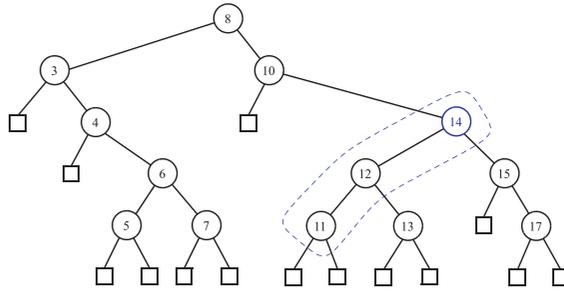
(d)
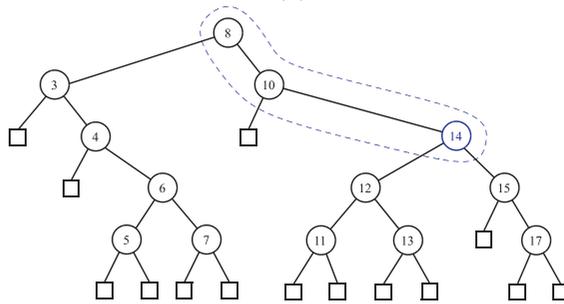
(e)
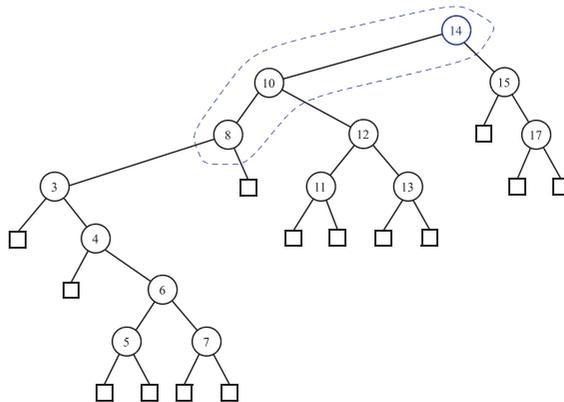
(f)

Figure 10.17: Example of splaying a node: (d) after the zig-zig; (e) the next step is again a zig-zig; (f) after the zig-zig (Continued from Figure 10.17.)

10.3.2   When to Splay

The rules that dictate when splaying is performed are as follows:

- When searching for key k, if k is found at a node x, we splay x, else we splay the parent of the external node at which the search terminates unsuccessfully. For example, the splaying in Figures 10.16 and 10.17 would be performed after searching successfully for key 14 or unsuccessfully for key 14.5.
- When inserting key k, we splay the newly created internal node where k gets inserted. For example, the splaying in Figures 10.16 and 10.17 would be performed if 14 were the newly inserted key. We show a sequence of insertions in a splay tree in Figure 10.18.
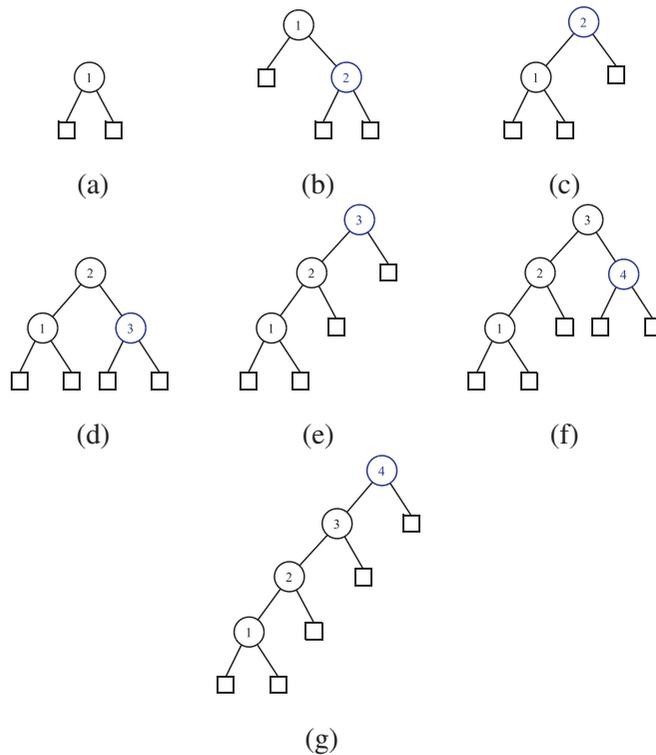
Figure 10.18: A sequence of insertions in a splay tree: (a) initial tree; (b) after inserting 2; (c) after splaying; (d) after inserting 3; (e) after splaying; (f) after inserting 4; (g) after splaying.

- When deleting a key k, we splay the parent of the node w that gets removed, that is, w is either the node storing k or one of its descendents. (Recall the removal algorithm for binary search trees.) An example of splaying following a deletion is shown in Figure 10.19.
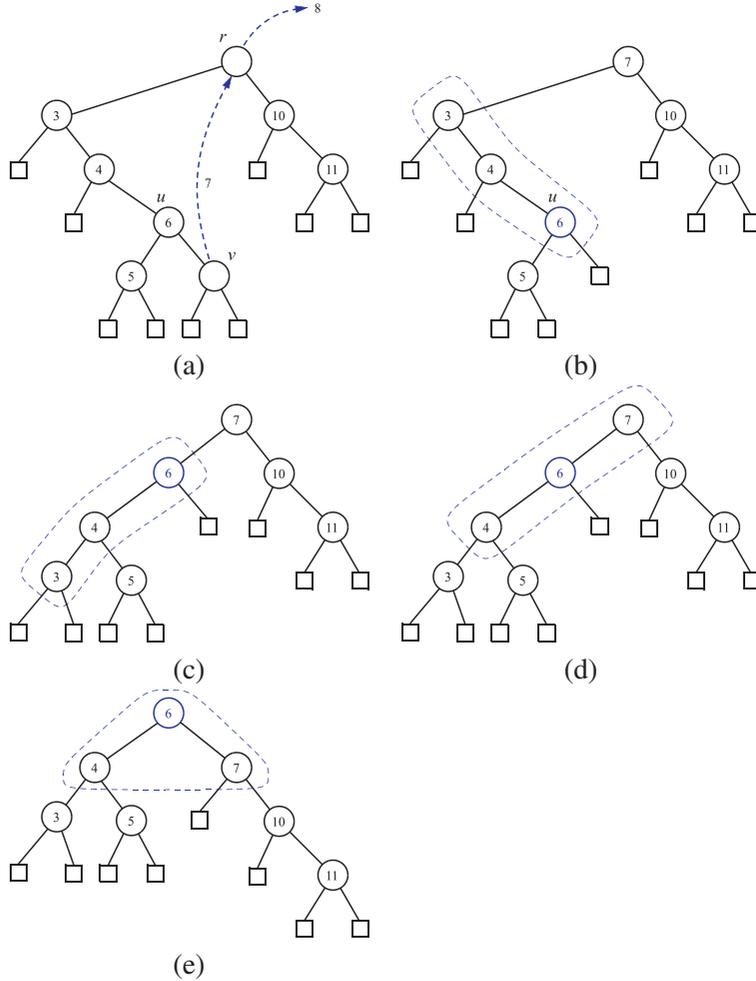
Figure 10.19: Deletion from a splay tree: (a) the deletion of 8 from node r is performed by moving the key of the right-most internal nodr v to r, in the left subtree of r, deleting v, and splaying the parent u of v; (b) splaying u starts with a zig-zig; (c) after the zig-zig; (d) the next step is a zig; (e) after the zig.

10.3.3   Amortized Analysis of Splaying ⋆

After a zig-zig or zig-zag, the depth of x decreases by two, and after a zig the depth of x decreases by one. Thus, if x has depth d, splaying x consists of a sequence of $\lfloor d/2 \rfloor$ zig-zigs and/or zig-zags, plus one final zig if d is odd. Since a single zig-zig, zig-zag, or zig effects a constant number of nodes, it can be done in $O(1)$ time. Thus, splaying a node x in a binary search tree T takes time $O(d)$, where d is the depth of x in T. In other words, the time for performing a splaying step for a node x is asymptotically the same as the time needed just to reach that node in a top-down search from the root of T.

Worst-Case Time

In the worst case, the overall running time of a search, insertion, or deletion in a splay tree of height h is $O(h)$, since the node we splay might be the deepest node in the tree. Moreover, it is possible for h to be as large as n, as shown in Figure 10.18. Thus, from a worst-case point of view, a splay tree is not an attractive data structure.

In spite of its poor worst-case performance, a splay tree performs well in an amortized sense. That is, in a sequence of intermixed searches, insertions, and deletions, each operation takes, on average, logarithmic time. We perform the amortized analysis of splay trees using the accounting method.

Amortized Performance of Splay Trees

For our analysis, we note that the time for performing a search, insertion, or deletion is proportional to the time for the associated splaying. So let us consider only splaying time.

Let T be a splay tree with n keys, and let v be a node of T. We define the **size** $n(v)$ of v as the number of nodes in the subtree rooted at v. Note that this definition implies that the size of an internal node is one more than the sum of the sizes of its two children. We define the **rank** $r(v)$ of a node v as the logarithm in base 2 of the size of v, that is, $r(v) = \log(n(v))$. Clearly, the root of T has the maximum size $(2n + 1)$ and the maximum rank, $\log(2n + 1)$, while each external node has size 1 and rank 0.

We use cyber-dollars to pay for the work we perform in splaying a node x in T, and we assume that one cyber-dollar pays for a zig, while two cyber-dollars pay for a zig-zig or a zig-zag. Hence, the cost of splaying a node at depth d is d cyber-dollars. We keep a virtual account storing cyber-dollars at each internal node of T. Note that this account exists only for the purpose of our amortized analysis, and does not need to be included in a data structure implementing the splay tree T.

An Accounting Analysis of Splaying

When we perform a splaying, we pay a certain number of cyber-dollars (the exact value of the payment will be determined at the end of our analysis). We distinguish three cases:

- If the payment is equal to the splaying work, then we use it all to pay for the splaying.

- If the payment is greater than the splaying work, we deposit the excess in the accounts of several nodes.

- If the payment is less than the splaying work, we make withdrawals from the accounts of several nodes to cover the deficiency.

We show below that a payment of $O(\log n)$ cyber-dollars per operation is sufficient to keep the system working, that is, to ensure that each node keeps a nonnegative account balance.

An Accounting Invariant for Splaying

We use a scheme in which transfers are made between the accounts of the nodes to ensure that there will always be enough cyber-dollars to withdraw for paying for splaying work when needed.

In order to use the accounting method to perform our analysis of splaying, we maintain the following invariant:

> *Before and after a splaying, each node v of T has r(v) cyber-dollars in its account.*

Note that the invariant is "financially sound," since it does not require us to make a preliminary deposit to endow a tree with zero keys.

Let $r(T)$ be the sum of the ranks of all the nodes of T. To preserve the invariant after a splaying, we must make a payment equal to the splaying work plus the total change in $r(T)$. We refer to a single zig, zig-zig, or zig-zag operation in a splaying as a splaying *substep*. Also, we denote the rank of a node v of T before and after a splaying substep with $r(v)$ and $r'(v)$, respectively. The following proposition gives an upper bound on the change of $r(T)$ caused by a single splaying substep. We repeatedly use this lemma in our analysis of a full splaying of a node to the root.

Proposition 10.3: *Let δ be the variation of $r(T)$ caused by a single splaying substep (a zig, zig-zig, or zig-zag) for a node x in T. We have the following:*

- *$\delta \le 3(r'(x) - r(x)) - 2$ if the substep is a zig-zig or zig-zag*
- *$\delta \le 3(r'(x) - r(x))$ if the substep is a zig*

Justification: We use the fact (see Proposition A.1, Appendix A) that, if $a > 0$, $b > 0$, and $c > a + b$,
$$\log a + \log b \le 2\log c - 2. \tag{10.6}$$

Let us consider the change in $r(T)$ caused by each type of splaying substep.

zig-zig: (Recall Figure 10.13.) Since the size of each node is one more than the size of its two children, note that only the ranks of x, y, and z change in a zig-zig operation, where y is the parent of x and z is the parent of y. Also, $r'(x) = r(z)$, $r'(y) \le r'(x)$, and $r(y) \ge r(x)$. Thus

$$
\begin{aligned}
\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
&\le r'(y) + r'(z) - r(x) - r(y) \\
&\le r'(x) + r'(z) - 2r(x). \tag{10.7}
\end{aligned}
$$

Note that $n(x) + n'(z) \le n'(x)$. By 10.6, $r(x) + r'(z) \le 2r'(x) - 2$, that is,
$$r'(z) \le 2r'(x) - r(x) - 2.$$

This inequality and 10.7 imply

$$
\begin{aligned}
\delta &\le r'(x) + (2r'(x) - r(x) - 2) - 2r(x) \\
&\le 3(r'(x) - r(x)) - 2.
\end{aligned}
$$

zig-zag: (Recall Figure 10.14.) Again, by the definition of size and rank, only the ranks of x, y, and z change, where y denotes the parent of x and z denotes the parent of y. Also, $r'(x) = r(z)$ and $r(x) \le r(y)$. Thus

$$
\begin{aligned}
\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\
&\le r'(y) + r'(z) - r(x) - r(y) \\
&\le r'(y) + r'(z) - 2r(x). \tag{10.8}
\end{aligned}
$$

Note that $n'(y) + n'(z) \le n'(x)$; hence, by 10.6, $r'(y) + r'(z) \le 2r'(x) - 2$. Thus

$$
\begin{aligned}
\delta &\le 2r'(x) - 2 - 2r(x) \\
&\le 3(r'(x) - r(x)) - 2.
\end{aligned}
$$

zig: (Recall Figure 10.15.) In this case, only the ranks of x and y change, where y denotes the parent of x. Also, $r'(y) \le r(y)$ and $r'(x) \ge r(x)$. Thus

$$
\begin{aligned}
\delta &= r'(y) + r'(x) - r(y) - r(x) \\
&\le r'(x) - r(x) \\
&\le 3(r'(x) - r(x)).
\end{aligned}
$$
∎

Proposition 10.4: *Let T be a splay tree with root t, and let Δ be the total variation of $r(T)$ caused by splaying a node x at depth d. We have*

$$\Delta \le 3(r(t) - r(x)) - d + 2.$$

Justification: Splaying node x consists of $p = \lceil d/2 \rceil$ splaying substeps, each of which is a zig-zig or a zig-zag, except possibly the last one, which is a zig if d is odd. Let $r_0(x) = r(x)$ be the initial rank of x, and for $i = 1, \ldots, p$, let $r_i(x)$ be the rank of x after the ith substep and δ_i be the variation of $r(T)$ caused by the ith substep. By Lemma 10.3, the total variation Δ of $r(T)$ caused by splaying x is

$$
\begin{aligned}
\Delta &= \sum_{i=1}^{p} \delta_i \\
&\le \sum_{i=1}^{p} (3(r_i(x) - r_{i-1}(x)) - 2) + 2 \\
&= 3(r_p(x) - r_0(x)) - 2p + 2 \\
&\le 3(r(t) - r(x)) - d + 2.
\end{aligned}
$$
■

By Proposition 10.4, if we make a payment of $3(r(t) - r(x)) + 2$ cyber-dollars towards the splaying of node x, we have enough cyber-dollars to maintain the invariant, keeping $r(v)$ cyber-dollars at each node v in T, and pay for the entire splaying work, which costs d dollars. Since the size of the root t is $2n + 1$, its rank $r(t) = \log(2n + 1)$. In addition, we have $r(x) < r(t)$. Thus, the payment to be made for splaying is $O(\log n)$ cyber-dollars. To complete our analysis, we have to compute the cost for maintaining the invariant when a node is inserted or deleted.

When inserting a new node v into a splay tree with n keys, the ranks of all the ancestors of v are increased. Namely, let $v_0, v_i, \ldots, v_d$ be the ancestors of v, where $v_0 = v$, v_i is the parent of v_{i-1}, and v_d is the root. For $i = 1, \ldots, d$, let $n'(v_i)$ and $n(v_i)$ be the size of v_i before and after the insertion, respectively, and let $r'(v_i)$ and $r(v_i)$ be the rank of v_i before and after the insertion, respectively. We have

$$n'(v_i) = n(v_i) + 1.$$

Also, since $n(v_i) + 1 \le n(v_{i+1})$, for $i = 0, 1, \ldots, d - 1$, we have the following for each i in this range

$$r'(v_i) = \log(n'(v_i)) = \log(n(v_i) + 1) \le \log(n(v_{i+1})) = r(v_{i+1}).$$

Thus, the total variation of $r(T)$ caused by the insertion is

$$
\begin{aligned}
\sum_{i=1}^{d} (r'(v_i) - r(v_i)) &\le r'(v_d) + \sum_{i=1}^{d-1} (r(v_{i+1}) - r(v_i)) \\
&= r'(v_d) - r(v_0) \\
&\le \log(2n + 1).
\end{aligned}
$$

Therefore, a payment of $O(\log n)$ cyber-dollars is sufficient to maintain the invariant when a new node is inserted.

When deleting a node v from a splay tree with n keys, the ranks of all the ancestors of v are decreased. Thus, the total variation of $r(T)$ caused by the deletion is negative, and we do not need to make any payment to maintain the invariant when a node is deleted. Therefore, we may summarize our amortized analysis in the following proposition (which is sometimes called the "balance proposition" for splay trees).

Proposition 10.5: *Consider a sequence of m operations on a splay tree, each one a search, insertion, or deletion, starting from a splay tree with zero keys. Also, let n_i be the number of keys in the tree after operation i, and n be the total number of insertions. The total running time for performing the sequence of operations is*

$$O\left(m + \sum_{i=1}^{m} \log n_i\right),$$

which is $O(m \log n)$.

In other words, the amortized running time of performing a search, insertion, or deletion in a splay tree is $O(\log n)$, where n is the size of the splay tree at the time. Thus, a splay tree can achieve logarithmic-time amortized performance for implementing an ordered map ADT. This amortized performance matches the worst-case performance of AVL trees, $(2,4)$ trees, and red-black trees, but it does so using a simple binary tree that does not need any extra balance information stored at each of its nodes. In addition, splay trees have a number of other interesting properties that are not shared by these other balanced search trees. We explore one such additional property in the following proposition (which is sometimes called the "Static Optimality" proposition for splay trees).

Proposition 10.6: *Consider a sequence of m operations on a splay tree, each one a search, insertion, or deletion, starting from a splay tree T with zero keys. Also, let $f(i)$ denote the number of times the entry i is accessed in the splay tree, that is, its frequency, and let n denote the total number of entries. Assuming that each entry is accessed at least once, then the total running time for performing the sequence of operations is*

$$O\left(m + \sum_{i=1}^{n} f(i) \log(m/f(i))\right).$$

We omit the proof of this proposition, but it is not as hard to justify as one might imagine. The remarkable thing is that this proposition states that the amortized running time of accessing an entry i is $O(\log(m/f(i)))$.

10.4 (2,4) Trees

Some data structures we discuss in this chapter, including $(2,4)$ trees, are multi-way search trees, that is, trees with internal nodes that have two or more children. Thus, before we define $(2,4)$ trees, let us discuss multi-way search trees.

10.4.1 Multi-Way Search Trees

Recall that multi-way trees are defined so that each internal node can have many children. In this section, we discuss how multi-way trees can be used as search trees. Recall that the **entries** that we store in a search tree are pairs of the form (k,x), where k is the **key** and x is the value associated with the key. However, we do not discuss how to perform updates in multi-way search trees now, since the details for update methods depend on additional properties we want to maintain for multi-way trees, which we discuss in Section 14.3.1.

Definition of a Multi-way Search Tree

Let v be a node of an ordered tree. We say that v is a *d-node* if v has d children. We define a ***multi-way search tree*** to be an ordered tree T that has the following properties, which are illustrated in Figure 10.1(a):

- Each internal node of T has at least two children. That is, each internal node is a d-node such that $d \geq 2$.
- Each internal d-node v of T with children $v_1, \ldots, v_d$ stores an ordered set of $d-1$ key-value entries $(k_1, x_1), \ldots, (k_{d-1}, x_{d-1})$, where $k_1 \leq \cdots \leq k_{d-1}$.
- Let us conventionally define $k_0 = -\infty$ and $k_d = +\infty$. For each entry (k,x) stored at a node in the subtree of v rooted at v_i, $i = 1, \ldots, d$, we have that $k_{i-1} \leq k \leq k_i$.

That is, if we think of the set of keys stored at v as including the special fictitious keys $k_0 = -\infty$ and $k_d = +\infty$, then a key k stored in the subtree of T rooted at a child node v_i must be "in between" two keys stored at v. This simple viewpoint gives rise to the rule that a d-node stores $d-1$ regular keys, and it also forms the basis of the algorithm for searching in a multi-way search tree.

By the above definition, the external nodes of a multi-way search do not store any entries and serve only as "placeholders," as has been our convention with binary search trees (Section 10.1); hence, a binary search tree can be viewed as a special case of a multi-way search tree, where each internal node stores one entry and has two children. In addition, while the external nodes could be *null*, we make the simplifying assumption here that they are actual nodes that don't store anything.
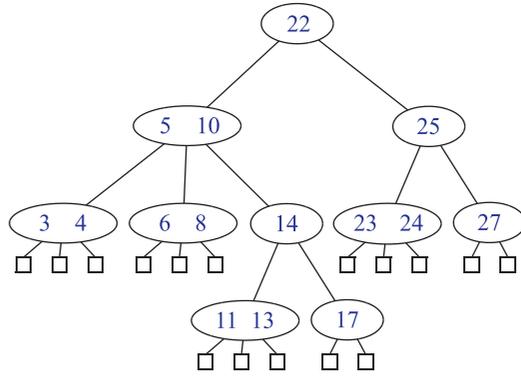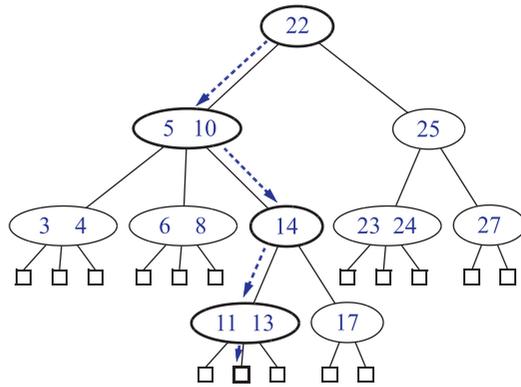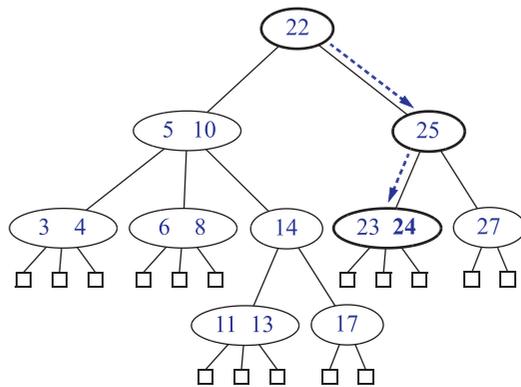
Figure 10.20: (a) A multi-way search tree T; (b) search path in T for key 12 (unsuccessful search); (c) search path in T for key 24 (successful search).

Whether internal nodes of a multi-way tree have two children or many, however, there is an interesting relationship between the number of entries and the number of external nodes.

Proposition 10.7: *An n-entry multi-way search tree has $n+1$ external nodes.*

We leave the justification of this proposition as an exercise (Exercise C-10.17).

Searching in a Multi-Way Tree

Given a multi-way search tree T, we note that searching for an entry with key k is simple. We perform such a search by tracing a path in T starting at the root. (See Figure 10.1(b) and (c).) When we are at a d-node v during this search, we compare the key k with the keys $k_1, \ldots, k_{d-1}$ stored at v. If $k = k_i$ for some i, the search is successfully completed. Otherwise, we continue the search in the child v_i of v such that $k_{i-1} < k < k_i$. (Recall that we conventionally define $k_0 = -\infty$ and $k_d = +\infty$.) If we reach an external node, then we know that there is no entry with key k in T, and the search terminates unsuccessfully.

Data Structures for Representing Multi-way Search Trees

In Section 7.1.4, we discuss a linked data structure for representing a general tree. This representation can also be used for a multi-way search tree. In fact, in using a general tree to implement a multi-way search tree, the only additional information that we need to store at each node is the set of entries (including keys) associated with that node. That is, we need to store with v a reference to some collection that stores the entries for v.

Recall that when we use a binary search tree to represent an ordered map M, we simply store a reference to a single entry at each internal node. In using a multi-way search tree T to represent M, we must store a reference to the ordered set of entries associated with v at each internal node v of T. This reasoning may at first seem like a circular argument, since we need a representation of an ordered map to represent an ordered map. We can avoid any circular arguments, however, by using the ***bootstrapping*** technique, where we use a previous (less advanced) solution to a problem to create a new (more advanced) solution. In this case, bootstrapping consists of representing the ordered set associated with each internal node using a map data structure that we have previously constructed (for example, a search table based on a sorted array, as shown in Section 9.3.1). In particular, assuming we already have a way of implementing ordered maps, we can realize a multi-way search tree by taking a tree T and storing such a map at each node of T.

The map we store at each node v is known as a ***secondary*** data structure, because we are using it to support the bigger, ***primary*** data structure. We denote the map stored at a node v of T as $M(v)$. The entries we store in $M(v)$ allow us to find which child node to move to next during a search operation. Specifically, for each node v of T, with children $v_1, \ldots, v_d$ and entries $(k_1, x_1), \ldots, (k_{d-1}, x_{d-1})$, we store, in the map $M(v)$, the entries

$$(k_1, (x_1, v_1)), (k_2, (x_2, v_2)), \ldots, (k_{d-1}, (x_{d-1}, v_{d-1})), (+\infty, (\emptyset, v_d)).$$

That is, an entry $(k_i, (x_i, v_i))$ of map $M(v)$ has key k_i and value (x_i, v_i). Note that the last entry stores the special key $+\infty$.

With the realization of the multi-way search tree T above, processing a d-node v while searching for an entry of T with key k can be done by performing a search operation to find the entry $(k_i, (x_i, v_i))$ in $M(v)$ with smallest key greater than or equal to k. We distinguish two cases:

- If $k < k_i$, then we continue the search by processing child v_i. (Note that if the special key $k_d = +\infty$ is returned, then k is greater than all the keys stored at node v, and we continue the search processing child v_d.)

- Otherwise ($k = k_i$), then the search terminates successfully.

Consider the space requirement for the above realization of a multi-way search tree T storing n entries. By Proposition 10.7, using any of the common realizations of an ordered map (Chapter 9) for the secondary structures of the nodes of T, the overall space requirement for T is $O(n)$.

Consider next the time spent answering a search in T. The time spent at a d-node v of T during a search depends on how we realize the secondary data structure $M(v)$. If $M(v)$ is realized with a sorted array (that is, an ordered search table), then we can process v in $O(\log d)$ time. If $M(v)$ is realized using an unsorted list instead, then processing v takes $O(d)$ time. Let $d_{\max}$ denote the maximum number of children of any node of T, and let h denote the height of T. The search time in a multi-way search tree is either $O(hd_{\max})$ or $O(h\log d_{\max})$, depending on the specific implementation of the secondary structures at the nodes of T (the map $M(v)$). If $d_{\max}$ is a constant, the running time for performing a search is $O(h)$, irrespective of the implementation of the secondary structures.

Thus, the primary efficiency goal for a multi-way search tree is to keep the height as small as possible, that is, we want h to be a logarithmic function of n, the total number of entries stored in the map. A search tree with logarithmic height such as this is called a ***balanced search tree***.

Definition of a $(2,4)$ Tree

A multi-way search tree that keeps the secondary data structures stored at each node small and also keeps the primary multi-way tree balanced is the **(2,4)** *tree*, which is sometimes called 2-4 tree or 2-3-4 tree. This data structure achieves these goals by maintaining two simple properties (see Figure 10.21):

Size Property: Every internal node has at most four children

Depth Property: All the external nodes have the same depth
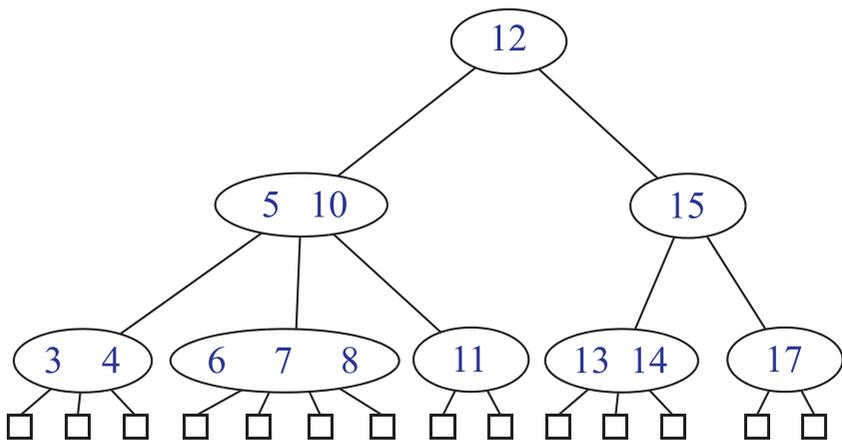
Figure 10.21: A $(2,4)$ tree.

Again, we assume that external nodes are empty and, for the sake of simplicity, we describe our search and update methods assuming that external nodes are real nodes, although this latter requirement is not strictly needed.

Enforcing the size property for $(2,4)$ trees keeps the nodes in the multi-way search tree simple. It also gives rise to the alternative name "2-3-4 tree," since it implies that each internal node in the tree has 2, 3, or 4 children. Another implication of this rule is that we can represent the map $M(v)$ stored at each internal node v using an unordered list or an ordered array, and still achieve $O(1)$-time performance for all operations (since $d_{max} = 4$). The depth property, on the other hand, enforces an important bound on the height of a $(2,4)$ tree.

Proposition 10.8: *The height of a $(2,4)$ tree storing n entries is $O(\log n)$.*

Justification: Let h be the height of a $(2,4)$ tree T storing n entries. We justify the proposition by showing that the claims

$$\frac{1}{2}\log(n+1) \le h \tag{10.9}$$

and

$$h \le \log(n+1) \tag{10.10}$$

are true.

   To justify these claims note first that, by the size property, we can have at most 4 nodes at depth 1, at most 4^2 nodes at depth 2, and so on. Thus, the number of external nodes in T is at most 4^h. Likewise, by the depth property and the definition of a $(2,4)$ tree, we must have at least 2 nodes at depth 1, at least 2^2 nodes at depth 2, and so on. Thus, the number of external nodes in T is at least 2^h. In addition, by Proposition 10.7, the number of external nodes in T is $n+1$. Therefore, we obtain

$$2^h \le n+1$$

and

$$n+1 \le 4^h.$$

Taking the logarithm in base 2 of each of the above terms, we get that

$$h \le \log(n+1)$$

and

$$\log(n+1) \le 2h,$$

which justifies our claims (10.9 and 10.10).   ∎

   Proposition 10.8 states that the size and depth properties are sufficient for keeping a multi-way tree balanced (Section 10.4.1). Moreover, this proposition implies that performing a search in a $(2,4)$ tree takes $O(\log n)$ time and that the specific realization of the secondary structures at the nodes is not a crucial design choice, since the maximum number of children d_{max} is a constant (4). We can, for example, use a simple ordered map implementation, such as an array-list search table, for each secondary structure.

10.4.2 Update Operations for (2,4) Trees

Maintaining the size and depth properties requires some effort after performing insertions and removals in a $(2,4)$ tree, however. We discuss these operations next.

Insertion

To insert a new entry (k,x), with key k, into a $(2,4)$ tree T, we first perform a search for k. Assuming that T has no entry with key k, this search terminates unsuccessfully at an external node z. Let v be the parent of z. We insert the new entry into node v and add a new child w (an external node) to v on the left of z. That is, we add entry (k,x,w) to the map $M(v)$.

Our insertion method preserves the depth property, since we add a new external node at the same level as existing external nodes. Nevertheless, it may violate the size property. Indeed, if a node v was previously a 4-node, then it may become a 5-node after the insertion, which causes the tree T to no longer be a $(2,4)$ tree. This type of violation of the size property is called an *overflow* at node v, and it must be resolved in order to restore the properties of a $(2,4)$ tree. Let $v_1, \ldots, v_5$ be the children of v, and let $k_1, \ldots, k_4$ be the keys stored at v. To remedy the overflow at node v, we perform a *split* operation on v as follows (see Figure 10.22):

- Replace v with two nodes v' and v'', where
    - v' is a 3-node with children v_1, v_2, v_3 storing keys k_1 and k_2
    - v'' is a 2-node with children v_4, v_5 storing key k_4

- If v was the root of T, create a new root node u; else, let u be the parent of v
- Insert key k_3 into u and make v' and v'' children of u, so that if v was child i of u, then v' and v'' become children i and $i+1$ of u, respectively

We show a sequence of insertions in a $(2,4)$ tree in Figure 10.23.
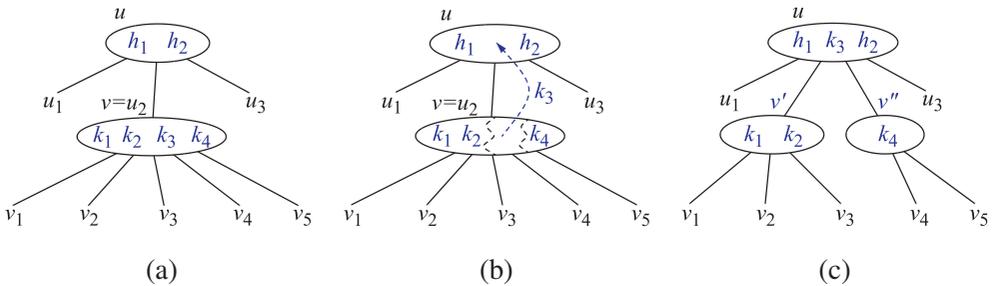
Figure 10.22: A node split: (a) overflow at a 5-node v; (b) the third key of v inserted into the parent u of v; (c) node v replaced with a 3-node v' and a 2-node v''.
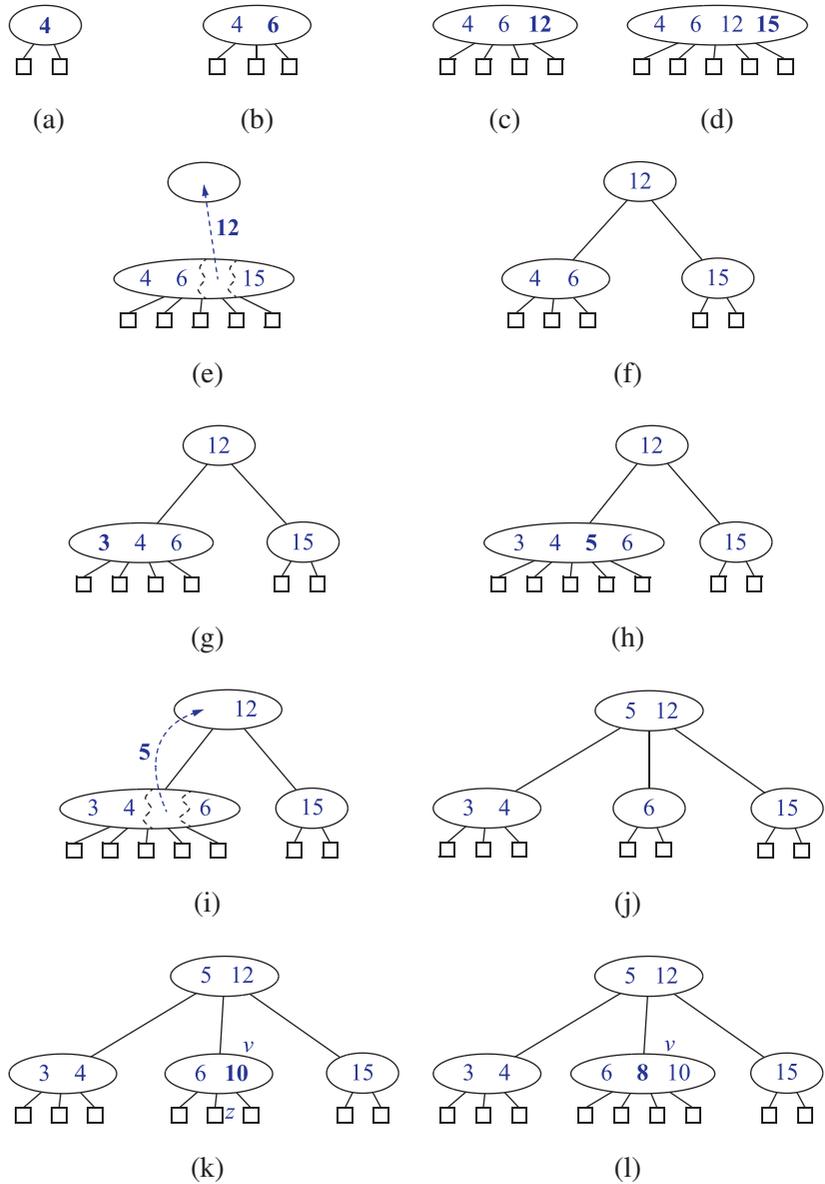
Figure 10.23: A sequence of insertions into a $(2,4)$ tree: (a) initial tree with one entry; (b) insertion of 6; (c) insertion of 12; (d) insertion of 15, which causes an overflow; (e) split, which causes the creation of a new root node; (f) after the split; (g) insertion of 3; (h) insertion of 5, which causes an overflow; (i) split; (j) after the split; (k) insertion of 10; (l) insertion of 8.

Analysis of Insertion in a (2,4) Tree

A split operation affects a constant number of nodes of the tree and $O(1)$ entries stored at such nodes. Thus, it can be implemented to run in $O(1)$ time.

As a consequence of a split operation on node v, a new overflow may occur at the parent u of v. If such an overflow occurs, it triggers a split at node u in turn. (See Figure 10.24.) A split operation either eliminates the overflow or propagates it into the parent of the current node. Hence, the number of split operations is bounded by the height of the tree, which is $O(\log n)$ by Proposition 10.8. Therefore, the total time to perform an insertion in a (2,4) tree is $O(\log n)$.
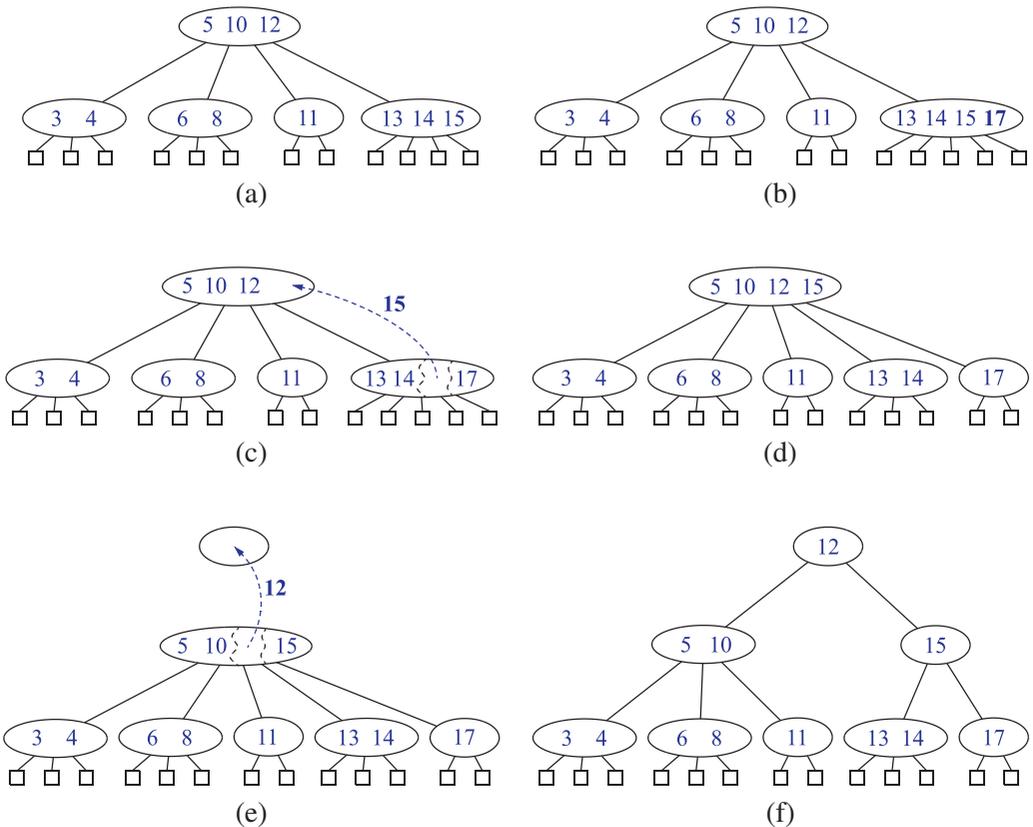
Figure 10.24: An insertion in a (2,4) tree that causes a cascading split: (a) before the insertion; (b) insertion of 17, causing an overflow; (c) a split; (d) after the split a new overflow occurs; (e) another split, creating a new root node; (f) final tree.

Removal

Let us now consider the removal of an entry with key k from a $(2,4)$ tree T. We begin such an operation by performing a search in T for an entry with key k. Removing such an entry from a $(2,4)$ tree can always be reduced to the case where the entry to be removed is stored at a node v whose children are external nodes. Suppose, for instance, that the entry with key k that we wish to remove is stored in the ith entry (k_i, x_i) at a node z that has only internal-node children. In this case, we swap the entry (k_i, x_i) with an appropriate entry that is stored at a node v with external-node children as follows (see Figure 10.25(d)):

1. We find the right-most internal node v in the subtree rooted at the ith child of z, noting that the children of node v are all external nodes.

2. We swap the entry (k_i, x_i) at z with the last entry of v.

Once we ensure that the entry to remove is stored at a node v with only external-node children (because either it was already at v or we swapped it into v), we simply remove the entry from v (that is, from the map $M(v)$) and remove the ith external node of v.

Removing an entry (and a child) from a node v as described above preserves the depth property, because we always remove an external node child from a node v with only external-node children. However, in removing such an external node we may violate the size property at v. Indeed, if v was previously a 2-node, then it becomes a 1-node with no entries after the removal (Figure 10.25(d) and (e)), which is not allowed in a $(2,4)$ tree. This type of violation of the size property is called an ***underflow*** at node v. To remedy an underflow, we check whether an immediate sibling of v is a 3-node or a 4-node. If we find such a sibling w, then we perform a ***transfer*** operation, in which we move a child of w to v, a key of w to the parent u of v and w, and a key of u to v. (See Figure 10.25(b) and (c).) If v has only one sibling, or if both immediate siblings of v are 2-nodes, then we perform a ***fusion*** operation, in which we merge v with a sibling, creating a new node v', and move a key from the parent u of v to v'. (See Figure 10.26(e) and (f).)

A fusion operation at node v may cause a new underflow to occur at the parent u of v, which in turn triggers a transfer or fusion at u. (See Figure 10.26.) Hence, the number of fusion operations is bounded by the height of the tree, which is $O(\log n)$ by Proposition 10.8. If an underflow propagates all the way up to the root, then the root is simply deleted. (See Figure 10.26(c) and (d).) We show a sequence of removals from a $(2,4)$ tree in Figures 10.25 and 10.26.
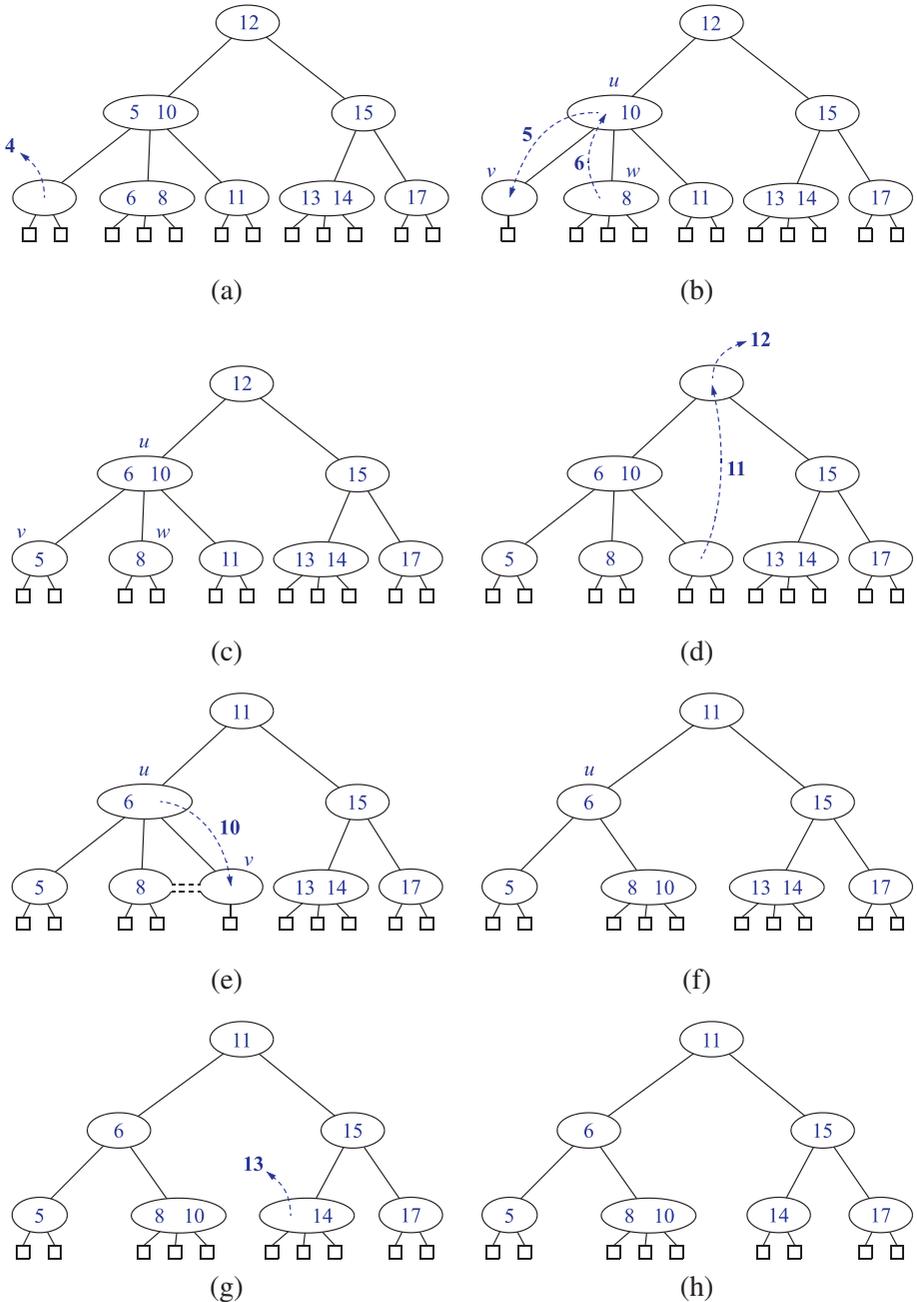
Figure 10.25: A sequence of removals from a $(2,4)$ tree: (a) removal of 4, causing an underflow; (b) a transfer operation; (c) after the transfer operation; (d) removal of 12, causing an underflow; (e) a fusion operation; (f) after the fusion operation; (g) removal of 13; (h) after removing 13.
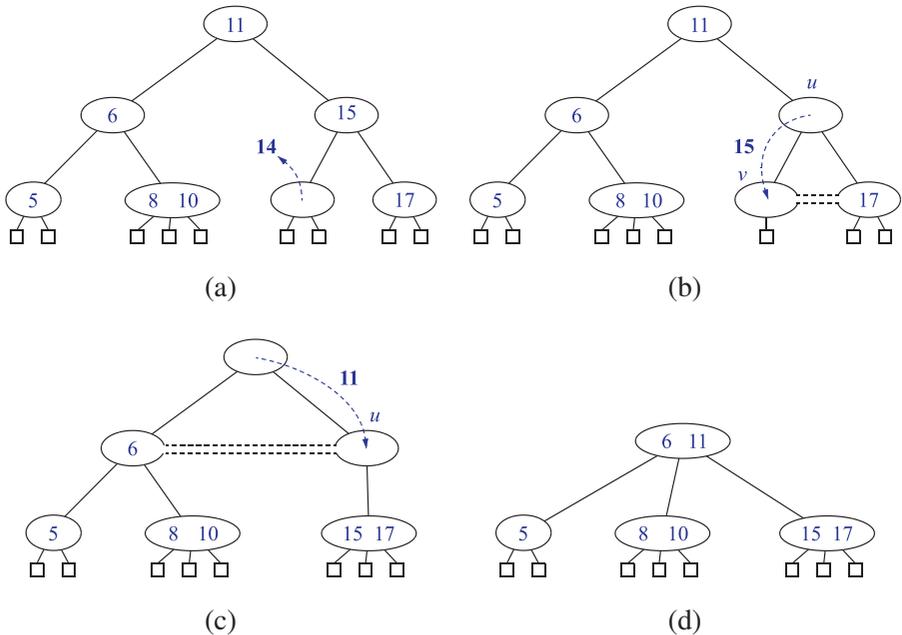
Figure 10.26: A propagating sequence of fusions in a $(2,4)$ tree: (a) removal of 14, which causes an underflow; (b) fusion, which causes another underflow; (c) second fusion operation, which causes the root to be removed; (d) final tree.

Performance of $(2,4)$ Trees

Table 10.3 summarizes the running times of the main operations of a map realized with a $(2,4)$ tree. The time complexity analysis is based on the following:

- The height of a $(2,4)$ tree storing n entries is $O(\log n)$, by Proposition 10.8
- A split, transfer, or fusion operation takes $O(1)$ time
- A search, insertion, or removal of an entry visits $O(\log n)$ nodes.

Operation	Time
size, empty	$O(1)$
find, insert, erase	$O(\log n)$

Table 10.3: Performance of an n-entry map realized by a $(2,4)$ tree. The space usage is $O(n)$.

Thus, $(2,4)$ trees provide for fast map search and update operations. $(2,4)$ trees also have an interesting relationship to the data structure we discuss next.

10.5 Red-Black Trees

Although AVL trees and $(2,4)$ trees have a number of nice properties, there are some map applications for which they are not well suited. For instance, AVL trees may require many restructure operations (rotations) to be performed after a removal, and $(2,4)$ trees may require many fusing or split operations to be performed after either an insertion or removal. The data structure we discuss in this section, the red-black tree, does not have these drawbacks, however, as it requires that only $O(1)$ structural changes be made after an update in order to stay balanced.

A **red-black tree** is a binary search tree (see Section 10.1) with nodes colored red and black in a way that satisfies the following properties:

Root Property: The root is black.

External Property: Every external node is black.

Internal Property: The children of a red node are black.

Depth Property: All the external nodes have the same **black depth**, defined as the number of black ancestors minus one. (Recall that a node is an ancestor of itself.)

An example of a red-black tree is shown in Figure 10.27.
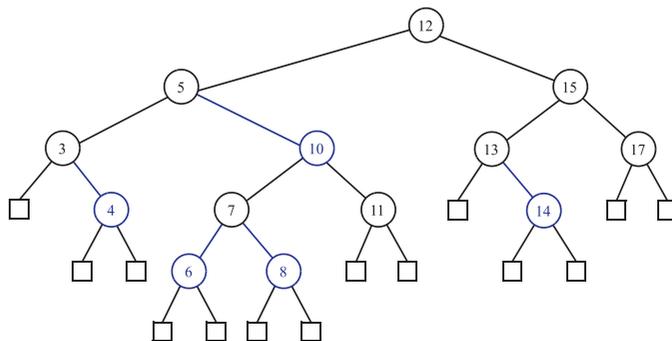
Figure 10.27: Red-black tree associated with the $(2,4)$ tree of Figure 10.21. Each external node of this red-black tree has 4 black ancestors (including itself); hence, it has black depth 3. We use the color blue instead of red. Also, we use the convention of giving an edge of the tree the same color as the child node.

As for previous types of search trees, we assume that entries are stored at the internal nodes of a red-black tree, with the external nodes being empty placeholders. Also, we assume that the external nodes are actual nodes, but we note that, at the expense of slightly more complicated methods, external nodes could be *null*.

We can make the red-black tree definition more intuitive by noting an interesting correspondence between red-black trees and $(2,4)$ trees as illustrated in Figure 10.28. Namely, given a red-black tree, we can construct a corresponding $(2,4)$ tree by merging every red node v into its parent and storing the entry from v at its parent. Conversely, we can transform any $(2,4)$ tree into a corresponding red-black tree by coloring each node black and performing the following transformation for each internal node v:

- If v is a 2-node, then keep the (black) children of v as is

- If v is a 3-node, then create a new red node w, give v's first two (black) children to w, and make w and v's third child be the two children of v

- If v is a 4-node, then create two new red nodes w and z, give v's first two (black) children to w, give v's last two (black) children to z, and make w and z be the two children of v
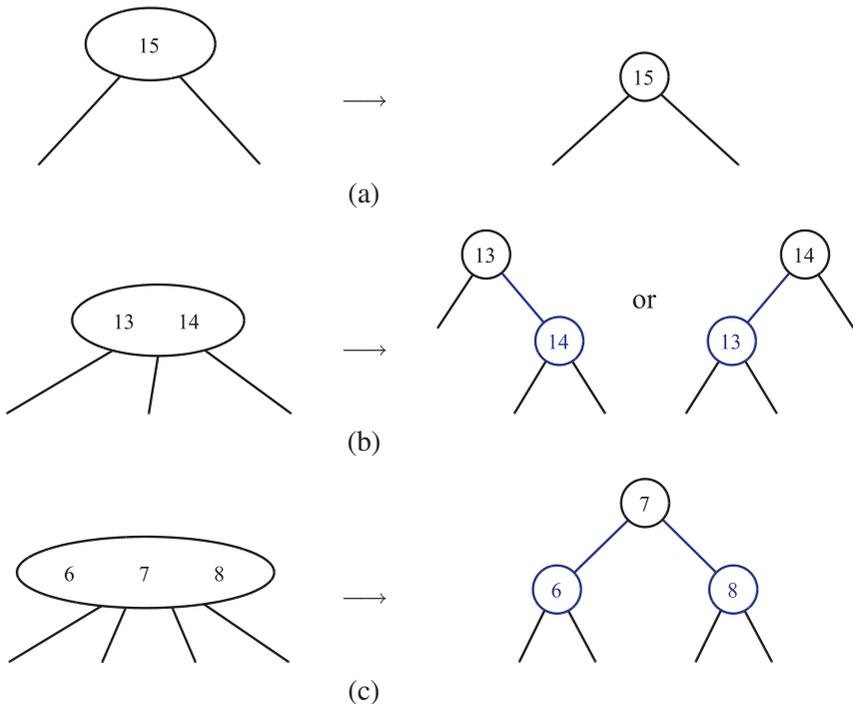
Figure 10.28: Correspondence between a $(2,4)$ tree and a red-black tree: (a) 2-node; (b) 3-node; (c) 4-node.

The correspondence between $(2,4)$ trees and red-black trees provides important intuition that we use in our discussion of how to perform updates in red-black trees. In fact, the update algorithms for red-black trees are mysteriously complex without this intuition.

Proposition 10.9: *The height of a red-black tree storing n entries is $O(\log n)$.*

Justification: Let T be a red-black tree storing n entries, and let h be the height of T. We justify this proposition by establishing the following fact

$$\log(n+1) \le h \le 2\log(n+1).$$

Let d be the common black depth of all the external nodes of T. Let T' be the $(2,4)$ tree associated with T, and let h' be the height of T'. Because of the correspondence between red-black trees and $(2,4)$ trees, we know that $h' = d$. Hence, by Proposition 10.8, $d = h' \le \log(n+1)$. By the internal node property, $h \le 2d$. Thus, we obtain $h \le 2\log(n+1)$. The other inequality, $\log(n+1) \le h$, follows from Proposition 7.10 and the fact that T has n internal nodes. ∎

We assume that a red-black tree is realized with a linked structure for binary trees (Section 7.3.4), in which we store a map entry and a color indicator at each node. Thus, the space requirement for storing n keys is $O(n)$. The algorithm for searching in a red-black tree T is the same as that for a standard binary search tree (Section 10.1). Thus, searching in a red-black tree takes $O(\log n)$ time.

10.5.1 Update Operations

Performing the update operations in a red-black tree is similar to that of a binary search tree, except that we must additionally restore the color properties.

Insertion

Now consider the insertion of an entry with key k into a red-black tree T, keeping in mind the correspondence between T and its associated $(2,4)$ tree T' and the insertion algorithm for T'. The algorithm initially proceeds as in a binary search tree (Section 10.1.2). Namely, we search for k in T until we reach an external node of T, and we replace this node with an internal node z, storing (k,x) and having two external-node children. If z is the root of T, we color z black, else we color z red. We also color the children of z black. This action corresponds to inserting (k,x) into a node of the $(2,4)$ tree T' with external children. In addition, this action preserves the root, external, and depth properties of T, but it may violate the internal property. Indeed, if z is not the root of T and the parent v of z is red, then we have a parent and a child (namely, v and z) that are both red. Note that by the root property, v cannot be the root of T, and by the internal property (which was previously satisfied), the parent u of v must be black. Since z and its parent are red, but z's grandparent u is black, we call this violation of the internal property a ***double red*** at node z.

To remedy a double red, we consider two cases.

Case 1: *The Sibling* **w** *of* **v** *is Black.* (See Figure 10.29.) In this case, the double
red denotes the fact that we have created in our red-black tree T a malformed
replacement for a corresponding 4-node of the $(2,4)$ tree T', which has as its
children the four black children of u, v, and z. Our malformed replacement
has one red node (v) that is the parent of another red node (z), while we want
it to have the two red nodes as siblings instead. To fix this problem, we
perform a ***trinode restructuring*** of T. The trinode restructuring is done by
the operation restructure(z), which consists of the following steps (see again
Figure 10.29; this operation is also discussed in Section 10.2):

- Take node z, its parent v, and grandparent u, and temporarily relabel
  them as a, b, and c, in left-to-right order, so that a, b, and c will be
  visited in this order by an inorder tree traversal.
- Replace the grandparent u with the node labeled b, and make nodes a
  and c the children of b, keeping inorder relationships unchanged.

After performing the restructure(z) operation, we color b black and we color
a and c red. Thus, the restructuring eliminates the double red problem.

(a)

(b)
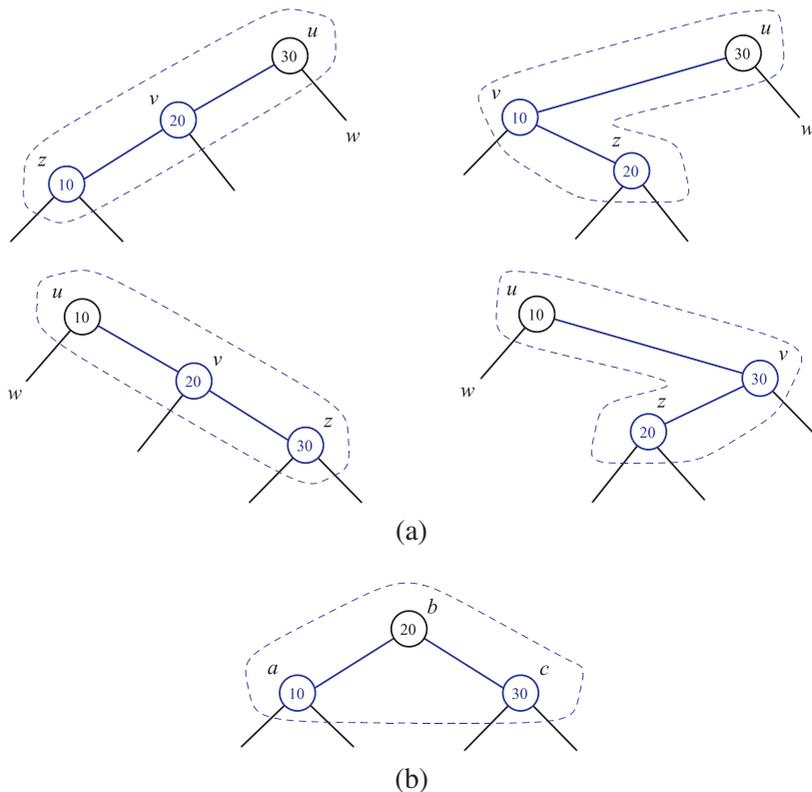
Figure 10.29: Restructuring a red-black tree to remedy a double red: (a) the four
configurations for u, v, and z before restructuring; (b) after restructuring.

Case 2: *The Sibling* **w** *of* **v** *is Red.* (See Figure 10.30.) In this case, the double red denotes an overflow in the corresponding $(2,4)$ tree T. To fix the problem, we perform the equivalent of a split operation. Namely, we do a ***recoloring***: we color v and w black and their parent u red (unless u is the root, in which case, it is colored black). It is possible that, after such a recoloring, the double red problem reappears, although higher up in the tree T, since u may have a red parent. If the double red problem reappears at u, then we repeat the consideration of the two cases at u. Thus, a recoloring either eliminates the double red problem at node z, or propagates it to the grandparent u of z. We continue going up T performing recolorings until we finally resolve the double red problem (with either a final recoloring or a trinode restructuring). Thus, the number of recolorings caused by an insertion is no more than half the height of tree T, that is, no more than $\log(n+1)$ by Proposition 10.9.

(a)

(b)
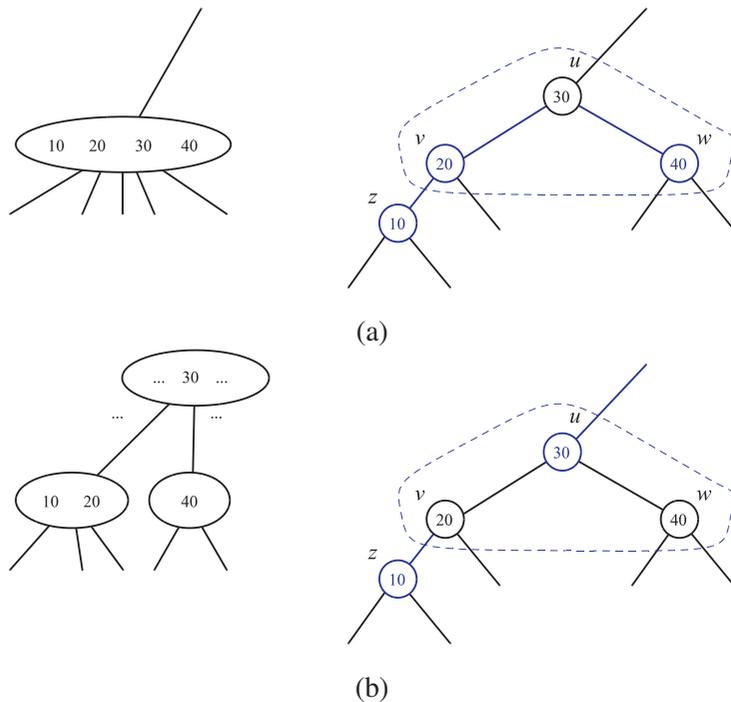
Figure 10.30: Recoloring to remedy the double red problem: (a) before recoloring and the corresponding 5-node in the associated $(2,4)$ tree before the split; (b) after the recoloring (and corresponding nodes in the associated $(2,4)$ tree after the split).

Figures 10.31 and 10.32 show a sequence of insertion operations in a red-black tree.
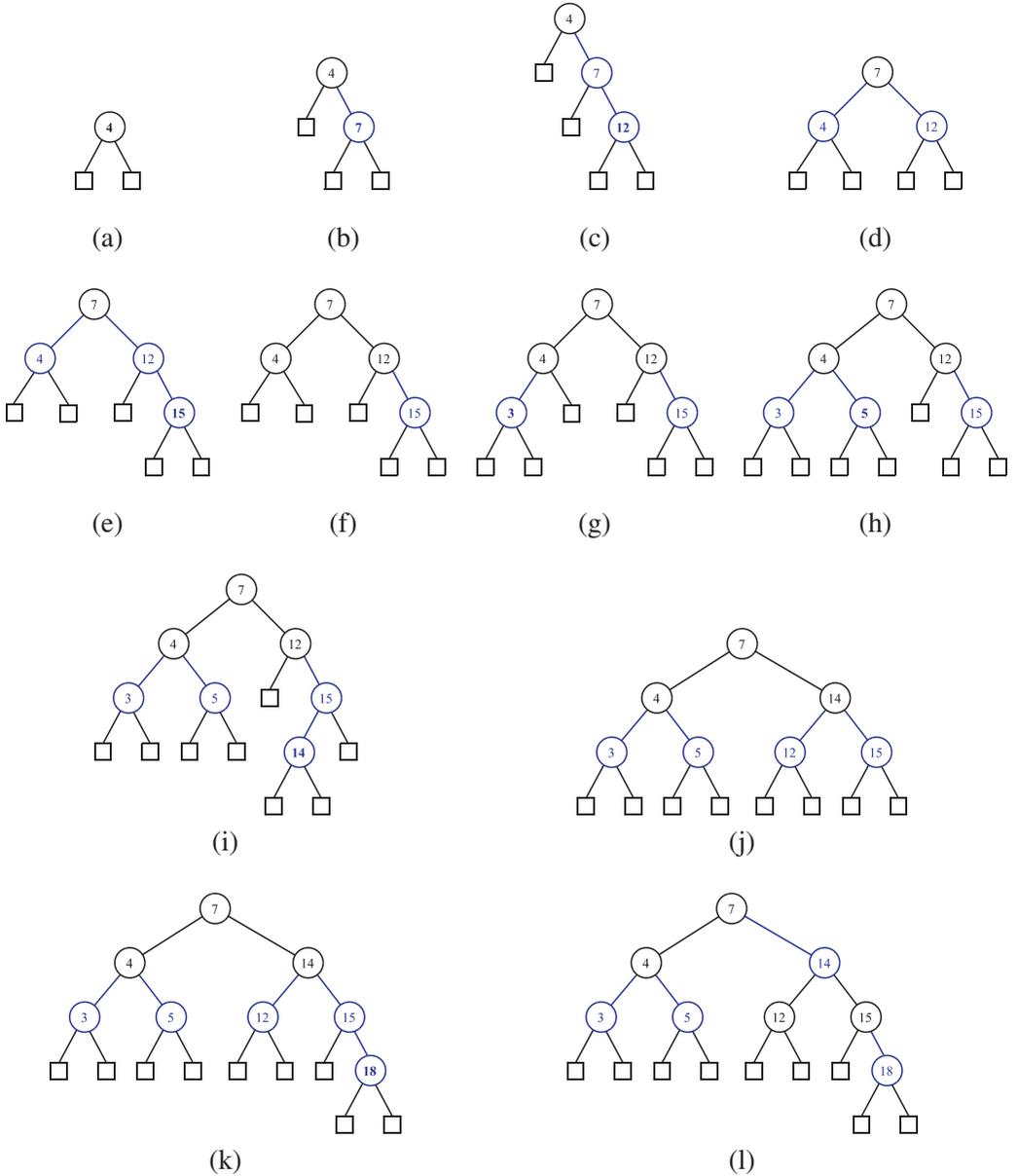
Figure 10.31: A sequence of insertions in a red-black tree: (a) initial tree; (b) insertion of 7; (c) insertion of 12, which causes a double red; (d) after restructuring; (e) insertion of 15, which causes a double red; (f) after recoloring (the root remains black); (g) insertion of 3; (h) insertion of 5; (i) insertion of 14, which causes a double red; (j) after restructuring; (k) insertion of 18, which causes a double red; (l) after recoloring. (Continues in Figure 10.32.)
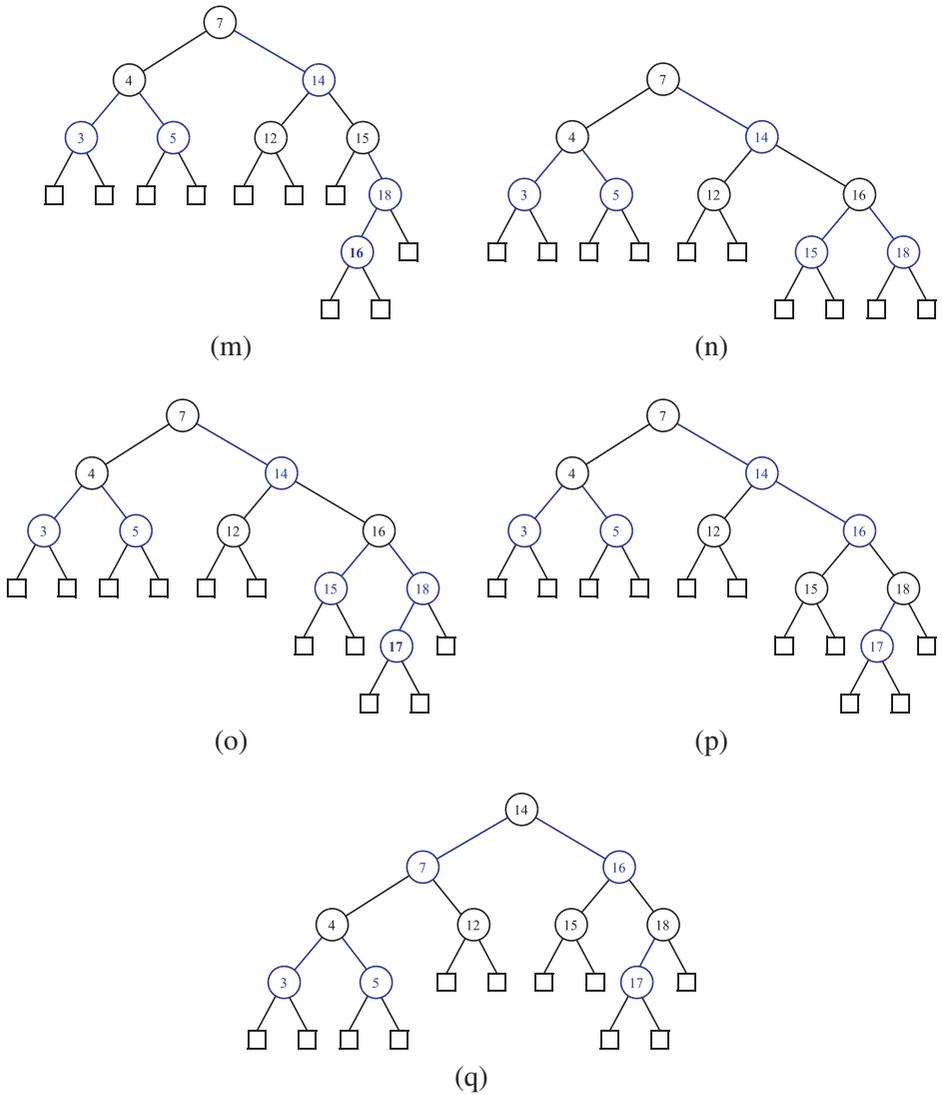
Figure 10.32: A sequence of insertions in a red-black tree: (m) insertion of 16, which causes a double red; (n) after restructuring; (o) insertion of 17, which causes a double red; (p) after recoloring there is again a double red, to be handled by a restructuring; (q) after restructuring. (Continued from Figure 10.31.)

The cases for insertion imply an interesting property for red-black trees. Namely, since the Case 1 action eliminates the double-red problem with a single trinode restructuring and the Case 2 action performs no restructuring operations, at most one restructuring is needed in a red-black tree insertion. By the above analysis and the fact that a restructuring or recoloring takes $O(1)$ time, we have the following.

Proposition 10.10: *The insertion of a key-value entry in a red-black tree storing n entries can be done in $O(\log n)$ time and requires $O(\log n)$ recolorings and one trinode restructuring (a restructure operation).*

Removal

Suppose now that we are asked to remove an entry with key k from a red-black tree T. Removing such an entry initially proceeds like a binary search tree (Section 10.1.2). First, we search for a node u storing such an entry. If node u does not have an external child, we find the internal node v following u in the inorder traversal of T, move the entry at v to u, and perform the removal at v. Thus, we may consider only the removal of an entry with key k stored at a node v with an external child w. Also, as we did for insertions, we keep in mind the correspondence between red-black tree T and its associated $(2,4)$ tree T' (and the removal algorithm for T').

To remove the entry with key k from a node v of T with an external child w we proceed as follows. Let r be the sibling of w and x be the parent of v. We remove nodes v and w, and make r a child of x. If v was red (hence r is black) or r is red (hence v was black), we color r black and we are done. If, instead, r is black and v was black, then, to preserve the depth property, we give r a fictitious **double black** color. We now have a color violation, called the double black problem. A double black in T denotes an underflow in the corresponding $(2,4)$ tree T'. Recall that x is the parent of the double black node r. To remedy the double-black problem at r, we consider three cases.

Case 1: *The Sibling* **y** *of* **r** *is Black and Has a Red Child* **z**. (See Figure 10.33.)
Resolving this case corresponds to a transfer operation in the $(2,4)$ tree T'. We perform a **trinode restructuring** by means of operation restructure(z). Recall that the operation restructure(z) takes the node z, its parent y, and grandparent x, labels them temporarily left to right as a, b, and c, and replaces x with the node labeled b, making it the parent of the other two. (See the description of restructure in Section 10.2.) We color a and c black, give b the former color of x, and color r black. This trinode restructuring eliminates the double black problem. Hence, at most one restructuring is performed in a removal operation in this case.

Figure 10.33: Restructuring of a red-black tree to remedy the double black problem: (a) and (b) configurations before the restructuring, where r is a right child and the associated nodes in the corresponding $(2,4)$ tree before the transfer (two other symmetric configurations where r is a left child are possible); (c) configuration after the restructuring and the associated nodes in the corresponding $(2,4)$ tree after the transfer. The grey color for node x in parts (a) and (b) and for node b in part (c) denotes the fact that this node may be colored either red or black.

Case 2: *The Sibling **y** of **r** is Black and Both Children of **y** Are Black.* (See Figures 10.34 and 10.35.) Resolving this case corresponds to a fusion operation in the corresponding $(2,4)$ tree T'. We do a ***recoloring***; we color r black, we color y red, and, if x is red, we color it black (Figure 10.34); otherwise, we color x ***double black*** (Figure 10.35). Hence, after this recoloring, the double black problem may reappear at the parent x of r. (See Figure 10.35.) That is, this recoloring either eliminates the double black problem or propagates it into the parent of the current node. We then repeat a consideration of these three cases at the parent. Thus, since Case 1 performs a trinode restructuring operation and stops (and, as we will soon see, Case 3 is similar), the number of recolorings caused by a removal is no more than $\log(n+1)$.

(a)

(b)
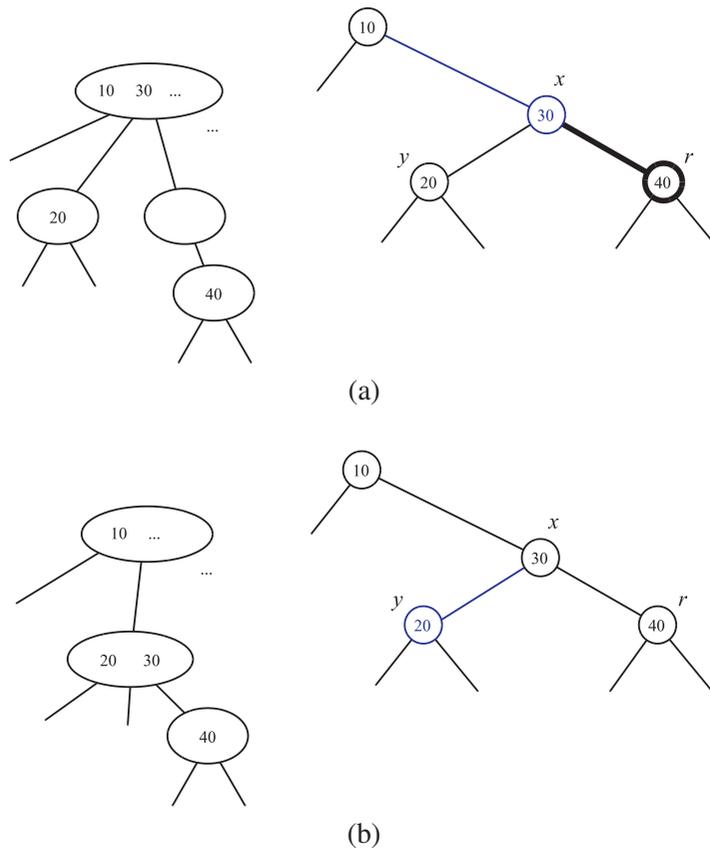
Figure 10.34: Recoloring of a red-black tree that fixes the double black problem: (a) before the recoloring and corresponding nodes in the associated $(2,4)$ tree before the fusion (other similar configurations are possible); (b) after the recoloring and corresponding nodes in the associated $(2,4)$ tree after the fusion.
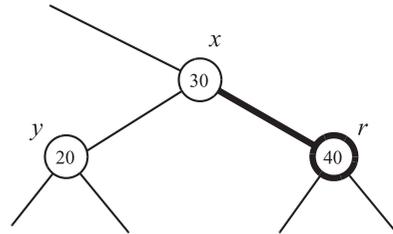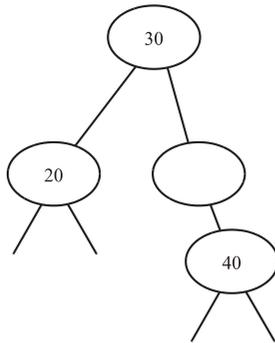
(a)
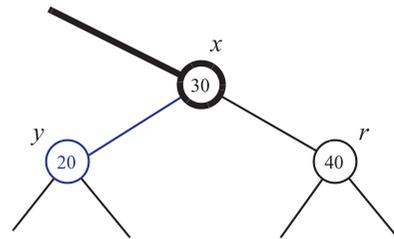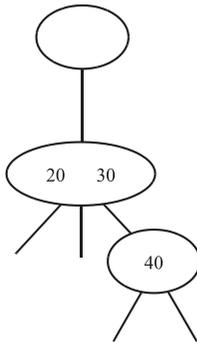
(b)

Figure 10.35: Recoloring of a red-black tree that propagates the double black problem: (a) configuration before the recoloring and corresponding nodes in the associated $(2,4)$ tree before the fusion (other similar configurations are possible); (b) configuration after the recoloring and corresponding nodes in the associated $(2,4)$ tree after the fusion.

Case 3: *The Sibling* **y** *of* **r** *Is Red.* (See Figure 10.36.) In this case, we perform
an *adjustment* operation, as follows. If y is the right child of x, let z be the
right child of y; otherwise, let z be the left child of y. Execute the trinode
restructuring operation restructure(z), which makes y the parent of x. Color
y black and x red. An adjustment corresponds to choosing a different rep-
resentation of a 3-node in the $(2,4)$ tree T'. After the adjustment operation,
the sibling of r is black, and either Case 1 or Case 2 applies, with a different
meaning of x and y. Note that if Case 2 applies, the double-black problem
cannot reappear. Thus, to complete Case 3 we make one more application
of either Case 1 or Case 2 above and we are done. Therefore, at most one
adjustment is performed in a removal operation.

(a)

(b)
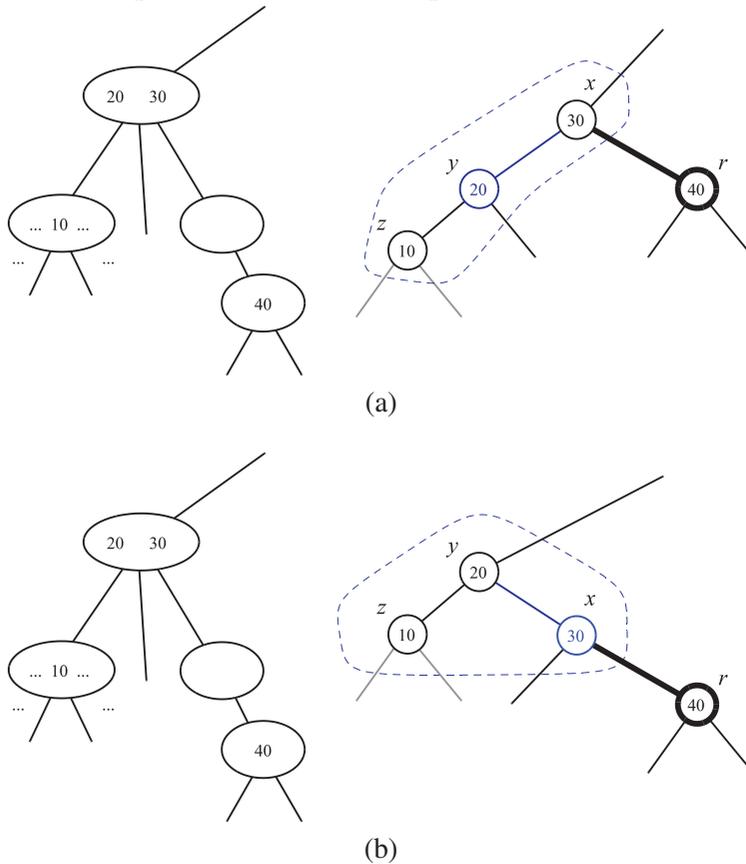
Figure 10.36: Adjustment of a red-black tree in the presence of a double black
problem: (a) configuration before the adjustment and corresponding nodes in the
associated $(2,4)$ tree (a symmetric configuration is possible); (b) configuration after
the adjustment with the same corresponding nodes in the associated $(2,4)$ tree.

From the above algorithm description, we see that the tree updating needed after a removal involves an upward march in the tree T, while performing at most a constant amount of work (in a restructuring, recoloring, or adjustment) per node. Thus, since any changes we make at a node in T during this upward march takes $O(1)$ time (because it affects a constant number of nodes), we have the following.

Proposition 10.11: *The algorithm for removing an entry from a red-black tree with n entries takes $O(\log n)$ time and performs $O(\log n)$ recolorings and at most one adjustment plus one additional trinode restructuring. Thus, it performs at most two restructure operations.*

In Figures 10.37 and 10.38, we show a sequence of removal operations on a red-black tree. We illustrate Case 1 restructurings in Figure 10.37(c) and (d). We illustrate Case 2 recolorings at several places in Figures 10.37 and 10.38. Finally, in Figure 10.38(i) and (j), we show an example of a Case 3 adjustment.
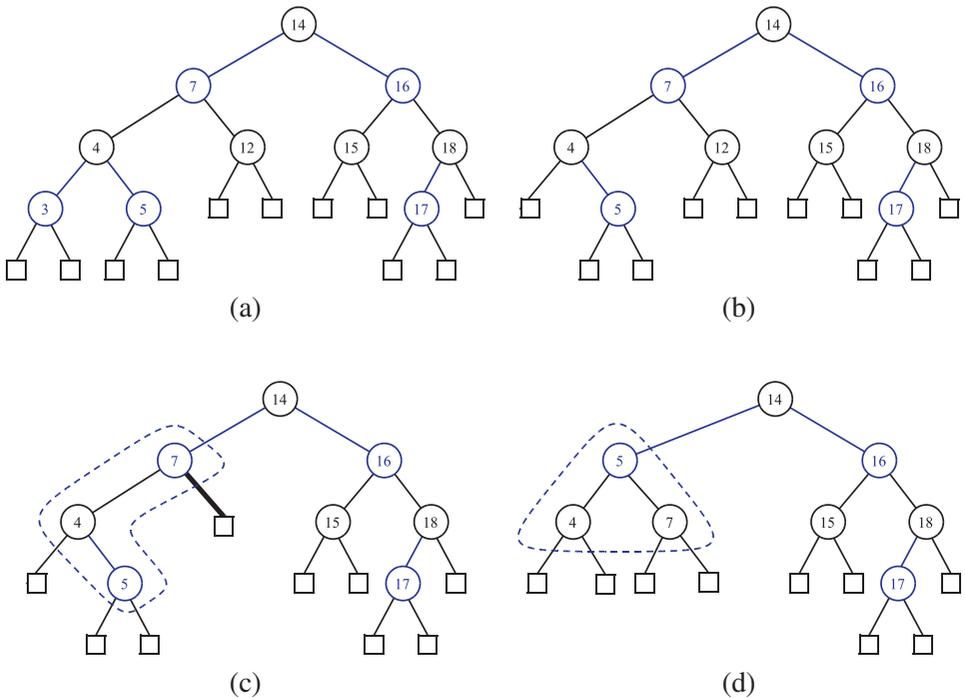
Figure 10.37: Sequence of removals from a red-black tree: (a) initial tree; (b) removal of 3; (c) removal of 12, causing a double black (handled by restructuring); (d) after restructuring. (Continues in Figure 10.38.)
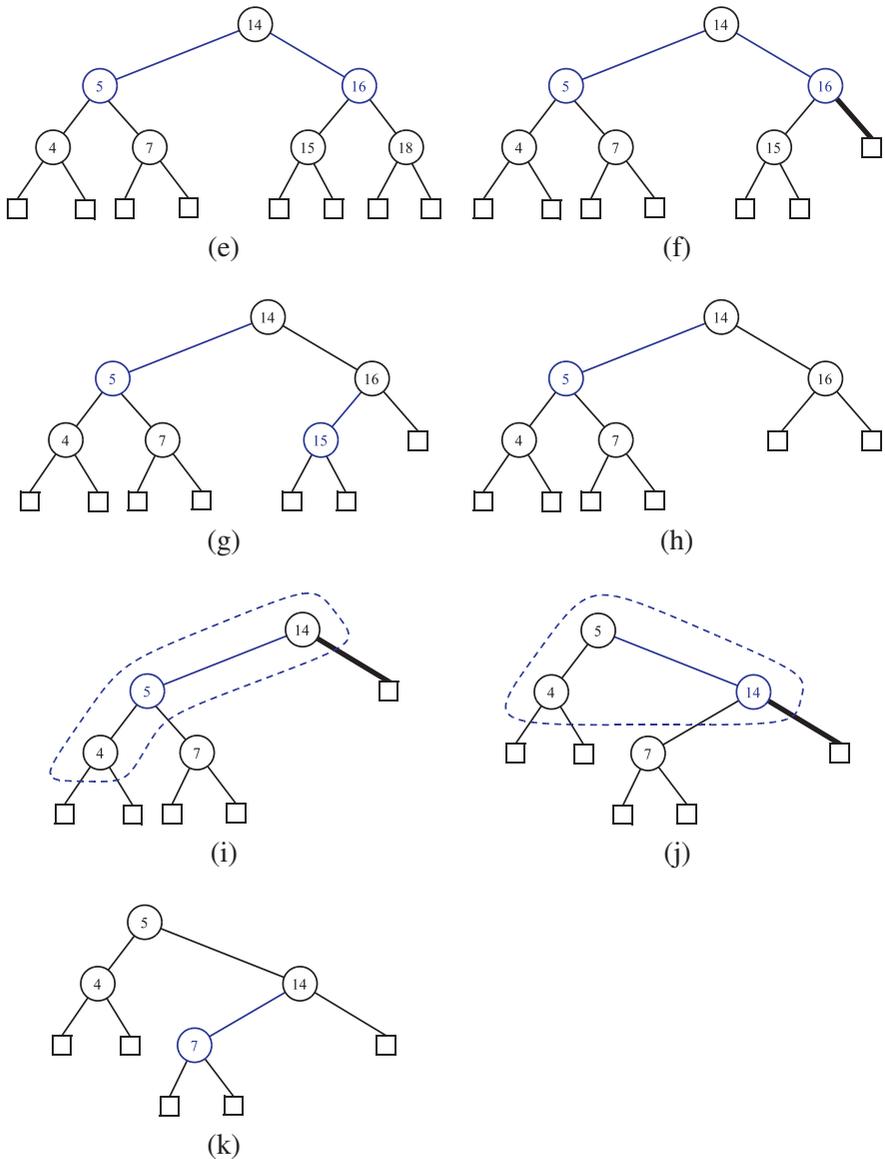
Figure 10.38: Sequence of removals in a red-black tree : (e) removal of 17; (f) removal of 18, causing a double black (handled by recoloring); (g) after recoloring; (h) removal of 15; (i) removal of 16, causing a double black (handled by an adjustment); (j) after the adjustment the double black needs to be handled by a recoloring; (k) after the recoloring. (Continued from Figure 10.37.)

Performance of Red-Black Trees

Table 10.4 summarizes the running times of the main operations of a map realized by means of a red-black tree. We illustrate the justification for these bounds in Figure 10.39.

Operation	*Time*
size, empty	$O(1)$
find, insert, erase	$O(\log n)$

Table 10.4: Performance of an n-entry map realized by a red-black tree. The space usage is $O(n)$.
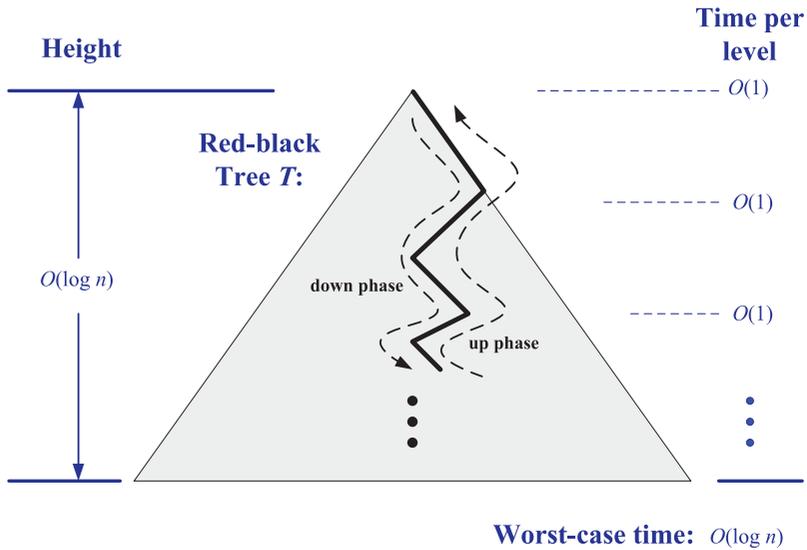
Figure 10.39: The running time of searches and updates in a red-black tree. The time performance is $O(1)$ per level, broken into a down phase, which typically involves searching, and an up phase, which typically involves recolorings and performing local trinode restructurings (rotations).

Thus, a red-black tree achieves logarithmic worst-case running times for both searching and updating in a map. The red-black tree data structure is slightly more complicated than its corresponding $(2,4)$ tree. Even so, a red-black tree has a conceptual advantage that only a constant number of trinode restructurings are ever needed to restore the balance in a red-black tree after an update.

10.5.2   C++ Implementation of a Red-Black Tree

In this section, we discuss a C++ implementation of the dictionary ADT by means of a red-black tree. It is interesting to note that the C++ Standard Template Library uses a red-black tree in its implementation of its classes map and multimap. The difference between the two is similar to the difference between our map and dictionary ADTs. The STL map class does not allow entries with duplicate keys, whereas the STL multimap does. There is a significant difference, however, in the behavior of the map's insert(k,x) function and our map's put(k,x) function. If the key k is not present, both functions insert the new entry (k,x) in the map. If the key is already present, the STL map simply ignores the request, and the current entry is unchanged. In contrast, our put function replaces the existing value with the new value x. The implementation presented in this section allows for multiple keys.

We present the major portions of the implementation in this section. To keep the presentation concise, we have omitted the implementations of a number of simpler utility functions.

We begin by presenting the enhanced entry class, called RBEntry. It is derived from the entry class of Code Fragment 10.3. It inherits the key and value members, and it defines a member variable *col*, which stores the color of the node. The color is either RED or BLACK. It provides member functions for accessing and setting this value. These functions have been protected, so a user cannot access them, but RBTree can.

```
enum Color {RED, BLACK};                          // node colors

template <typename E>
class RBEntry : public E {                         // a red-black entry
private:
  Color col;                                       // node color
protected:                                         // local types
  typedef typename E::Key K;                       // key type
  typedef typename E::Value V;                     // value type
  Color color() const { return col; }              // get color
  bool isRed() const { return col == RED; }
  bool isBlack() const { return col == BLACK; }
  void setColor(Color c) { col = c; }
public:                                            // public functions
  RBEntry(const K& k = K(), const V& v = V())      // constructor
    : E(k,v), col(BLACK) { }
  friend class RBTree<E>;                           // allow RBTree access
};
```

Code Fragment 10.19: A key-value entry for class RBTree, containing the associated node's color.

In Code Fragment 10.20, we present the class definition for RBTree. The declaration is almost entirely analogous to that of AVLTree, except that the utility functions used to maintain the structure are different. We have chosen to present only the two most interesting utility functions, remedyDoubleRed and remedyDoubleBlack. The meanings of most of the omitted utilities are easy to infer. (For example hasTwoExternalChildren(v) determines whether a node v has two external children.)

```
template <typename E>                          // a red-black tree
class RBTree : public SearchTree< RBEntry<E> > {
public:                                        // public types
  typedef RBEntry<E> RBEntry;                  // an entry
  typedef typename SearchTree<RBEntry>::Iterator Iterator; // an iterator
protected:                                     // local types
  typedef typename RBEntry::Key K;             // a key
  typedef typename RBEntry::Value V;           // a value
  typedef SearchTree<RBEntry> ST;              // a search tree
  typedef typename ST::TPos TPos;              // a tree position
public:                                        // public functions
  RBTree();                                    // constructor
  Iterator insert(const K& k, const V& x);     // insert (k,x)
  void erase(const K& k) throw(NonexistentElement); // remove key k entry
  void erase(const Iterator& p);               // remove entry at p
protected:                                     // utility functions
  void remedyDoubleRed(const TPos& z);         // fix double-red z
  void remedyDoubleBlack(const TPos& r);       // fix double-black r
  // ...(other utilities omitted)
};
```

Code Fragment 10.20: Class RBTree, which implements a dictionary ADT using a red-black tree.

We first discuss the implementation of the function insert(k,x), which is given in Code Fragment 10.21. We invoke the inserter utility function of SearchTree, which returns the position of the inserted node. If this node is the root of the search tree, we set its color to black. Otherwise, we set its color to red and check whether restructuring is needed by invoking remedyDoubleRed.

This latter utility performs the necessary checks and restructuring presented in the discussion of insertion in Section 10.5.1. Let z denote the location of the newly inserted node. If both z and its parent are red, we need to remedy the situation. To do so, we consider two cases. Let v denote z's parent and let w be v's sibling. If w is black, we fall under Case 1 of the insertion update procedure. We apply restructuring at z. The top vertex of the resulting subtree, denoted by v, is set to black, and its two children are set to red.

On the other hand, if w is red, then we fall under Case 2 of the update procedure.

We resolve the situation by coloring both *v* and its sibling *w* black. If their common parent is not the root, we set its color to red. This may induce another double-red problem at *v*'s parent *u*, so we invoke the function recursively on *u*.

```
/* RBTree⟨E⟩ :: */                                     // insert (k,x)
  Iterator insert(const K& k, const V& x) {
    TPos v = inserter(k, x);                            // insert in base tree
    if (v == ST::root())
      setBlack(v);                                      // root is always black
    else {
      setRed(v);
      remedyDoubleRed(v);                               // rebalance if needed
    }
    return Iterator(v);
  }

/* RBTree⟨E⟩ :: */                                     // fix double-red z
  void remedyDoubleRed(const TPos& z) {
    TPos v = z.parent();                                // v is z's parent
    if (v == ST::root() || v−>isBlack()) return;        // v is black, all ok
                                                        // z, v are double-red
    if (sibling(v)−>isBlack())  {                       // Case 1: restructuring
      v = restructure(z);
      setBlack(v);                                      // top vertex now black
      setRed(v.left()); setRed(v.right());              // set children red
    }
    else  {                                             // Case 2: recoloring
      setBlack(v); setBlack(sibling(v));                // set v and sibling black
      TPos u = v.parent();                              // u is v's parent
      if (u == ST::root()) return;
      setRed(u);                                        // make u red
      remedyDoubleRed(u);                               // may need to fix u now
    }
  }
```

Code Fragment 10.21: The functions related to insertion for class RBTree. The function insert invokes the inserter utility function, which was given in Code Fragment 10.10.

Finally, in Code Fragment 10.22, we present the implementation of the removal function for the red-black tree. (We have omitted the simpler iterator-based erase function.) The removal follows the process discussed in Section 10.5.1. We first search for the key to be removed, and generate an exception if it is not found. Otherwise, we invoke the eraser utility of class SearchTree, which returns the position of the node *r* that replaced the deleted node. If either *r* or its former parent was red, we color *r* black and we are done. Otherwise, we face a potential double-black problem. We handle this by invoking the function remedyDoubleBlack.

```
/* RBTree⟨E⟩ :: */                                  // remove key k entry
  void erase(const K& k) throw(NonexistentElement) {
    TPos u = finder(k, ST::root());                 // find the node
    if (Iterator(u) == ST::end())
      throw NonexistentElement("Erase of nonexistent");
    TPos r = eraser(u);                             // remove u
    if (r == ST::root() || r−>isRed() || wasParentRed(r))
      setBlack(r);                                  // fix by color change
    else                                            // r, parent both black
      remedyDoubleBlack(r);                         // fix double-black r
  }

/* RBTree⟨E⟩ :: */                                  // fix double-black r
  void remedyDoubleBlack(const TPos& r) {
    TPos x = r.parent();                            // r's parent
    TPos y = sibling(r);                            // r's sibling
    if (y−>isBlack()) {
      if (y.left()−>isRed() || y.right()−>isRed()) {  // Case 1: restructuring
                                                    // z is y's red child
        TPos z = (y.left()−>isRed() ? y.left() : y.right());
        Color topColor = x−>color();                // save top vertex color
        z = restructure(z);                         // restructure x,y,z
        setColor(z, topColor);                      // give z saved color
        setBlack(r);                                // set r black
        setBlack(z.left()); setBlack(z.right());    // set z's children black
      }
      else {                                        // Case 2: recoloring
        setBlack(r); setRed(y);                     // r=black, y=red
        if (x−>isBlack() && !(x == ST::root()))
          remedyDoubleBlack(x);                     // fix double-black x
        setBlack(x);
      }
    }
    else {                                          // Case 3: adjustment
      TPos z = (y == x.right() ? y.right() : y.left());  // grandchild on y's side
      restructure(z);                               // restructure x,y,z
      setBlack(y); setRed(x);                       // y=black, x=red
      remedyDoubleBlack(r);                         // fix r by Case 1 or 2
    }
  }
```

Code Fragment 10.22: The functions related to removal for class RBTree. The function erase invokes the eraser utility function, which was given in Code Fragment 10.11.

10.6 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-10.1 If we insert the entries $(1,A)$, $(2,B)$, $(3,C)$, $(4,D)$, and $(5,E)$, in this order, into an initially empty binary search tree, what will it look like?

R-10.2 We defined a binary search tree so that keys equal to a node's key can be in either the left or right subtree of that node. Suppose we change the definition so that we restrict equal keys to the right subtree. What must a subtree of a binary search tree containing only equal keys look like in this case?

R-10.3 Insert, into an empty binary search tree, entries with keys 30, 40, 24, 58, 48, 26, 11, 13 (in this order). Draw the tree after each insertion.

R-10.4 How many different binary search trees can store the keys $\{1,2,3\}$?

R-10.5 Jack claims that the order in which a fixed set of entries is inserted into a binary search tree does not matter—the same tree results every time. Give a small example that proves he is wrong.

R-10.6 Rose claims that the order in which a fixed set of entries is inserted into an AVL tree does not matter—the same AVL tree results every time. Give a small example that proves she is wrong.

R-10.7 Are the rotations in Figures 10.9 and 10.11 single or double rotations?

R-10.8 Draw the AVL tree resulting from the insertion of an entry with key 52 into the AVL tree of Figure 10.11(b).

R-10.9 Draw the AVL tree resulting from the removal of the entry with key 62 from the AVL tree of Figure 10.11(b).

R-10.10 Explain why performing a rotation in an n-node binary tree represented using a vector takes $\Omega(n)$ time.

R-10.11 Is the search tree of Figure 10.1(a) a $(2,4)$ tree? Why or why not?

R-10.12 An alternative way of performing a split at a node v in a $(2,4)$ tree is to partition v into v' and v'', with v' being a 2-node and v'' a 3-node. Which of the keys k_1, k_2, k_3, or k_4 do we store at v's parent in this case? Why?

R-10.13 Cal claims that a $(2,4)$ tree storing a set of entries will always have the same structure, regardless of the order in which the entries are inserted. Show that he is wrong.

R-10.14 Draw four different red-black trees that correspond to the same $(2,4)$ tree.

R-10.15 Consider the set of keys $K = \{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\}$.

a. Draw a $(2,4)$ tree storing K as its keys using the fewest number of nodes.

b. Draw a $(2,4)$ tree storing K as its keys using the maximum number of nodes.

R-10.16 Consider the sequence of keys $(5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1)$. Draw the result of inserting entries with these keys (in the given order) into

a. An initially empty $(2,4)$ tree.

b. An initially empty red-black tree.

R-10.17 For the following statements about red-black trees, provide a justification for each true statement and a counterexample for each false one.

a. A subtree of a red-black tree is itself a red-black tree.

b. The sibling of an external node is either external or it is red.

c. There is a unique $(2,4)$ tree associated with a given red-black tree.

d. There is a unique red-black tree associated with a given $(2,4)$ tree.

R-10.18 Draw an example red-black tree that is not an AVL tree.

R-10.19 Consider a tree T storing 100,000 entries. What is the worst-case height of T in the following cases?

a. T is an AVL tree.

b. T is a $(2,4)$ tree.

c. T is a red-black tree.

d. T is a splay tree.

e. T is a binary search tree.

R-10.20 Perform the following sequence of operations in an initially empty splay tree and draw the tree after each set of operations.

a. Insert keys 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, in this order.

b. Search for keys 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, in this order.

c. Delete keys 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, in this order.

R-10.21 What does a splay tree look like if its entries are accessed in increasing order by their keys?

R-10.22 Explain how to use an AVL tree or a red-black tree to sort n comparable elements in $O(n\log n)$ time in the worst case.

R-10.23 Can we use a splay tree to sort n comparable elements in $O(n\log n)$ time in the *worst case*? Why or why not?

R-10.24 Explain why you would get the same output in an inorder listing of the entries in a binary search tree, T, independent of whether T is maintained to be an AVL tree, splay tree, or red-black tree.

Creativity

C-10.1 Describe a modification to the binary search tree data structure that would allow you to find the median entry, that is the entry with rank $\lfloor n/2 \rfloor$, in a binary search tree. Describe both the modification and the algorithm for finding the median assuming all keys are distinct.

C-10.2 Design a variation of algorithm TreeSearch for performing the operation findAll(k) in an ordered dictionary implemented with a binary search tree T, and show that it runs in time $O(h+s)$, where h is the height of T and s is the size of the collection returned.

C-10.3 Describe how to perform an operation eraseAll(k), which removes all the entries whose keys equal k in an ordered dictionary implemented with a binary search tree T, and show that this method runs in time $O(h+s)$, where h is the height of T and s is the size of the iterator returned.

C-10.4 Draw a schematic of an AVL tree such that a single erase operation could require $\Omega(\log n)$ trinode restructurings (or rotations) from a leaf to the root in order to restore the height-balance property.

C-10.5 Show how to perform an operation, eraseAll(k), which removes all entries with keys equal to K, in an ordered dictionary implemented with an AVL tree in time $O(s \log n)$, where n is the number of entries in the map and s is the size of the iterator returned.

C-10.6 Describe the changes that would need to be made to the binary search tree implementation given in the book to allow it to be used to support an ordered dictionary, where we allow for different entries with equal keys.

C-10.7 If we maintain a reference to the position of the left-most internal node of an AVL tree, then operation first (Section 9.3) can be performed in $O(1)$ time. Describe how the implementation of the other map functions needs to be modified to maintain a reference to the left-most position.

C-10.8 Show that any n-node binary tree can be converted to any other n-node binary tree using $O(n)$ rotations.

C-10.9 Let M be an ordered map with n entries implemented by means of an AVL tree. Show how to implement the following operation on M in time $O(\log n + s)$, where s is the size of the iterator returned.

     findAllInRange(k_1, k_2): Return an iterator of all the entries in M with key k such that $k_1 \le k \le k_2$.

C-10.10 Let M be an ordered map with n entries. Show how to modify the AVL tree to implement the following function for M in time $O(\log n)$.

     countAllInRange(k_1, k_2): Compute and return the number of entries in M with key k such that $k_1 \le k \le k_2$.

C-10.11 Draw a splay tree, T_1, together with the sequence of updates that produced it, and a red-black tree, T_2, on the same set of ten entries, such that a preorder traversal of T_1 would be the same as a preorder traversal of T_2.

C-10.12 Show that the nodes that become unbalanced in an AVL tree during an insert operation may be nonconsecutive on the path from the newly inserted node to the root.

C-10.13 Show that at most one node in an AVL tree becomes unbalanced after operation removeAboveExternal is performed within the execution of a erase map operation.

C-10.14 Show that at most one trinode restructuring operation is needed to restore balance after any insertion in an AVL tree.

C-10.15 Let T and U be $(2,4)$ trees storing n and m entries, respectively, such that all the entries in T have keys less than the keys of all the entries in U. Describe an $O(\log n + \log m)$ time method for *joining* T and U into a single tree that stores all the entries in T and U.

C-10.16 Repeat the previous problem for red-black trees T and U.

C-10.17 Justify Proposition 10.7.

C-10.18 The Boolean indicator used to mark nodes in a red-black tree as being "red" or "black" is not strictly needed when we have distinct keys. Describe a scheme for implementing a red-black tree without adding any extra space to standard binary search tree nodes.

C-10.19 Let T be a red-black tree storing n entries, and let k be the key of an entry in T. Show how to construct from T, in $O(\log n)$ time, two red-black trees T' and T'', such that T' contains all the keys of T less than k, and T'' contains all the keys of T greater than k. This operation destroys T.

C-10.20 Show that the nodes of any AVL tree T can be colored "red" and "black" so that T becomes a red-black tree.

C-10.21 The *mergeable heap* ADT consists of operations insert(k,x), removeMin(), unionWith(h), and min(), where the unionWith(h) operation performs a union of the mergeable heap h with the present one, destroying the old versions of both. Describe a concrete implementation of the mergeable heap ADT that achieves $O(\log n)$ performance for all its operations.

C-10.22 Consider a variation of splay trees, called *half-splay trees*, where splaying a node at depth d stops as soon as the node reaches depth $\lfloor d/2 \rfloor$. Perform an amortized analysis of half-splay trees.

C-10.23 The standard splaying step requires two passes, one downward pass to find the node x to splay, followed by an upward pass to splay the node x. Describe a method for splaying and searching for x in one downward pass. Each substep now requires that you consider the next two nodes in the path down to x, with a possible zig substep performed at the end. Describe how to perform the zig-zig, zig-zag, and zig steps.

C-10.24 Describe a sequence of accesses to an n-node splay tree T, where n is odd, that results in T consisting of a single chain of internal nodes with external node children, such that the internal-node path down T alternates between left children and right children.

C-10.25 Explain how to implement a vector of n elements so that the functions insert and at take $O(\log n)$ time in the worst case.

Projects

P-10.1 Write a program that performs a simple n-body simulation, called "Jumping Leprechauns." This simulation involves n leprechauns, numbered 1 to n. It maintains a gold value g_i for each leprechaun i, which begins with each leprechaun starting out with a million dollars worth of gold, that is, $g_i = 1\,000\,000$ for each $i = 1, 2, \ldots, n$. In addition, the simulation also maintains, for each leprechaun, i, a place on the horizon, which is represented as a double-precision floating point number, x_i. In each iteration of the simulation, the simulation processes the leprechauns in order. Processing a leprechaun i during this iteration begins by computing a new place on the horizon for i, which is determined by the assignment

$$x_i \leftarrow x_i + rg_i,$$

where r is a random floating-point number between -1 and 1. The leprechaun i then steals half the gold from the nearest leprechauns on either side of him and adds this gold to his gold value, g_i. Write a program that can perform a series of iterations in this simulation for a given number, n, of leprechauns. You must maintain the set of horizon positions using an ordered map data structure described in this chapter.

P-10.2 Extend class BinarySearchTree (Section 10.1.3) to support the functions of the ordered map ADT (see Section 9.3).

P-10.3 Implement a class RestructurableNodeBinaryTree that supports the functions of the binary tree ADT, plus a function restructure for performing a rotation operation. This class is a component of the implementation of an AVL tree given in Section 10.2.2.

P-10.4 Write a C++ class that implements all the functions of the ordered map ADT (see Section 9.3) using an AVL tree.

P-10.5 Write a C++ class that implements all the functions of the ordered map ADT (see Section 9.3) using a $(2,4)$ tree.

P-10.6 Write a C++ class that implements all the functions of the ordered map ADT (see Section 9.3) using a red-black tree.

P-10.7 Form a three-programmer team and have each member implement a map using a different search tree data structure. Perform a cooperative experimental study to compare the speed of these three implementations.

P-10.8 Write a C++ class that can take any red-black tree and convert it into its corresponding $(2,4)$ tree and can take any $(2,4)$ tree and convert it into its corresponding red-black tree.

P-10.9 Implement the map ADT using a splay tree, and compare its performance experimentally with the STL map class, which uses a red-black tree.

P-10.10 Prepare an implementation of splay trees that uses bottom-up splaying as described in this chapter and another that uses top-down splaying as described in Exercise C-10.23. Perform extensive experimental studies to see which implementation is better in practice, if any.

P-10.11 Implement a binary search tree data structure so that it can support the dictionary ADT, where different entries can have equal keys. In addition, implement the functions entrySetPreorder(), entrySetInorder(), and entrySetPostorder(), which produce an iterable collection of the entries in the binary search tree in the same order they would respectively be visited in a preorder, inorder, and postorder traversal of the tree.

Chapter Notes

Some of the data structures discussed in this chapter are extensively covered by Knuth in his *Sorting and Searching* book [60], and by Mehlhorn in [73]. AVL trees are due to Adel'son-Vel'skii and Landis [1], who invented this class of balanced search trees in 1962. Binary search trees, AVL trees, and hashing are described in Knuth's *Sorting and Searching* [60] book. Average-height analyses for binary search trees can be found in the books by Aho, Hopcroft, and Ullman [5] and Cormen, Leiserson, Rivest and Stein [25]. The handbook by Gonnet and Baeza-Yates [37] contains a number of theoretical and experimental comparisons among map implementations. Aho, Hopcroft, and Ullman [4] discuss $(2,3)$ trees, which are similar to $(2,4)$ trees. [9]. Variations and interesting properties of red-black trees are presented in a paper by Guibas and Sedgewick [42]. The reader interested in learning more about different balanced tree data structures is referred to the books by Mehlhorn [73] and Tarjan [95], and the book chapter by Mehlhorn and Tsakalidis [75]. Knuth [60] is excellent additional reading that includes early approaches to balancing trees. Splay trees were invented by Sleator and Tarjan [89] (see also [95]).

This page intentionally left blank

Chapter

11

Sorting, Sets, and Selection

Contents

11.1 Merge-Sort

In this section, we present a sorting technique, called ***merge-sort***, which can be described in a simple and compact way using recursion.

11.1.1 Divide-and-Conquer

Merge-sort is based on an algorithmic design pattern called ***divide-and-conquer***. The divide-and-conquer pattern consists of the following three steps:

1. ***Divide:*** If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution obtained. Otherwise, divide the input data into two or more disjoint subsets.
2. ***Recur:*** Recursively solve the subproblems associated with the subsets.
3. ***Conquer:*** Take the solutions to the subproblems and "merge" them into a solution to the original problem.

Using Divide-and-Conquer for Sorting

Recall that in a sorting problem we are given a sequence of n objects, stored in a linked list or an array, together with some comparator defining a total order on these objects, and we are asked to produce an ordered representation of these objects. To allow for sorting of either representation, we describe our sorting algorithm at a high level for sequences and explain the details needed to implement it for linked lists and arrays. To sort a sequence S with n elements using the three divide-and-conquer steps, the merge-sort algorithm proceeds as follows:

1. ***Divide:*** If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S; that is, S_1 contains the first $\lceil n/2 \rceil$ elements of S, and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements.
2. ***Recur:*** Recursively sort sequences S_1 and S_2.
3. ***Conquer:*** Put back the elements into S by merging the sorted sequences S_1 and S_2 into a sorted sequence.

In reference to the divide step, we recall that the notation $\lceil x \rceil$ indicates the ***ceiling*** of x, that is, the smallest integer m, such that $x \leq m$. Similarly, the notation $\lfloor x \rfloor$ indicates the ***floor*** of x, that is, the largest integer k, such that $k \leq x$.

We can visualize an execution of the merge-sort algorithm by means of a binary tree T, called the ***merge-sort tree***. Each node of T represents a recursive invocation (or call) of the merge-sort algorithm. We associate the sequence S that is processed by the invocation associated with v, with each node v of T. The children of node v are associated with the recursive calls that process the subsequences S_1 and S_2 of S. The external nodes of T are associated with individual elements of S, corresponding to instances of the algorithm that make no recursive calls.

Figure 11.1 summarizes an execution of the merge-sort algorithm by showing the input and output sequences processed at each node of the merge-sort tree. The step-by-step evolution of the merge-sort tree is shown in Figures 11.2 through 11.4.

This algorithm visualization in terms of the merge-sort tree helps us analyze the running time of the merge-sort algorithm. In particular, since the size of the input sequence roughly halves at each recursive call of merge-sort, the height of the merge-sort tree is about $\log n$ (recall that the base of log is 2 if omitted).
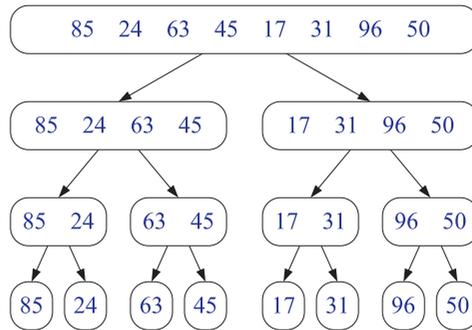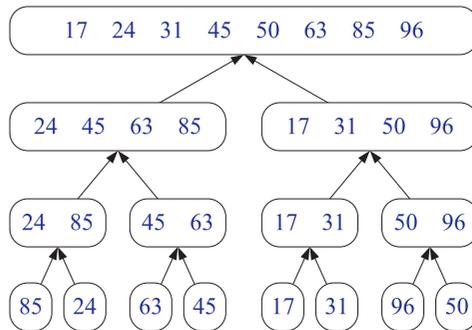
Figure 11.1: Merge-sort tree T for an execution of the merge-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of T; (b) output sequences generated at each node of T.
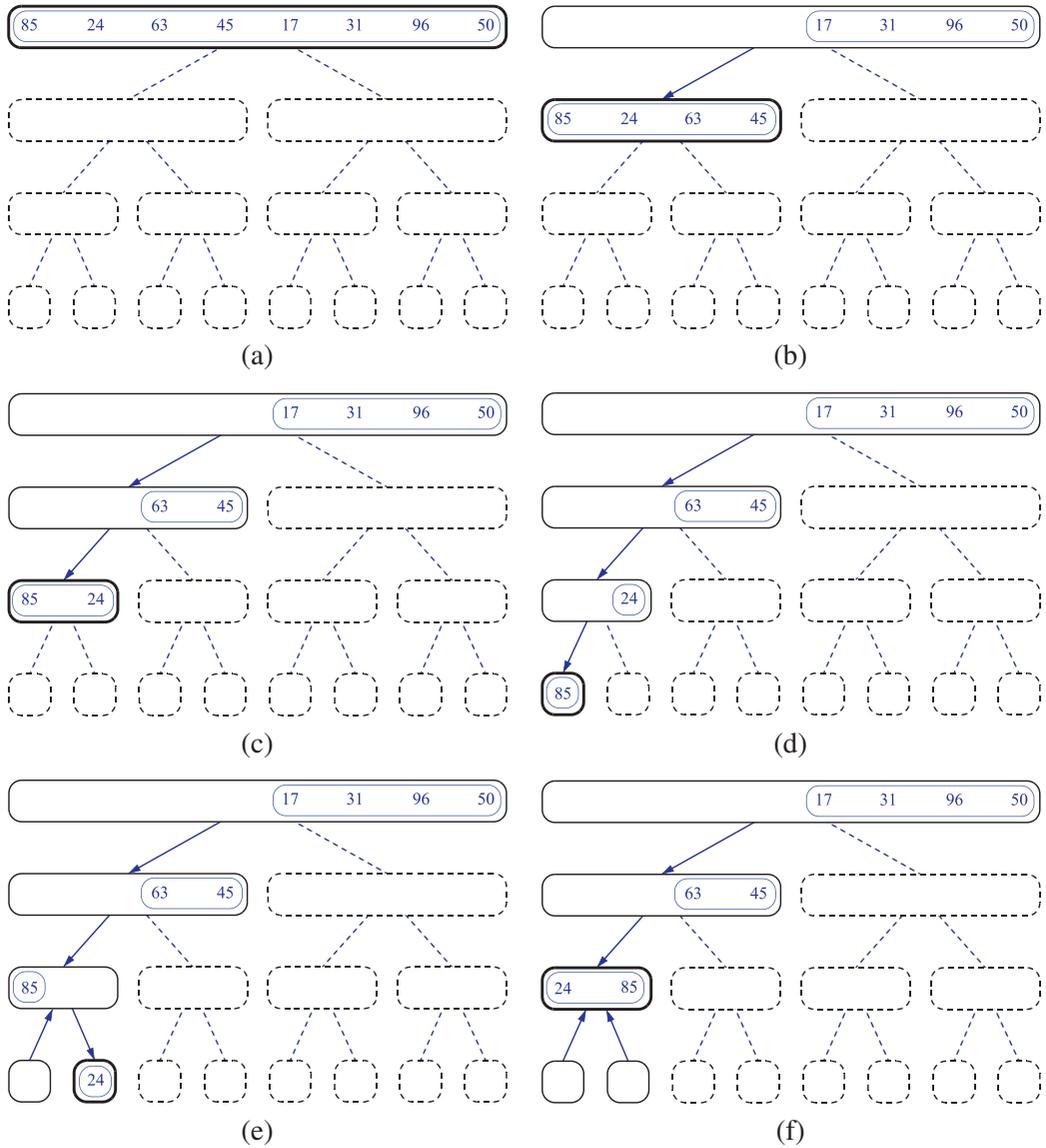
Figure 11.2: Visualization of an execution of merge-sort. Each node of the tree represents a recursive call of merge-sort. The nodes drawn with dashed lines represent calls that have not been made yet. The node drawn with thick lines represents the current call. The empty nodes drawn with thin lines represent completed calls. The remaining nodes (drawn with thin lines and not empty) represent calls that are waiting for a child invocation to return. (Continues in Figure 11.3.)
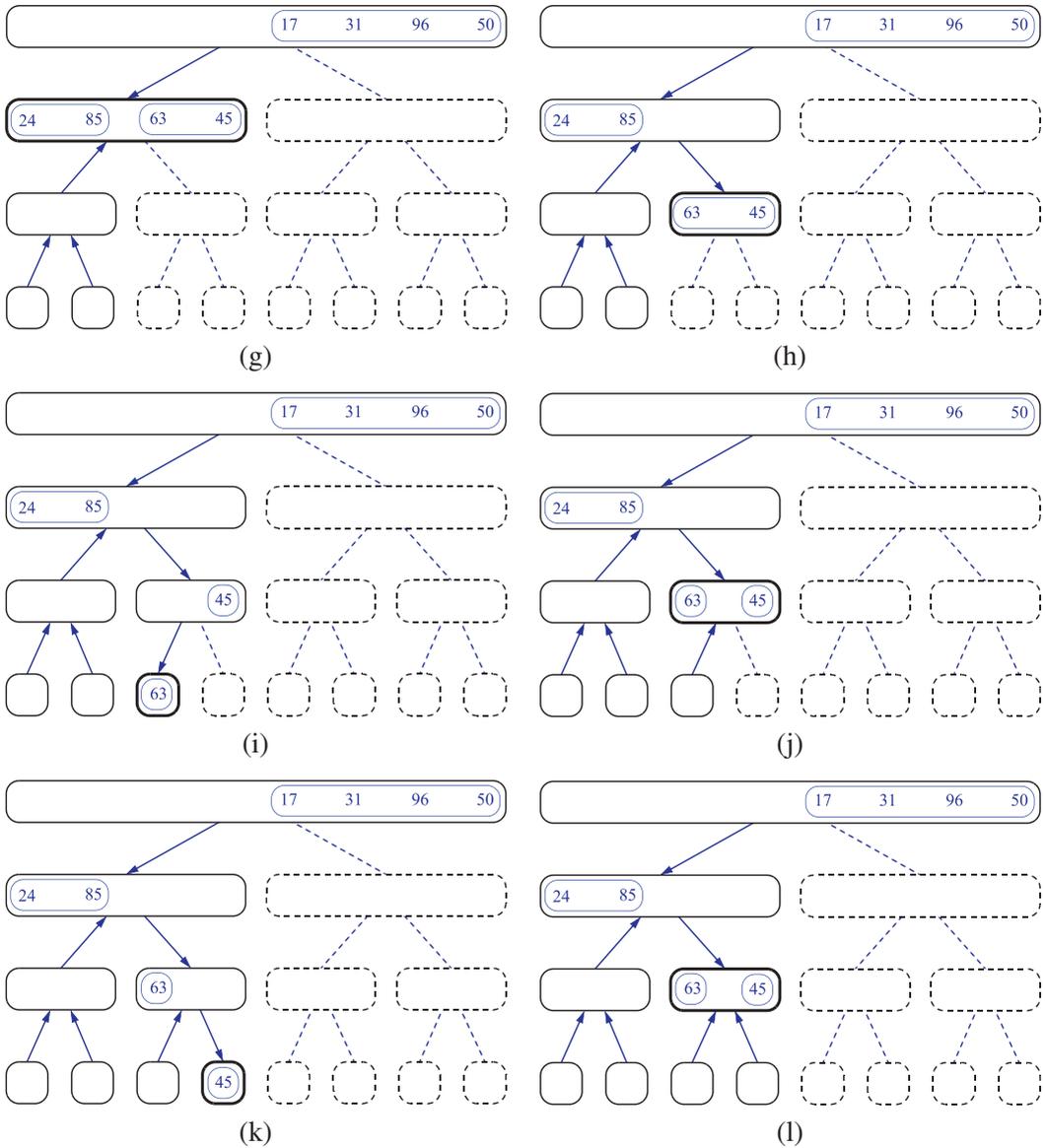
Figure 11.3: Visualization of an execution of merge-sort. (Continues in Figure 11.4.)
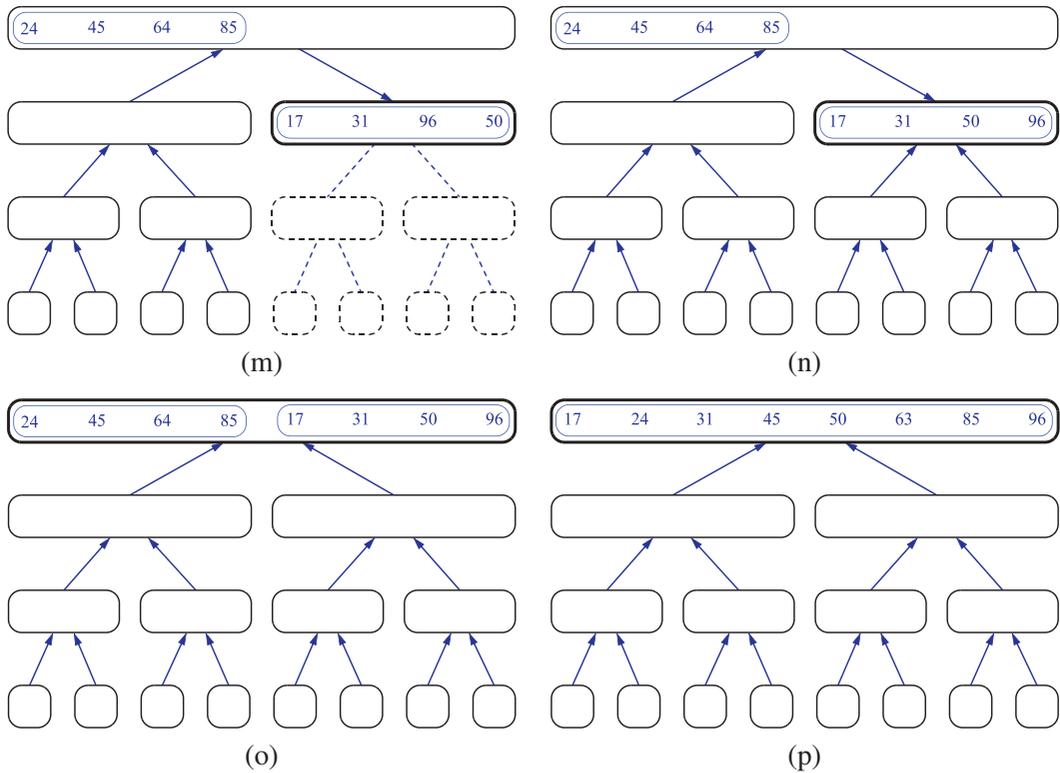
Figure 11.4: Visualization of an execution of merge-sort. Several invocations are omitted between (l) and (m) and between (m) and (n). Note the conquer step performed in step (p). (Continued from Figure 11.3.)

Proposition 11.1: *The merge-sort tree associated with an execution of merge-sort on a sequence of size n has height $\lceil \log n \rceil$.*

We leave the justification of Proposition 11.1 as a simple exercise (R-11.4). We use this proposition to analyze the running time of the merge-sort algorithm.

Having given an overview of merge-sort and an illustration of how it works, let us consider each of the steps of this divide-and-conquer algorithm in more detail. The divide and recur steps of the merge-sort algorithm are simple; dividing a sequence of size n involves separating it at the element with index $\lceil n/2 \rceil$, and the recursive calls simply involve passing these smaller sequences as parameters. The difficult step is the conquer step, which merges two sorted sequences into a single sorted sequence. Thus, before we present our analysis of merge-sort, we need to say more about how this is done.

11.1.2 Merging Arrays and Lists

To merge two sorted sequences, it is helpful to know if they are implemented as arrays or lists. We begin with the array implementation, which we show in Code Fragment 11.1. We illustrate a step in the merge of two sorted arrays in Figure 11.5.

Algorithm merge(S_1, S_2, S):

> **Input:** Sorted sequences S_1 and S_2 and an empty sequence S, all of which are implemented as arrays
>
> **Output:** Sorted sequence S containing the elements from S_1 and S_2
>
> $i \leftarrow j \leftarrow 0$
> **while** $i < S_1.\text{size}()$ **and** $j < S_2.\text{size}()$ **do**
>> **if** $S_1[i] \leq S_2[j]$ **then**
>>> $S.\text{insertBack}(S_1[i])$ {copy ith element of S_1 to end of S}
>>> $i \leftarrow i + 1$
>>
>> **else**
>>> $S.\text{insertBack}(S_2[j])$ {copy jth element of S_2 to end of S}
>>> $j \leftarrow j + 1$
>
> **while** $i < S_1.\text{size}()$ **do** {copy the remaining elements of S_1 to S}
>> $S.\text{insertBack}(S_1[i])$
>> $i \leftarrow i + 1$
>
> **while** $j < S_2.\text{size}()$ **do** {copy the remaining elements of S_2 to S}
>> $S.\text{insertBack}(S_2[j])$
>> $j \leftarrow j + 1$

Code Fragment 11.1: Algorithm for merging two sorted array-based sequences.
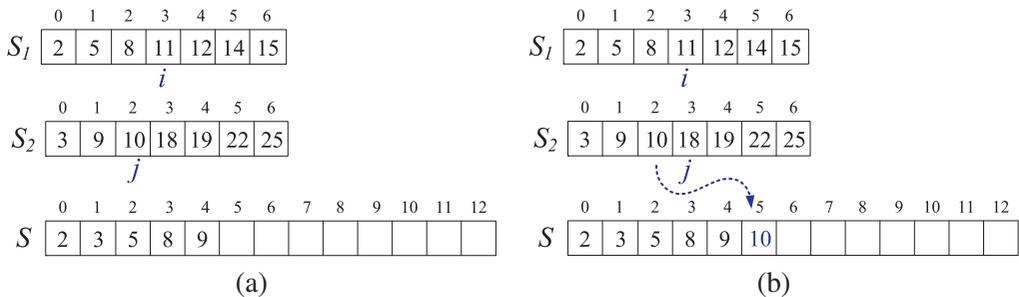
Figure 11.5: A step in the merge of two sorted arrays: (a) before the copy step; (b) after the copy step.

Merging Two Sorted Lists

In Code Fragment 11.2, we give a list-based version of algorithm merge, for merging two sorted sequences, S_1 and S_2, implemented as linked lists. The main idea is to iteratively remove the smallest element from the front of one of the two lists and add it to the end of the output sequence, S, until one of the two input lists is empty, at which point we copy the remainder of the other list to S. We show an example execution of this version of algorithm merge in Figure 11.6.

Algorithm merge(S_1, S_2, S):
    Input: Sorted sequences S_1 and S_2 and an empty sequence S, implemented as
        linked lists
    Output: Sorted sequence S containing the elements from S_1 and S_2
    while S_1 is not empty **and** S_2 is not empty **do**
        if S_1.front().element() $\leq$ S_2.front().element() **then**
            {move the first element of S_1 at the end of S}
            S.insertBack(S_1.eraseFront())
        else
            {move the first element of S_2 at the end of S}
            S.insertBack(S_2.eraseFront())
    {move the remaining elements of S_1 to S}
    while S_1 is not empty **do**
        S.insertBack(S_1.eraseFront())
    {move the remaining elements of S_2 to S}
    while S_2 is not empty **do**
        S.insertBack(S_2.eraseFront())

Code Fragment 11.2: Algorithm merge for merging two sorted sequences implemented as linked lists.

The Running Time for Merging

We analyze the running time of the merge algorithm by making some simple observations. Let n_1 and n_2 be the number of elements of S_1 and S_2, respectively. Algorithm merge has three **while** loops. Independent of whether we are analyzing the array-based version or the list-based version, the operations performed inside each loop take $O(1)$ time each. The key observation is that during each iteration of one of the loops, one element is copied or moved from either S_1 or S_2 into S (and that element is no longer considered). Since no insertions are performed into S_1 or S_2, this observation implies that the overall number of iterations of the three loops is $n_1 + n_2$. Thus, the running time of algorithm merge is $O(n_1 + n_2)$.
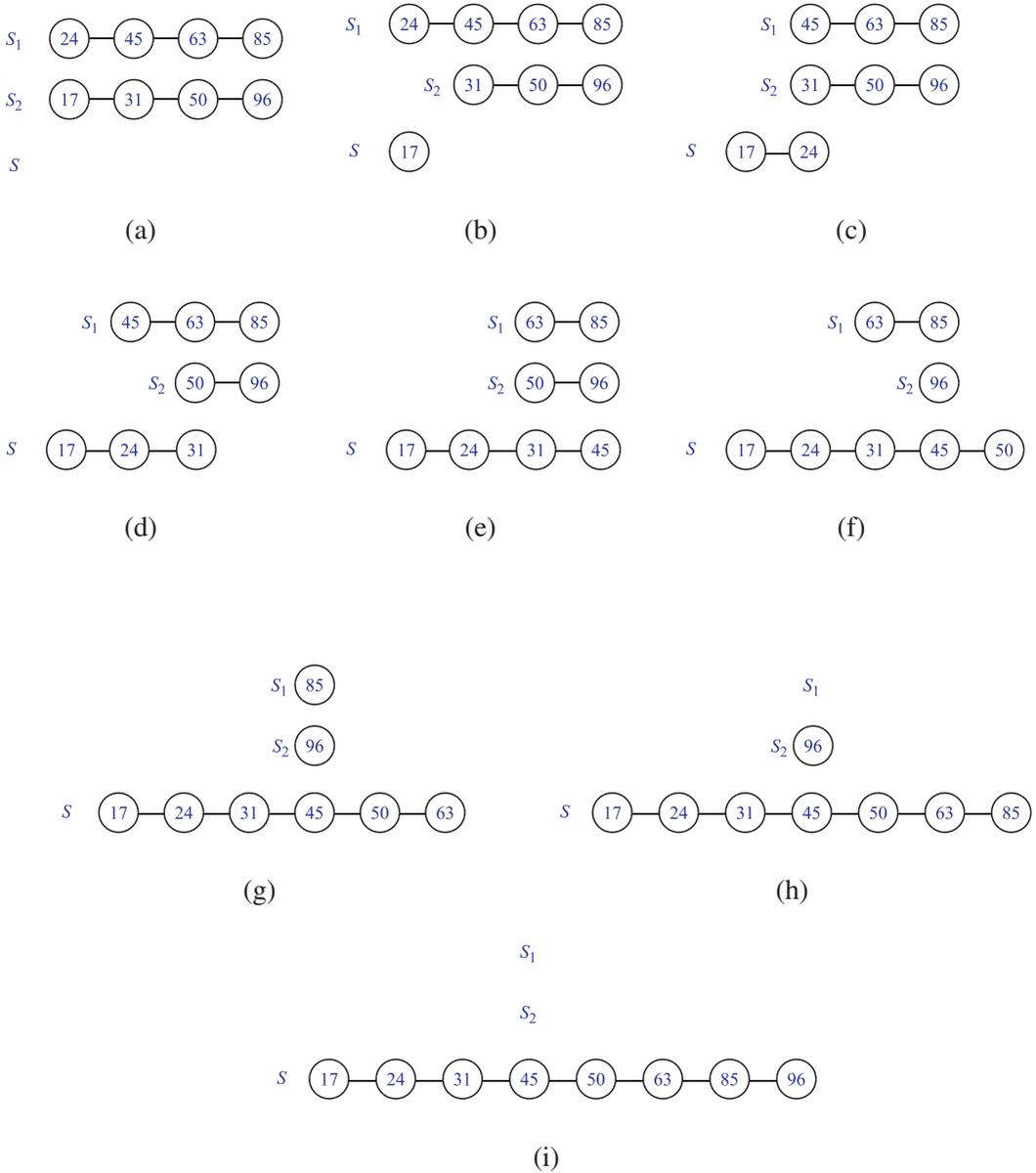
Figure 11.6: An execution of the algorithm merge shown in Code Fragment 11.2.

11.1.3   The Running Time of Merge-Sort

Now that we have given the details of the merge-sort algorithm in both its array-based and list-based versions, and we have analyzed the running time of the crucial merge algorithm used in the conquer step, let us analyze the running time of the entire merge-sort algorithm, assuming it is given an input sequence of n elements. For simplicity, we restrict our attention to the case where n is a power of 2. We leave it as an exercise (Exercise R-11.7) to show that the result of our analysis also holds when n is not a power of 2.

As we did in the analysis of the merge algorithm, we assume that the input sequence S and the auxiliary sequences S_1 and S_2, created by each recursive call of merge-sort, are implemented by either arrays or linked lists (the same as S), so that merging two sorted sequences can be done in linear time.

As we mentioned earlier, we analyze the merge-sort algorithm by referring to the merge-sort tree T. (Recall Figures 11.2 through 11.4.) We call the ***time spent at a node*** v of T the running time of the recursive call associated with v, excluding the time taken waiting for the recursive calls associated with the children of v to terminate. In other words, the time spent at node v includes the running times of the divide and conquer steps, but excludes the running time of the recur step. We have already observed that the details of the divide step are straightforward; this step runs in time proportional to the size of the sequence for v. In addition, as discussed above, the conquer step, which consists of merging two sorted subsequences, also takes linear time, independent of whether we are dealing with arrays or linked lists. That is, letting i denote the depth of node v, the time spent at node v is $O(n/2^i)$, since the size of the sequence handled by the recursive call associated with v is equal to $n/2^i$.

Looking at the tree T more globally, as shown in Figure 11.7, we see that, given our definition of "time spent at a node," the running time of merge-sort is equal to the sum of the times spent at the nodes of T. Observe that T has exactly 2^i nodes at depth i. This simple observation has an important consequence, for it implies that the overall time spent at all the nodes of T at depth i is $O(2^i \cdot n/2^i)$, which is $O(n)$. By Proposition 11.1, the height of T is $\lceil \log n \rceil$. Thus, since the time spent at each of the $\lceil \log n \rceil + 1$ levels of T is $O(n)$, we have the following result.

Proposition 11.2: *Algorithm merge-sort sorts a sequence S of size n in $O(n \log n)$ time, assuming two elements of S can be compared in $O(1)$ time.*

In other words, the merge-sort algorithm asymptotically matches the fast running time of the heap-sort algorithm.
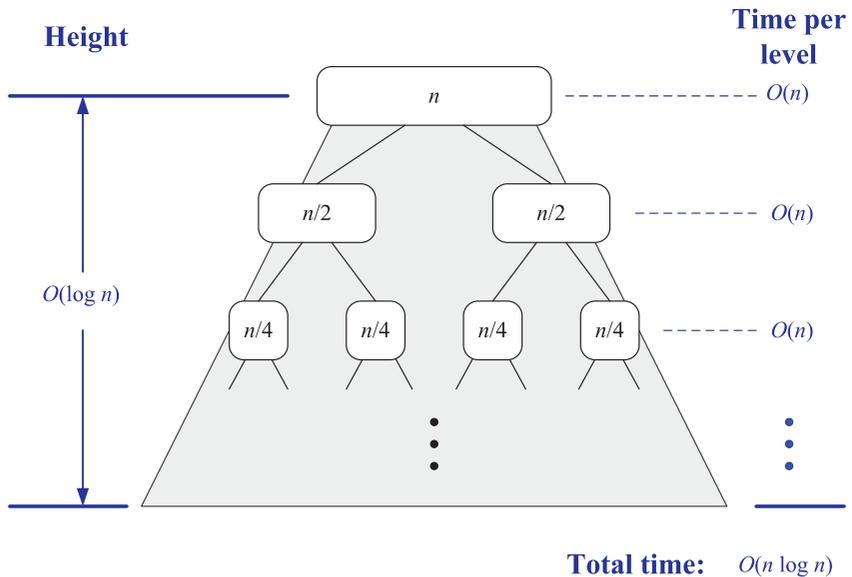
Figure 11.7: A visual time analysis of the merge-sort tree T. Each node is shown labeled with the size of its subproblem.

11.1.4   C++ Implementations of Merge-Sort

In this subsection, we present two complete C++ implementations of the merge-sort algorithm, one for lists and one for vectors. In both cases a comparator object (see Section 8.1.2) is used to decide the relative order of the elements. Recall that a comparator is a class that implements the less-than operator by overloading the "()" operator. For example, given a comparator object *less*, the relational test $x < y$ can be implemented with $less(x, y)$, and the test $x \leq y$ can be implemented as $!less(y, x)$.

First, in Code Fragment 11.3, we present a C++ implementation of a list-based merge-sort algorithm as the recursive function mergeSort. We represent each sequence as an STL list (Section 6.2.4). The merge process is loosely based on the algorithm presented in Code Fragment 11.2. The main function mergeSort partitions the input list S into two auxiliary lists, S_1 and S_2, of roughly equal sizes. They are each sorted recursively, and the results are then combined by invoking the function merge. The function merge repeatedly moves the smaller element of the two lists S_1 and S_2 into the output list S.

Functions from our list ADT, such as front and insertBack, have been replaced by their STL equivalents, such as begin and push_back, respectively. Access to elements of the list is provided by list iterators. Given an iterator p, recall that $*p$ accesses the current element, and $*p++$ accesses the current element and increments the iterator to the next element of the list.

Each list is modified by insertions and deletions only at the head and tail; hence, each list update takes $O(1)$ time, assuming any list implementation based on doubly linked lists (see Table 6.2). For a list S of size n, function mergeSort(S,c) runs in time $O(n \log n)$.

```
template <typename E, typename C>              // merge-sort S
void mergeSort(list<E>& S, const C& less) {
  typedef typename list<E>::iterator Itor;     // sequence of elements
  int n = S.size();
  if (n <= 1) return;                          // already sorted
  list<E> S1, S2;
  Itor p = S.begin();
  for (int i = 0; i < n/2; i++) S1.push_back(*p++); // copy first half to S1
  for (int i = n/2; i < n; i++) S2.push_back(*p++); // copy second half to S2
  S.clear();                                   // clear S's contents
  mergeSort(S1, less);                         // recur on first half
  mergeSort(S2, less);                         // recur on second half
  merge(S1, S2, S, less);                      // merge S1 and S2 into S
}

template <typename E, typename C>              // merge utility
void merge(list<E>& S1, list<E>& S2, list<E>& S, const C& less) {
  typedef typename list<E>::iterator Itor;     // sequence of elements
  Itor p1 = S1.begin();
  Itor p2 = S2.begin();
  while(p1 != S1.end() && p2 != S2.end()) {    // until either is empty
    if(less(*p1, *p2))                         // append smaller to S
      S.push_back(*p1++);
    else
      S.push_back(*p2++);
  }
  while(p1 != S1.end())                        // copy rest of S1 to S
    S.push_back(*p1++);
  while(p2 != S2.end())                        // copy rest of S2 to S
    S.push_back(*p2++);
}
```

Code Fragment 11.3: Functions mergeSort and merge implementing a list-based merge-sort algorithm.

Next, in Code Fragment 11.4, we present a nonrecursive vector-based version of merge-sort, which also runs in $O(n \log n)$ time. It is a bit faster than recursive list-based merge-sort in practice, as it avoids the extra overheads of recursive calls and node creation. The main idea is to perform merge-sort bottom-up, performing the merges level-by-level going up the merge-sort tree. The input is an STL vector S.

We begin by merging every odd-even pair of elements into sorted runs of length

two. In order to keep from overwriting the vector elements, we copy elements from an input vector *in* to an output vector *out*. For example, we merge *in*[0] and *in*[1] into the subvector *out*[0..1], then we merge *in*[2] and *in*[3] into the subvector *out*[2..3], and so on.

We then swap the rolls of *in* and *out*, and we merge these runs of length two into runs of length four. For example we merge *in*[0..1] with *in*[2..3] into the subvector *out*[0..3], then we merge *in*[4..5] with *in*[6..7] into the subvector *out*[4..7]. We then merge consecutive runs of length four into new runs of length eight, and so on, until the array is sorted.

The variable *b* stores the start of the runs and *m* stores their length. The variables *i*, *j*, and *k* store the current indices in the various subvectors. When swapping vectors, we do not copy their entire contents. Instead, we maintain pointers to the two arrays and swap these pointers at the end of each round of subvector merges.

11.1.5 Merge-Sort and Recurrence Equations ⋆

There is another way to justify that the running time of the merge-sort algorithm is $O(n\log n)$ (Proposition 11.2). Namely, we can deal more directly with the recursive nature of the merge-sort algorithm. In this section, we present such an analysis of the running time of merge-sort, and in so doing introduce the mathematical concept of a ***recurrence equation*** (also known as ***recurrence relation***).

Let the function $t(n)$ denote the worst-case running time of merge-sort on an input sequence of size n. Since merge-sort is recursive, we can characterize function $t(n)$ by means of an equation where the function $t(n)$ is recursively expressed in terms of itself. In order to simplify our characterization of $t(n)$, let us focus our attention on the case where n is a power of 2. We leave the problem of showing that our asymptotic characterization still holds in the general case as an exercise (Exercise R-11.7). In this case, we can specify the definition of $t(n)$ as

$$t(n) = \begin{cases} b & \text{if } n \leq 1 \\ 2t(n/2) + cn & \text{otherwise.} \end{cases}$$

An expression such as the one above is called a ***recurrence equation***, since the function appears on both the left- and right-hand sides of the equal sign. Although such a characterization is correct and accurate, what we really want is a big-Oh type of characterization of $t(n)$ that does not involve the function $t(n)$ itself. That is, we want a ***closed-form*** characterization of $t(n)$.

We can obtain a closed-form solution by applying the definition of a recurrence equation, assuming n is relatively large. For example, after one more application of the equation above, we can write a new recurrence for $t(n)$ as

$$\begin{aligned} t(n) &= 2(2t(n/2^2) + (cn/2)) + cn \\ &= 2^2 t(n/2^2) + 2(cn/2) + cn = 2^2 t(n/2^2) + 2cn. \end{aligned}$$

```
template <typename E, typename C>              // merge-sort S
void mergeSort(vector<E>& S, const C& less) {
  typedef vector<E> vect;
  int n = S.size();
  vect v1(S);  vect* in = &v1;                  // initial input vector
  vect v2(n);  vect* out = &v2;                 // initial output vector
  for (int m = 1; m < n; m *= 2) {              // list sizes doubling
    for (int b = 0; b < n; b += 2*m) {          // beginning of list
      merge(*in, *out, less, b, m);             // merge sublists
    }
    std::swap(in, out);                         // swap input with output
  }
  S = *in;                                      // copy sorted array to S
}
                           // merge in[b. .b+m-1] and in[b+m. .b+2*m-1]
template <typename E, typename C>
void merge(vector<E>& in, vector<E>& out, const C& less, int b, int m) {
  int i = b;                                    // index into run #1
  int j = b + m;                                // index into run #2
  int n = in.size();
  int e1 = std::min(b + m, n);                  // end of run #1
  int e2 = std::min(b + 2*m, n);                // end of run #2
  int k = b;
  while ((i < e1) && (j < e2)) {
    if(!less(in[j], in[i]))                     // append smaller to S
      out[k++] = in[i++];
    else
      out[k++] = in[j++];
  }
  while (i < e1)                                // copy rest of run 1 to S
    out[k++] = in[i++];
  while (j < e2)                                // copy rest of run 2 to S
    out[k++] = in[j++];
}
```

Code Fragment 11.4: Functions mergeSort and merge implementing a vector-based merge-sort algorithm. We employ the STL functions swap, which swaps two values, and min, which returns the minimum of two values. Note that the call to swap swaps only the pointers to the arrays and, hence, runs in $O(1)$ time.

If we apply the equation again, we get $t(n) = 2^3 t(n/2^3) + 3cn$. At this point, we should see a pattern emerging, so that after applying this equation i times we get

$$t(n) = 2^i t(n/2^i) + icn.$$

The issue that remains, then, is to determine when to stop this process. To see when to stop, recall that we switch to the closed form $t(n) = b$ when $n \leq 1$, which occurs when $2^i = n$. In other words, this occurs when $i = \log n$. Making this substitution, then, yields

$$
\begin{aligned}
t(n) &= 2^{\log n} t(n/2^{\log n}) + (\log n)cn \\
&= nt(1) + cn \log n \\
&= nb + cn \log n.
\end{aligned}
$$

That is, we get an alternative justification of the fact that $t(n)$ is $O(n \log n)$.

11.2 Quick-Sort

The next sorting algorithm we discuss is called **quick-sort**. Like merge-sort, this algorithm is also based on the **divide-and-conquer** paradigm, but it uses this technique in a somewhat opposite manner, as all the hard work is done **before** the recursive calls.

High-Level Description of Quick-Sort

The quick-sort algorithm sorts a sequence S using a simple recursive approach. The main idea is to apply the divide-and-conquer technique, whereby we divide S into subsequences, recur to sort each subsequence, and then combine the sorted subsequences by a simple concatenation. In particular, the quick-sort algorithm consists of the following three steps (see Figure 11.8):

1. **Divide:** If S has at least two elements (nothing needs to be done if S has zero or one element), select a specific element x from S, which is called the **pivot**. As is common practice, choose the pivot x to be the last element in S. Remove all the elements from S and put them into three sequences:
   - L, storing the elements in S less than x
   - E, storing the elements in S equal to x
   - G, storing the elements in S greater than x.

   Of course, if the elements of S are all distinct, then E holds just one element— the pivot itself.
2. **Recur:** Recursively sort sequences L and G.
3. **Conquer:** Put back the elements into S in order by first inserting the elements of L, then those of E, and finally those of G.
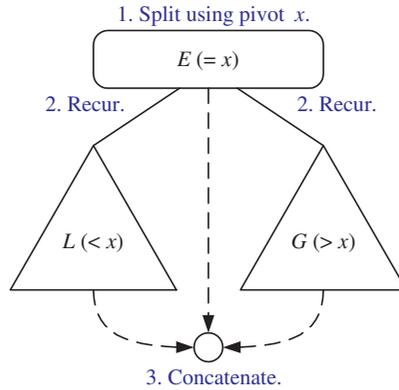
Figure 11.8: A visual schematic of the quick-sort algorithm.

Like merge-sort, the execution of quick-sort can be visualized by means of a bi-
nary recursion tree, called the ***quick-sort tree***. Figure 11.9 summarizes an execution
of the quick-sort algorithm by showing the input and output sequences processed at
each node of the quick-sort tree. The step-by-step evolution of the quick-sort tree
is shown in Figures 11.10, 11.11, and 11.12.

Unlike merge-sort, however, the height of the quick-sort tree associated with
an execution of quick-sort is linear in the worst case. This happens, for example,
if the sequence consists of n distinct elements and is already sorted. Indeed, in this
case, the standard choice of the pivot as the largest element yields a subsequence
L of size $n-1$, while subsequence E has size 1 and subsequence G has size 0. At
each invocation of quick-sort on subsequence L, the size decreases by 1. Hence,
the height of the quick-sort tree is $n-1$.

Performing Quick-Sort on Arrays and Lists

In Code Fragment 11.5, we give a pseudo-code description of the quick-sort algo-
rithm that is efficient for sequences implemented as arrays or linked lists. The algo-
rithm follows the template for quick-sort given above, adding the detail of scanning
the input sequence S backwards to divide it into the lists L, E, and G of elements
that are respectively less than, equal to, and greater than the pivot. We perform this
scan backwards, since removing the last element in a sequence is a constant-time
operation independent of whether the sequence is implemented as an array or a
linked list. We then recur on the L and G lists, and copy the sorted lists L, E, and
G back to S. We perform this latter set of copies in the forward direction, since in-
serting elements at the end of a sequence is a constant-time operation independent
of whether the sequence is implemented as an array or a linked list.
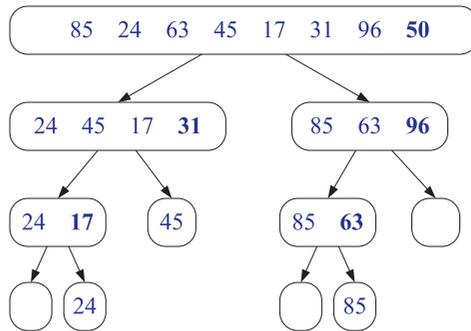
(a)
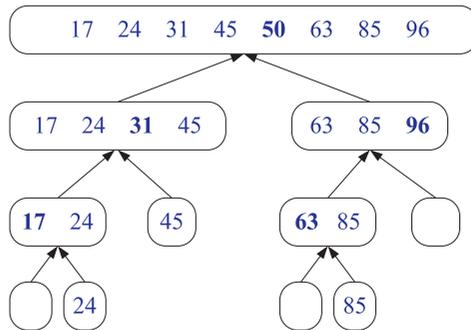
(b)

Figure 11.9: Quick-sort tree T for an execution of the quick-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of T; (b) output sequences generated at each node of T. The pivot used at each level of the recursion is shown in bold.
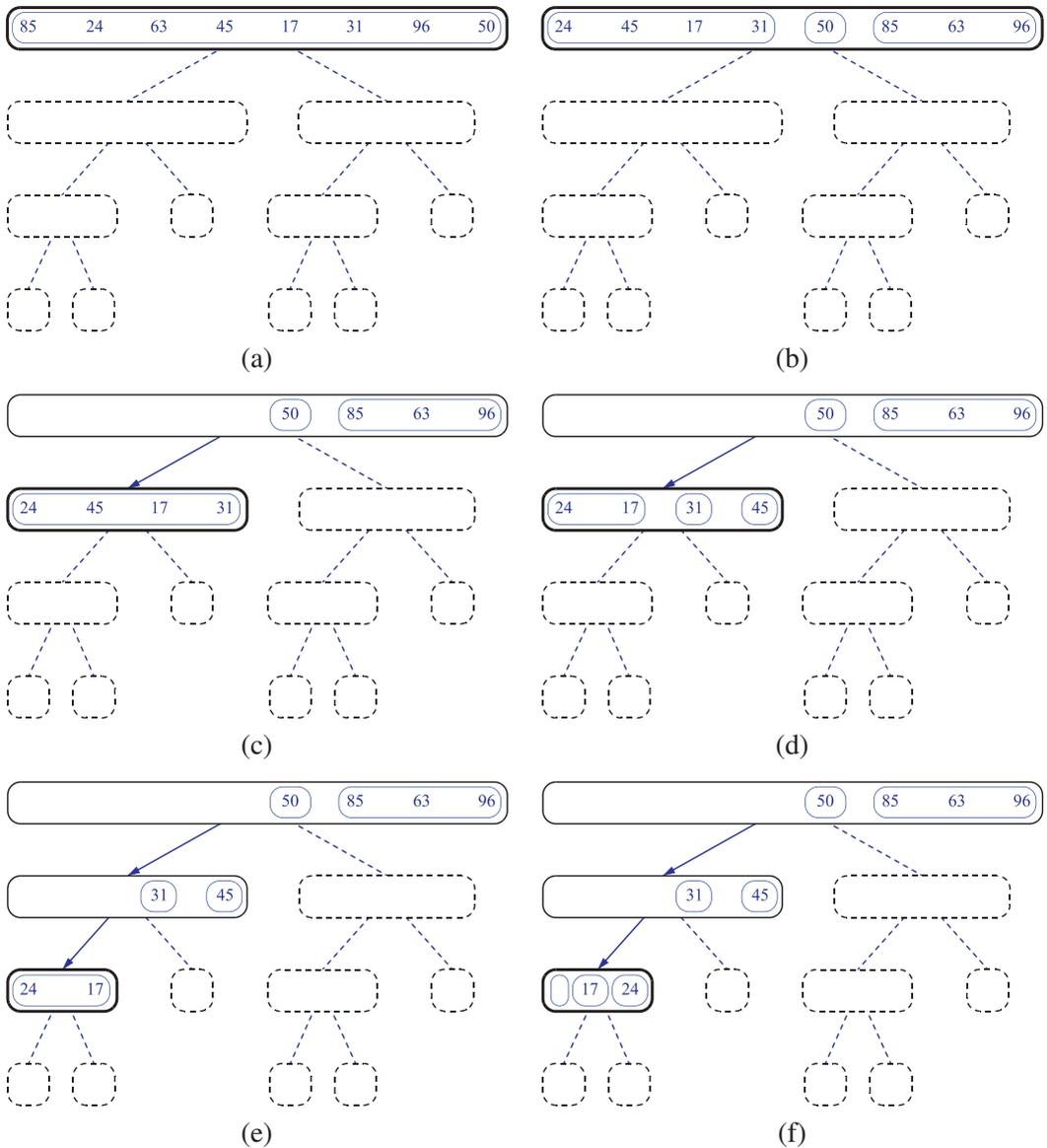
Figure 11.10: Visualization of quick-sort. Each node of the tree represents a recursive call. The nodes drawn with dashed lines represent calls that have not been made yet. The node drawn with thick lines represents the running invocation. The empty nodes drawn with thin lines represent terminated calls. The remaining nodes represent suspended calls (that is, active invocations that are waiting for a child invocation to return). Note the divide steps performed in (b), (d), and (f). (Continues in Figure 11.11.)
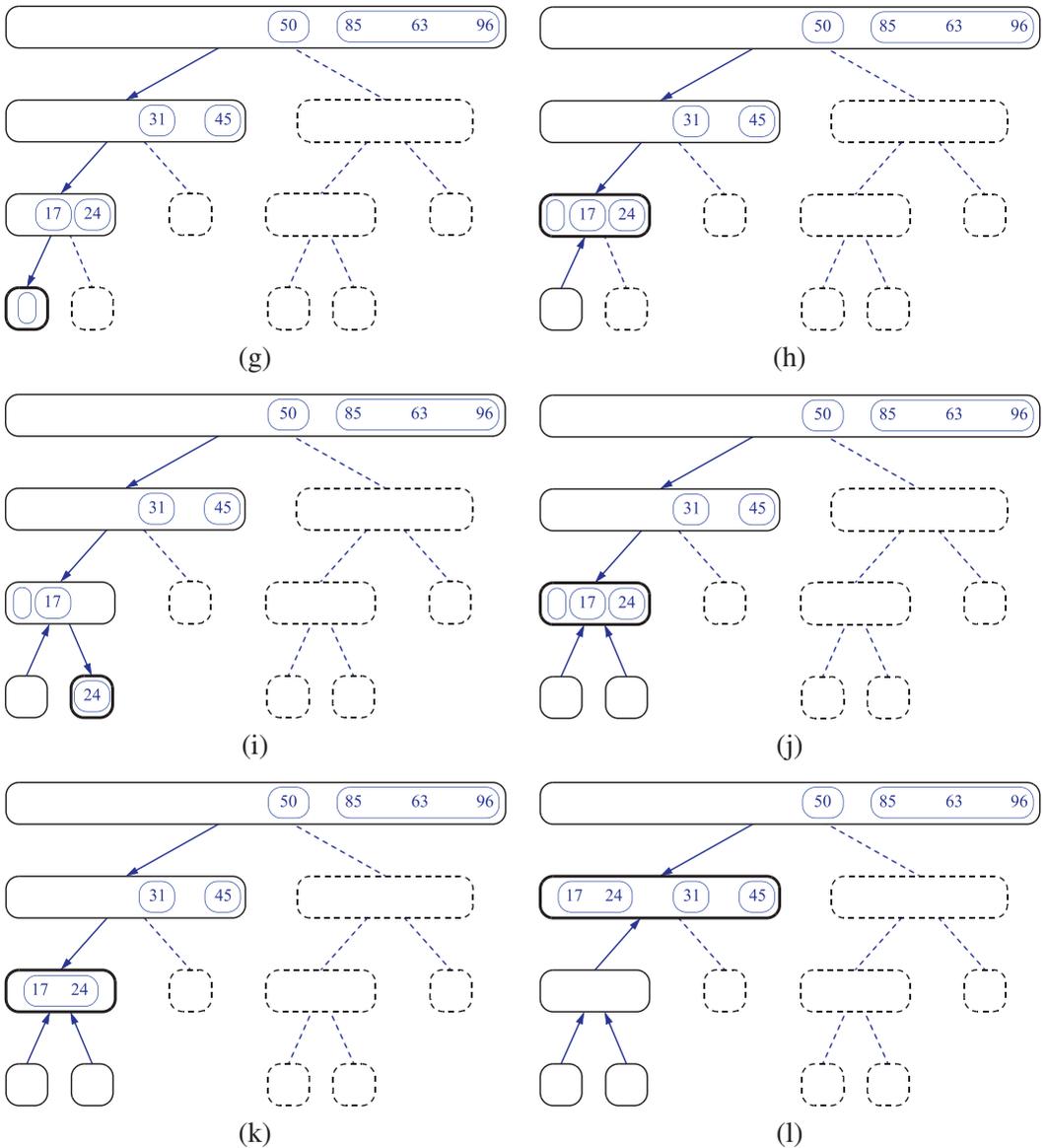
Figure 11.11: Visualization of an execution of quick-sort. Note the conquer step performed in (k). (Continues in Figure 11.12.)
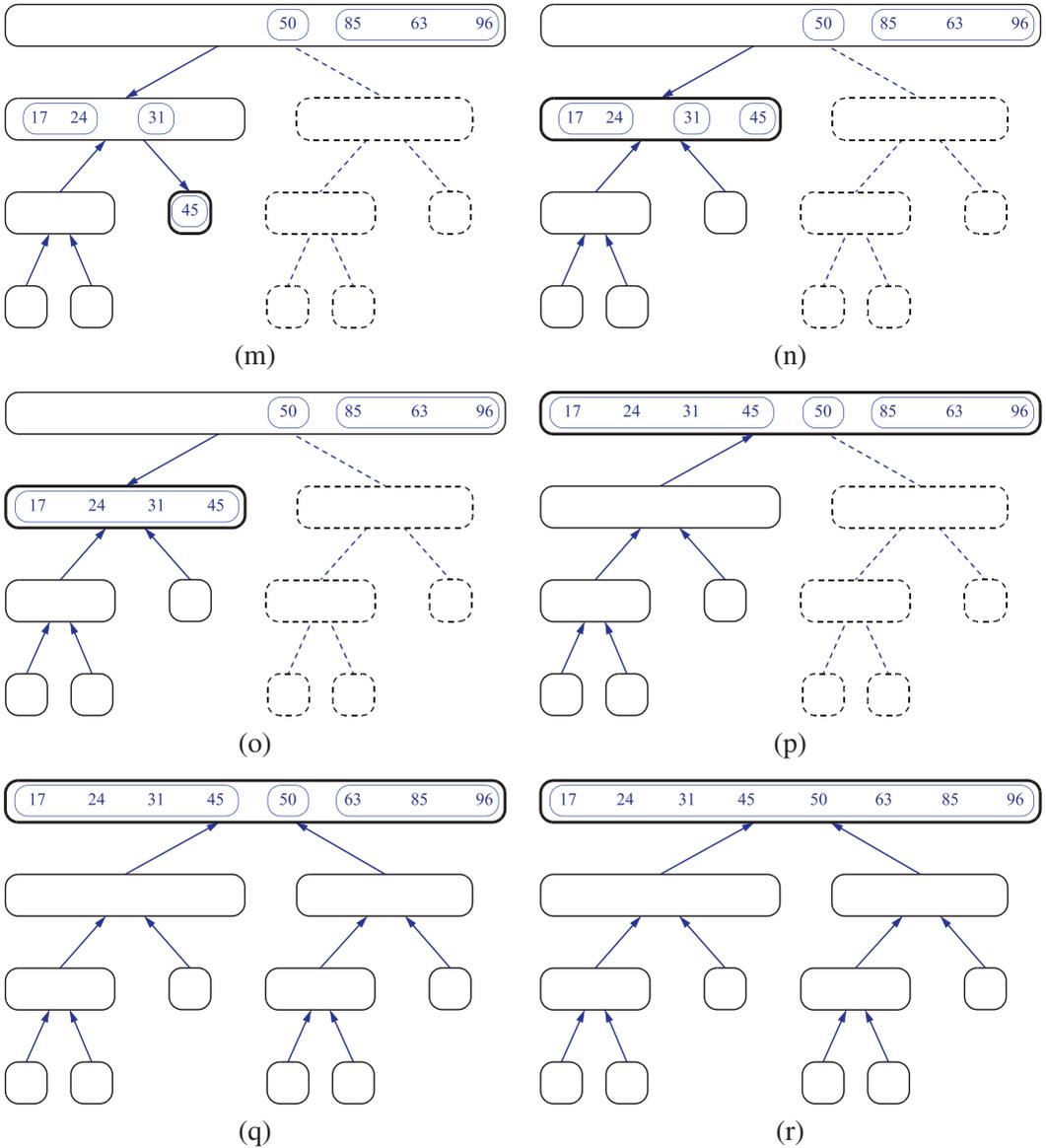
Figure 11.12: Visualization of an execution of quick-sort. Several invocations between (p) and (q) have been omitted. Note the conquer steps performed in (o) and (r). (Continued from Figure 11.11.)

Algorithm QuickSort(*S*):
   Input: A sequence *S* implemented as an array or linked list
   Output: The sequence *S* in sorted order
   if *S*.size() $\leq$ 1 **then**
     return   {*S* is already sorted in this case}
   $p \leftarrow S$.back().element()   {the pivot}
   Let *L*, *E*, and *G* be empty list-based sequences
   while !*S*.empty() **do** {scan *S* backwards, dividing it into *L*, *E*, and *G*}
     if *S*.back().element() < *p* **then**
       L.insertBack(*S*.eraseBack())
     else if *S*.back().element() = *p* **then**
       E.insertBack(*S*.eraseBack())
     else {the last element in *S* is greater than *p*}
       G.insertBack(*S*.eraseBack())
   QuickSort(*L*)   {Recur on the elements less than *p*}
   QuickSort(*G*)   {Recur on the elements greater than *p*}
   while !*L*.empty() **do** {copy back to *S* the sorted elements less than *p*}
     S.insertBack(*L*.eraseFront())
   while !*E*.empty() **do** {copy back to *S* the elements equal to *p*}
     S.insertBack(*E*.eraseFront())
   while !*G*.empty() **do** {copy back to *S* the sorted elements greater than *p*}
     S.insertBack(*G*.eraseFront())
   return   {*S* is now in sorted order}

Code Fragment 11.5: Quick-sort for an input sequence *S* implemented with a linked list or an array.

Running Time of Quick-Sort

We can analyze the running time of quick-sort with the same technique used for merge-sort in Section 11.1.3. Namely, we can identify the time spent at each node of the quick-sort tree T and sum up the running times for all the nodes.

Examining Code Fragment 11.5, we see that the divide step and the conquer step of quick-sort can be implemented in linear time. Thus, the time spent at a node v of T is proportional to the ***input size*** $s(v)$ of v, defined as the size of the sequence handled by the invocation of quick-sort associated with node v. Since subsequence E has at least one element (the pivot), the sum of the input sizes of the children of v is at most $s(v) - 1$.

Given a quick-sort tree T, let s_i denote the sum of the input sizes of the nodes at depth i in T. Clearly, $s_0 = n$, since the root r of T is associated with the entire sequence. Also, $s_1 \leq n - 1$, since the pivot is not propagated to the children of r. Consider next s_2. If both children of r have nonzero input size, then $s_2 = n - 3$. Otherwise (one child of the root has zero size, the other has size $n - 1$), $s_2 = n - 2$. Thus, $s_2 \leq n - 2$. Continuing this line of reasoning, we obtain that $s_i \leq n - i$. As observed in Section 11.2, the height of T is $n - 1$ in the worst case. Thus, the worst-case running time of quick-sort is $O\left(\sum_{i=0}^{n-1} s_i\right)$, which is $O\left(\sum_{i=0}^{n-1} (n-i)\right)$, that is, $O\left(\sum_{i=1}^{n} i\right)$. By Proposition 4.3, $\sum_{i=1}^{n} i$ is $O(n^2)$. Thus, quick-sort runs in $O(n^2)$ worst-case time.

Given its name, we would expect quick-sort to run quickly. However, the quadratic bound above indicates that quick-sort is slow in the worst case. Paradoxically, this worst-case behavior occurs for problem instances when sorting should be easy—if the sequence is already sorted.

Going back to our analysis, note that the best case for quick-sort on a sequence of distinct elements occurs when subsequences L and G happen to have roughly the same size. That is, in the best case, we have

$$
\begin{aligned}
s_0 &= n \\
s_1 &= n - 1 \\
s_2 &= n - (1 + 2) = n - 3 \\
&\;\;\vdots \\
s_i &= n - (1 + 2 + 2^2 + \cdots + 2^{i-1}) = n - (2^i - 1).
\end{aligned}
$$

Thus, in the best case, T has height $O(\log n)$ and quick-sort runs in $O(n \log n)$ time. We leave the justification of this fact as an exercise (Exercise R-11.12).

The informal intuition behind the expected behavior of quick-sort is that at each invocation the pivot will probably divide the input sequence about equally. Thus, we expect the average running time quick-sort to be similar to the best-case running time, that is, $O(n \log n)$. In the next section, we see that introducing randomization makes quick-sort behave exactly in this way.

11.2.1 Randomized Quick-Sort

One common method for analyzing quick-sort is to assume that the pivot always divides the sequence almost equally. We feel such an assumption would presuppose knowledge about the input distribution that is typically not available, however. For example, we would have to assume that we will rarely be given "almost" sorted sequences to sort, which are actually common in many applications. Fortunately, this assumption is not needed in order for us to match our intuition to quick-sort's behavior.

In general, we desire some way of getting close to the best-case running time for quick-sort. The way to get close to the best-case running time, of course, is for the pivot to divide the input sequence S almost equally. If this outcome were to occur, then it would result in a running time that is asymptotically the same as the best-case running time. That is, having pivots close to the "middle" of the set of elements leads to an $O(n \log n)$ running time for quick-sort.

Picking Pivots at Random

Since the goal of the partition step of the quick-sort method is to divide the sequence S almost equally, let us introduce randomization into the algorithm and pick a ***random element*** of the input sequence as the pivot. That is, instead of picking the pivot as the last element of S, we pick an element of S at random as the pivot, keeping the rest of the algorithm unchanged. This variation of quick-sort is called ***randomized quick-sort***. The following proposition shows that the expected running time of randomized quick-sort on a sequence with n elements is $O(n \log n)$. This expectation is taken over all the possible random choices the algorithm makes, and is independent of any assumptions about the distribution of the possible input sequences the algorithm is likely to be given.

Proposition 11.3: *The expected running time of randomized quick-sort on a sequence S of size n is $O(n \log n)$.*

Justification: We assume two elements of S can be compared in $O(1)$ time. Consider a single recursive call of randomized quick-sort, and let n denote the size of the input for this call. Say that this call is "good" if the pivot chosen is such that subsequences L and G have size at least $n/4$ and at most $3n/4$ each; otherwise, a call is "bad."

Now, consider the implications of our choosing a pivot uniformly at random. Note that there are $n/2$ possible good choices for the pivot for any given call of size n of the randomized quick-sort algorithm. Thus, the probability that any call is good is $1/2$. Note further that a good call will at least partition a list of size n into two lists of size $3n/4$ and $n/4$, and a bad call could be as bad as producing a single call of size $n-1$.

Now consider a recursion trace for randomized quick-sort. This trace defines a binary tree, T, such that each node in T corresponds to a different recursive call on a subproblem of sorting a portion of the original list.

Say that a node v in T is in *size group i* if the size of v's subproblem is greater than $(3/4)^{i+1}n$ and at most $(3/4)^i n$. Let us analyze the expected time spent working on all the subproblems for nodes in size group i. By the linearity of expectation (Proposition A.19), the expected time for working on all these subproblems is the sum of the expected times for each one. Some of these nodes correspond to good calls and some correspond to bad calls. But note that, since a good call occurs with probability $1/2$, the expected number of consecutive calls we have to make before getting a good call is 2. Moreover, notice that as soon as we have a good call for a node in size group i, its children will be in size groups higher than i. Thus, for any element x from the input list, the expected number of nodes in size group i containing x in their subproblems is 2. In other words, the expected total size of all the subproblems in size group i is $2n$. Since the nonrecursive work we perform for any subproblem is proportional to its size, this implies that the total expected time spent processing subproblems for nodes in size group i is $O(n)$.

The number of size groups is $\log_{4/3} n$, since repeatedly multiplying by $3/4$ is the same as repeatedly dividing by $4/3$. That is, the number of size groups is $O(\log n)$. Therefore, the total expected running time of randomized quick-sort is $O(n\log n)$. (See Figure 11.13.) ■
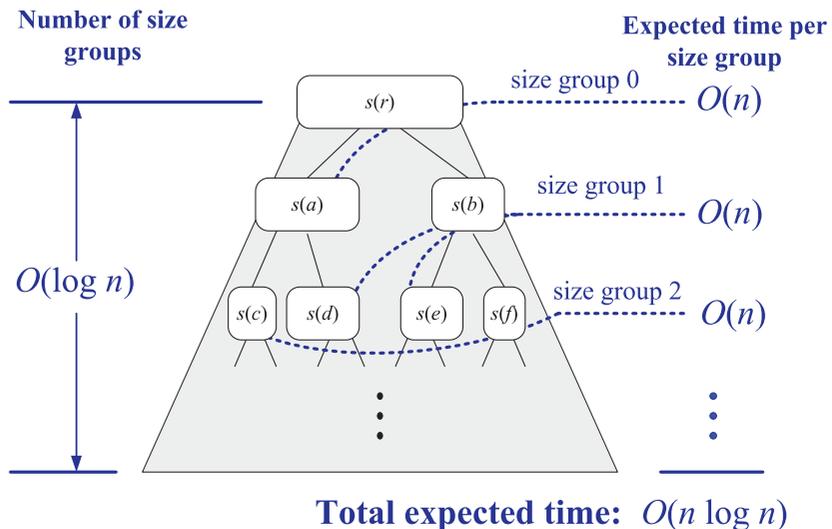
Figure 11.13: A visual time analysis of the quick-sort tree T. Each node is shown labeled with the size of its subproblem.

Actually, we can show that the running time of randomized quick-sort is $O(n\log n)$ with high probability.

11.2.2 C++ Implementations and Optimizations

Recall from Section 8.3.5 that a sorting algorithm is ***in-place*** if it uses only a small amount of memory in addition to that needed for the objects being sorted themselves. The merge-sort algorithm, as described above, does not use this optimization technique, and making it be in-place seems to be quite difficult. In-place sorting is not inherently difficult, however. For, as with heap-sort, quick-sort can be adapted to be in-place, and this is the version of quick-sort that is used in most deployed implementations.

Performing the quick-sort algorithm in-place requires a bit of ingenuity, however, for we must use the input sequence itself to store the subsequences for all the recursive calls. We show algorithm inPlaceQuickSort, which performs in-place quick-sort, in Code Fragment 11.6. Algorithm inPlaceQuickSort assumes that the input sequence, S, is given as an array of ***distinct*** elements. The reason for this restriction is explored in Exercise R-11.15. The extension to the general case is discussed in Exercise C-11.9.

Algorithm inPlaceQuickSort(S, a, b):

    Input: An array S of distinct elements; integers a and b

    Output: Array S with elements originally from indices from a to b, inclusive,
       sorted in nondecreasing order from indices a to b

    if $a \geq b$ **then return**   {at most one element in subrange}
    $p \leftarrow S[b]$ {the pivot}
    $l \leftarrow a$ {will scan rightward}
    $r \leftarrow b - 1$ {will scan leftward}
    while $l \leq r$ **do**
      {find an element larger than the pivot}
      while $l \leq r$ **and** $S[l] \leq p$ **do**
        $l \leftarrow l + 1$
      {find an element smaller than the pivot}
      while $r \geq l$ **and** $S[r] \geq p$ **do**
        $r \leftarrow r - 1$
      if $l < r$ **then**
        swap the elements at $S[l]$ and $S[r]$
    {put the pivot into its final place}
    swap the elements at $S[l]$ and $S[b]$
    {recursive calls}
    inPlaceQuickSort($S, a, l - 1$)
    inPlaceQuickSort($S, l + 1, b$)
    {we are done at this point, since the sorted subarrays are already consecutive}

        Code Fragment 11.6: In-place quick-sort for an input array S.

In-place quick-sort modifies the input sequence using element swapping and does not explicitly create subsequences. Indeed, a subsequence of the input sequence is implicitly represented by a range of positions specified by a left-most index l and a right-most index r. The divide step is performed by scanning the array simultaneously from l forward and from r backward, swapping pairs of elements that are in reverse order as shown in Figure 11.14. When these two indices "meet," subvectors L and G are on opposite sides of the meeting point. The algorithm completes by recurring on these two subvectors.

In-place quick-sort reduces the running time caused by the creation of new sequences and the movement of elements between them by a constant factor. It is so efficient that the STL's sorting algorithm is based in part on quick-sort.
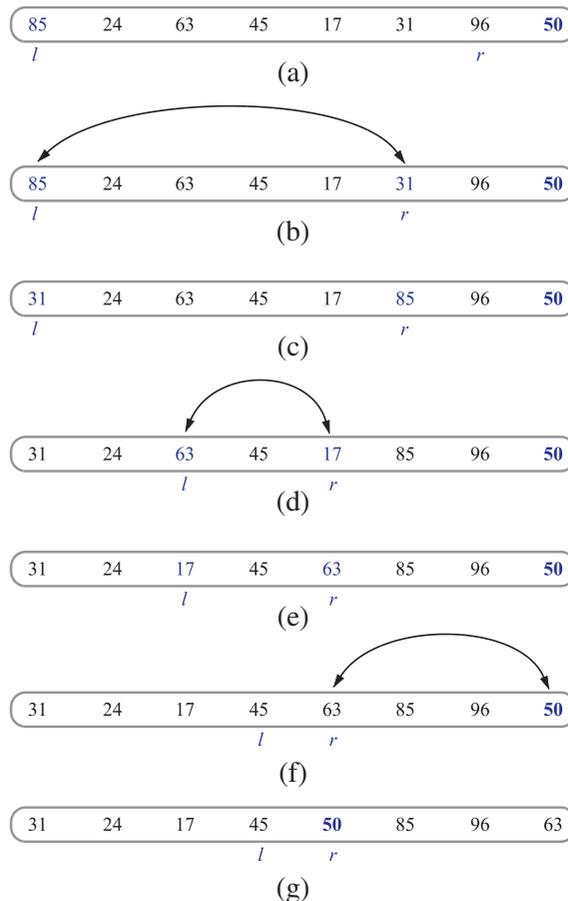
Figure 11.14: Divide step of in-place quick-sort. Index l scans the sequence from left to right, and index r scans the sequence from right to left. A swap is performed when l is at an element larger than the pivot and r is at an element smaller than the pivot. A final swap with the pivot completes the divide step.

We show a C++ version of in-place quick-sort in Code Fragment 11.7. The input to the sorting procedure is an STL vector of elements and a comparator object, which provides the less-than function. Our implementation is a straightforward adaptation of Code Fragment 11.6. The main procedure, quickSort, invokes the recursive procedure quickSortStep to do most of the work.

```
template <typename E, typename C>              // quick-sort S
void quickSort(std::vector<E>& S, const C& less) {
    if (S.size() <= 1) return;                 // already sorted
    quickSortStep(S, 0, S.size()−1, less);     // call sort utility
}

template <typename E, typename C>
void quickSortStep(std::vector<E>& S, int a, int b, const C& less) {
    if (a >= b) return;                        // 0 or 1 left? done
    E pivot = S[b];                            // select last as pivot
    int l = a;                                 // left edge
    int r = b − 1;                             // right edge
    while (l <= r) {
        while (l <= r && !less(pivot, S[l])) l++;   // scan right till larger
        while (r >= l && !less(S[r], pivot)) r−−;   // scan left till smaller
        if (l < r)                             // both elements found
            std::swap(S[l], S[r]);
    }                                          // until indices cross
    std::swap(S[l], S[b]);                     // store pivot at l
    quickSortStep(S, a, l−1, less);            // recur on both sides
    quickSortStep(S, l+1, b, less);
}
```

Code Fragment 11.7: A coding of in-place quick-sort, assuming distinct elements.

The function quickSortStep is given indices *a* and *b*, which indicate the bounds of the subvector to be sorted. The pivot element is chosen to be the last element of the vector. The indices *l* and *r* mark the left and right ends of the subvectors being processed in the partitioning function. They are initialized to *a* and *b* − 1, respectively. During each round, elements that are on the wrong side of the pivot are swapped with each other, until these markers bump into each other.

Much of the efficiency of quick-sort depends on how the pivot is chosen. As we have seen, quick-sort is most efficient if the pivot is near the middle of the subvector being sorted. Our choice of setting the pivot to the last element of the subvector relies on the assumption that the last element is reflective of the median key valye. A better choice, if the subvector is moderately sized, is to select the pivot as the median of three values, taken respectively from the front, middle, and tail of the array. This is referred to as the *median-of-three* heuristic. It tends to perform well in practice, and is faster than selecting a random pivot through the use of a random-number generator.

11.3   Studying Sorting through an Algorithmic Lens

Recapping our discussions on sorting to this point, we have described several methods with either a worst-case or expected running time of $O(n \log n)$ on an input sequence of size n. These methods include merge-sort and quick-sort, described in this chapter, as well as heap-sort (Section 8.3.5). In this section, we study sorting as an algorithmic problem, addressing general issues about sorting algorithms.

11.3.1   A Lower Bound for Sorting

A natural first question to ask is whether we can sort any faster than $O(n \log n)$ time. Interestingly, if the computational primitive used by a sorting algorithm is the comparison of two elements, then this is, in fact, the best we can do—comparison-based sorting has an $\Omega(n \log n)$ worst-case lower bound on its running time. (Recall the notation $\Omega(\cdot)$ from Section 4.2.3.) To focus on the main cost of comparison-based sorting, let us only count comparisons, for the sake of a lower bound.

Suppose we are given a sequence $S = (x_0, x_1, \ldots, x_{n-1})$ that we wish to sort, and assume that all the elements of S are distinct (this is not really a restriction since we are deriving a lower bound). We do not care if S is implemented as an array or a linked list, for the sake of our lower bound, since we are only counting comparisons. Each time a sorting algorithm compares two elements x_i and x_j, that is, it asks, "is $x_i < x_j$?", there are two outcomes: "yes" or "no." Based on the result of this comparison, the sorting algorithm may perform some internal calculations (which we are not counting here) and eventually performs another comparison between two other elements of S, which again has two outcomes. Therefore, we can represent a comparison-based sorting algorithm with a decision tree T (recall Example 7.8). That is, each internal node v in T corresponds to a comparison and the edges from node v' to its children correspond to the computations resulting from either a "yes" or "no" answer. It is important to note that the hypothetical sorting algorithm in question probably has no explicit knowledge of the tree T. T simply represents all the possible sequences of comparisons that a sorting algorithm might make, starting from the first comparison (associated with the root) and ending with the last comparison (associated with the parent of an external node).

Each possible initial ordering, or *permutation*, of the elements in S causes our hypothetical sorting algorithm to execute a series of comparisons, traversing a path in T from the root to some external node. Let us associate with each external node v in T, then, the set of permutations of S that cause our sorting algorithm to end up in v. The most important observation in our lower-bound argument is that each external node v in T can represent the sequence of comparisons for at most one permutation of S. The justification for this claim is simple: if two different

permutations P_1 and P_2 of S are associated with the same external node, then there are at least two objects x_i and x_j, such that x_i is before x_j in P_1 but x_i is after x_j in P_2. At the same time, the output associated with v must be a specific reordering of S, with either x_i or x_j appearing before the other. But if P_1 and P_2 both cause the sorting algorithm to output the elements of S in this order, then that implies there is a way to trick the algorithm into outputting x_i and x_j in the wrong order. Since this cannot be allowed by a correct sorting algorithm, each external node of T must be associated with exactly one permutation of S. We use this property of the decision tree associated with a sorting algorithm to prove the following result.

Proposition 11.4: *The running time of any comparison-based algorithm for sorting an n-element sequence is $\Omega(n\log n)$ in the worst case.*

Justification: The running time of a comparison-based sorting algorithm must be greater than or equal to the height of the decision tree T associated with this algorithm, as described above. (See Figure 11.15.) By the argument above, each external node in T must be associated with one permutation of S. Moreover, each permutation of S must result in a different external node of T. The number of permutations of n objects is $n! = n(n-1)(n-2)\cdots 2 \cdot 1$. Thus, T must have at least $n!$ external nodes. By Proposition 7.10, the height of T is at least $\log(n!)$. This immediately justifies the proposition, because there are at least $n/2$ terms that are greater than or equal to $n/2$ in the product $n!$; hence

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\log\frac{n}{2},$$

which is $\Omega(n\log n)$. ∎
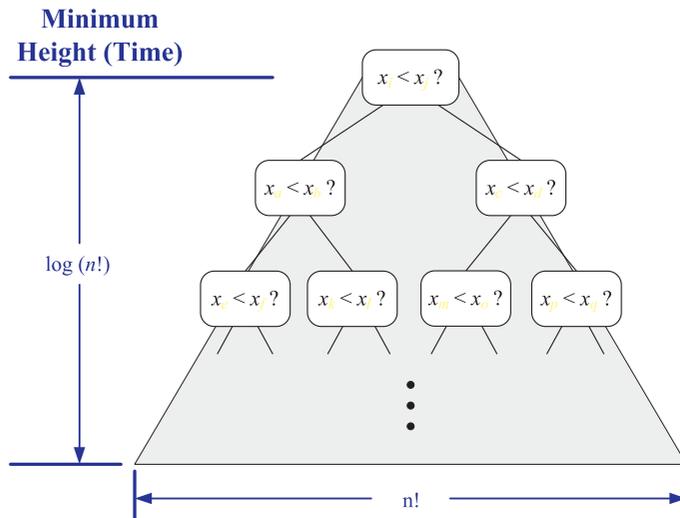
Figure 11.15: Visualizing the lower bound for comparison-based sorting.

11.3.2   Linear-Time Sorting: Bucket-Sort and Radix-Sort

In the previous section, we showed that $\Omega(n \log n)$ time is necessary, in the worst case, to sort an n-element sequence with a comparison-based sorting algorithm. A natural question to ask, then, is whether there are other kinds of sorting algorithms that can be designed to run asymptotically faster than $O(n \log n)$ time. Interestingly, such algorithms exist, but they require special assumptions about the input sequence to be sorted. Even so, such scenarios often arise in practice, so discussing them is worthwhile. In this section, we consider the problem of sorting a sequence of entries, each a key-value pair, where the keys have a restricted type.

Bucket-Sort

Consider a sequence S of n entries whose keys are integers in the range $[0, N-1]$, for some integer $N \geq 2$, and suppose that S should be sorted according to the keys of the entries. In this case, it is possible to sort S in $O(n+N)$ time. It might seem surprising, but this implies, for example, that if N is $O(n)$, then we can sort S in $O(n)$ time. Of course, the crucial point is that, because of the restrictive assumption about the format of the elements, we can avoid using comparisons.

The main idea is to use an algorithm called ***bucket-sort***, which is not based on comparisons, but on using keys as indices into a bucket array B that has cells indexed from 0 to $N-1$. An entry with key k is placed in the "bucket" $B[k]$, which itself is a sequence (of entries with key k). After inserting each entry of the input sequence S into its bucket, we can put the entries back into S in sorted order by enumerating the contents of the buckets $B[0], B[1], \ldots, B[N-1]$ in order. We describe the bucket-sort algorithm in Code Fragment 11.8.

Algorithm bucketSort(S):
    Input: Sequence S of entries with integer keys in the range $[0, N-1]$
    Output: Sequence S sorted in nondecreasing order of the keys
    let B be an array of N sequences, each of which is initially empty
    for each entry e in S **do**
        $k \leftarrow e.\mathsf{key}()$
        remove e from S and insert it at the end bucket (sequence) $B[k]$
    for $i \leftarrow 0$ to $N-1$ **do**
        for each entry e in sequence $B[i]$ **do**
            remove e from $B[i]$ and insert it at the end of S

Code Fragment 11.8: Bucket-sort.

It is easy to see that bucket-sort runs in $O(n+N)$ time and uses $O(n+N)$ space. Hence, bucket-sort is efficient when the range N of values for the keys is small compared to the sequence size n, say $N = O(n)$ or $N = O(n\log n)$. Still, its performance deteriorates as N grows compared to n.

An important property of the bucket-sort algorithm is that it works correctly even if there are many different elements with the same key. Indeed, we described it in a way that anticipates such occurrences.

Stable Sorting

When sorting key-value pairs, an important issue is how equal keys are handled. Let $S = ((k_0, x_0), \ldots, (k_{n-1}, x_{n-1}))$ be a sequence of such entries. We say that a sorting algorithm is *stable* if, for any two entries (k_i, x_i) and (k_j, x_j) of S, such that $k_i = k_j$ and (k_i, x_i) precedes (k_j, x_j) in S before sorting (that is, $i < j$), entry (k_i, x_i) also precedes entry (k_j, x_j) after sorting. Stability is important for a sorting algorithm because applications may want to preserve the initial ordering of elements with the same key.

Our informal description of bucket-sort in Code Fragment 11.8 does not guarantee stability. This is not inherent in the bucket-sort method itself, however, for we can easily modify our description to make bucket-sort stable, while still preserving its $O(n+N)$ running time. Indeed, we can obtain a stable bucket-sort algorithm by always removing the *first* element from sequence S and from the sequences $B[i]$ during the execution of the algorithm.

Radix-Sort

One of the reasons that stable sorting is so important is that it allows the bucket-sort approach to be applied to more general contexts than to sort integers. Suppose, for example, that we want to sort entries with keys that are pairs (k, l), where k and l are integers in the range $[0, N-1]$, for some integer $N \geq 2$. In a context such as this, it is natural to define an ordering on these keys using the *lexicographical* (dictionary) convention, where $(k_1, l_1) < (k_2, l_2)$ if $k_1 < k_2$ or if $k_1 = k_2$ and $l_1 < l_2$ (Section 8.1.2). This is a pair-wise version of the lexicographic comparison function, usually applied to equal-length character strings (and it easily generalizes to tuples of d numbers for $d > 2$).

The *radix-sort* algorithm sorts a sequence S of entries with keys that are pairs, by applying a stable bucket-sort on the sequence twice; first using one component of the pair as the ordering key and then using the second component. But which order is correct? Should we first sort on the k's (the first component) and then on the l's (the second component), or should it be the other way around?

Before we answer this question, we consider the following example.

Example 11.5: *Consider the following sequence S (we show only the keys):*

$$S = ((3,3),(1,5),(2,5),(1,2),(2,3),(1,7),(3,2),(2,2)).$$

If we sort S stably on the first component, then we get the sequence

$$S_1 = ((1,5),(1,2),(1,7),(2,5),(2,3),(2,2),(3,3),(3,2)).$$

If we then stably sort this sequence S_1 using the second component, then we get the sequence

$$S_{1,2} = ((1,2),(2,2),(3,2),(2,3),(3,3),(1,5),(2,5),(1,7)),$$

which is not exactly a sorted sequence. On the other hand, if we first stably sort S using the second component, then we get the sequence

$$S_2 = ((1,2),(3,2),(2,2),(3,3),(2,3),(1,5),(2,5),(1,7)).$$

If we then stably sort sequence S_2 using the first component, then we get the sequence

$$S_{2,1} = ((1,2),(1,5),(1,7),(2,2),(2,3),(2,5),(3,2),(3,3)),$$

which is indeed sequence S lexicographically ordered.

So, from this example, we are led to believe that we should first sort using the second component and then again using the first component. This intuition is exactly right. By first stably sorting by the second component and then again by the first component, we guarantee that if two entries are equal in the second sort (by the first component), then their relative order in the starting sequence (which is sorted by the second component) is preserved. Thus, the resulting sequence is guaranteed to be sorted lexicographically every time. We leave the determination of how this approach can be extended to triples and other d-tuples of numbers as a simple exercise (Exercise R-11.20). We can summarize this section as follows:

Proposition 11.6: *Let S be a sequence of n key-value pairs, each of which has a key $(k_1,k_2,\ldots,k_d)$, where k_i is an integer in the range $[0,N-1]$ for some integer $N \geq 2$. We can sort S lexicographically in time $O(d(n+N))$ using radix-sort.*

As important as it is, sorting is not the only interesting problem dealing with a total order relation on a set of elements. There are some applications, for example, that do not require an ordered listing of an entire set, but nevertheless call for some amount of ordering information about the set. Before we study such a problem (called "selection"), let us step back and briefly compare all of the sorting algorithms we have studied so far.

11.3.3   Comparing Sorting Algorithms

At this point, it might be useful for us to take a breath and consider all the algorithms we have studied in this book to sort an n-element vector, node list, or general sequence.

Considering Running Time and Other Factors

We have studied several methods, such as insertion-sort and selection-sort, that have $O(n^2)$-time behavior in the average and worst case. We have also studied several methods with $O(n \log n)$-time behavior, including heap-sort, merge-sort, and quick-sort. Finally, we have studied a special class of sorting algorithms, namely, the bucket-sort and radix-sort methods, that run in linear time for certain types of keys. Certainly, the selection-sort algorithm is a poor choice in any application, since it runs in $O(n^2)$ time even in the best case. But, of the remaining sorting algorithms, which is the best?

As with many things in life, there is no clear "best" sorting algorithm from the remaining candidates. The sorting algorithm best suited for a particular application depends on several properties of that application. We can offer some guidance and observations, therefore, based on the known properties of the "good" sorting algorithms.

Insertion-Sort

If implemented well, the running time of ***insertion-sort*** is $O(n + m)$, where m is the number of ***inversions*** (that is, the number of pairs of elements out of order). Thus, insertion-sort is an excellent algorithm for sorting small sequences (say, less than 50 elements), because insertion-sort is simple to program, and small sequences necessarily have few inversions. Also, insertion-sort is quite effective for sorting sequences that are already "almost" sorted. By "almost," we mean that the number of inversions is small. But the $O(n^2)$-time performance of insertion-sort makes it a poor choice outside of these special contexts.

Merge-Sort

Merge-sort, on the other hand, runs in $O(n \log n)$ time in the worst case, which is optimal for comparison-based sorting methods. Still, experimental studies have shown that, since it is difficult to make merge-sort run in-place, the overheads needed to implement merge-sort make it less attractive than the in-place implementations of heap-sort and quick-sort for sequences that can fit entirely in a computer's main memory area. Even so, merge-sort is an excellent algorithm for situations

where the input cannot all fit into main memory, but must be stored in blocks on an external memory device, such as a disk. In these contexts, the way that merge-sort processes runs of data in long merge streams makes the best use of all the data brought into main memory in a block from disk. Thus, for external memory sorting, the merge-sort algorithm tends to minimize the total number of disk reads and writes needed, which makes the merge-sort algorithm superior in such contexts.

Quick-Sort

Experimental studies have shown that if an input sequence can fit entirely in main memory, then the in-place versions of quick-sort and heap-sort run faster than merge-sort. The extra overhead needed for copying nodes or entries puts merge-sort at a disadvantage to quick-sort and heap-sort in these applications. In fact, quick-sort tends, on average, to beat heap-sort in these tests. So, *quick-sort* is an excellent choice as a general-purpose, in-memory sorting algorithm. Indeed, it is included in the qsort sorting utility provided in C language libraries. Still, its $O(n^2)$ time worst-case performance makes quick-sort a poor choice in real-time applications where we must make guarantees on the time needed to complete a sorting operation.

Heap-Sort

In real-time scenarios where we have a fixed amount of time to perform a sorting operation and the input data can fit into main memory, the *heap-sort* algorithm is probably the best choice. It runs in $O(n \log n)$ worst-case time and can easily be made to execute in-place.

Bucket-Sort and Radix-Sort

Finally, if our application involves sorting entries with small integer keys or d-tuples of small integer keys, then *bucket-sort* or *radix-sort* is an excellent choice, because it runs in $O(d(n+N))$ time, where $[0, N-1]$ is the range of integer keys (and $d = 1$ for bucket sort). Thus, if $d(n+N)$ is significantly "below" the $n \log n$ function, then this sorting method should run faster than even quick-sort or heap-sort.

Thus, our study of all these different sorting algorithms provides us with a versatile collection of sorting methods in our algorithm engineering "toolbox."

11.4 Sets and Union/Find Structures

In this section, we study sets, including operations that define them and operations that can be applied to entire sets.

11.4.1 The Set ADT

A *set* is a collection of distinct objects. That is, there are no duplicate elements in a set, and there is no explicit notion of keys or even an order. Even so, if the elements in a set are comparable, then we can maintain sets to be ordered. The fundamental functions of the set ADT for a set S are the following:

insert(e): Insert the element e into S and return an iterator referring to its location; if the element already exists the operation is ignored.

find(e): If S contains e, return an iterator p referring to this entry, else return end.

erase(e): Remove the element e from S.

begin(): Return an iterator to the beginning of S.

end(): Return an iterator to an imaginary position just beyond the end of S.

The C++ Standard Template Library provides a class set that contains all of these functions. It actually implements an ordered set, and supports the following additional operations as well.

lower_bound(e): Return an iterator to the largest element less than or equal to e.

upper_bound(e): Return an iterator to the smallest element greater than or equal to e.

equal_range(e): Return an iterator range of elements that are equal to e.

The STL set is templated with the element type. As with the other STL classes we have seen so far, the set is an example of a container, and hence supports access by iterators. In order to declare an object of type set, it is necessary to first include the definition file called "set." The set is part of the std namespace, and hence it is necessary either to use "std::set" or to provide an appropriate "**using**" statement.

The STL set is implemented by adapting the STL ordered map (which is based on a red-black tree). Each entry has the property that the key and element are both equal to e. That is, each entry is of the form (e, e).

11.4.2   Mergable Sets and the Template Method Pattern

Let us explore a further extension of the ordered set ADT that allows for operations between pairs of sets. This also serves to motivate a software engineering design pattern known as the **template method**.

First, we recall the mathematical definitions of the **union**, **intersection**, and **subtraction** of two sets A and B:

$$
\begin{aligned}
A \cup B &= \{x\colon x \text{ is in } A \text{ or } x \text{ is in } B\}, \\
A \cap B &= \{x\colon x \text{ is in } A \text{ and } x \text{ is in } B\}, \\
A - B &= \{x\colon x \text{ is in } A \text{ and } x \text{ is not in } B\}.
\end{aligned}
$$

Example 11.7: *Most Internet search engines store, for each word x in their dictionary database, a set, $W(x)$, of Web pages that contain x, where each Web page is identified by a unique Internet address. When presented with a query for a word x, such a search engine need only return the Web pages in the set $W(x)$, sorted according to some proprietary priority ranking of page "importance." But when presented with a two-word query for words x and y, such a search engine must first compute the intersection $W(x) \cap W(y)$, and then return the Web pages in the resulting set sorted by priority. Several search engines use the set intersection algorithm described in this section for this computation.*

Fundamental Methods of the Mergable Set ADT

The fundamental functions of the mergable set ADT, acting on a set A, are as follows:

union(B): Replace A with the union of A and B, that is, execute $A \leftarrow A \cup B$.

intersect(B): Replace A with the intersection of A and B, that is, execute $A \leftarrow A \cap B$.

subtract(B): Replace A with the difference of A and B, that is, execute $A \leftarrow A - B$.

A Simple Mergable Set Implementation

One of the simplest ways of implementing a set is to store its elements in an ordered sequence. This implementation is included in several software libraries for generic data structures, for example. Therefore, let us consider implementing the set ADT with an ordered sequence (we consider other implementations in several exercises). Any consistent total order relation among the elements of the set can be used, provided the same order is used for all the sets.

We implement each of the three fundamental set operations using a generic version of the merge algorithm that takes, as input, two sorted sequences representing the input sets, and constructs a sequence representing the output set, be it the union, intersection, or subtraction of the input sets. Incidentally, we have defined these operations so that they modify the contents of the set A involved. Alternatively, we could have defined these functions so that they do not modify A but return a new set instead.

The generic merge algorithm iteratively examines and compares the current elements a and b of the input sequence A and B, respectively, and finds out whether $a < b$, $a = b$, or $a > b$. Then, based on the outcome of this comparison, it determines whether it should copy one of the elements a and b to the end of the output sequence C. This determination is made based on the particular operation we are performing, be it a union, intersection, or subtraction. For example, in a union operation, we proceed as follows:

- If $a < b$, we copy a to the end of C and advance to the next element of A
- If $a = b$, we copy a to the end of C and advance to the next elements of A and B
- If $a > b$, we copy b to the end of C and advance to the next element of B

Performance of Generic Merging

Let us analyze the running time of generic merging. At each iteration, we compare two elements of the input sequences A and B, possibly copy one element to the output sequence, and advance the current element of A, B, or both. Assuming that comparing and copying elements takes $O(1)$ time, the total running time is $O(n_A + n_B)$, where n_A is the size of A and n_B is the size of B; that is, generic merging takes time proportional to the number of elements. Thus, we have the following:

Proposition 11.8: *The set ADT can be implemented with an ordered sequence and a generic merge scheme that supports operations* union, intersect, *and* subtract *in $O(n)$ time, where n denotes the sum of sizes of the sets involved.*

Generic Merging as a Template Method Pattern

The generic merge algorithm is based on the ***template method pattern*** (see Section 7.3.7). The template method pattern is a software engineering design pattern describing a generic computation mechanism that can be specialized by redefining certain steps. In this case, we describe a method that merges two sequences into one and can be specialized by the behavior of three abstract methods.

Code Fragments 11.9 and 11.10 show the class Merge providing a C++ implementation of the generic merge algorithm. This class has no data members. It defines a public function merge, which merges the two lists A and B, and stores the

result in C. It provides three virtual functions, fromA, fromB, and fromBoth. These are pure virtual functions (that is, they are not defined here), but are overridden in subclasses of Merge, to achieve a desired effect. The function fromA specifies the action to be taken when the next element to be selected in the merger is from A. Similarly, fromB specifies the action when the next element to be selected is from B. Finally, fromBoth is the action to be taken when the two elements of A and B are equal, and hence both are to be selected.

```
template <typename E>
class Merge {                                    // generic Merge
public:                                          // global types
  typedef std::list<E> List;                     // list type
  void merge(List& A, List& B, List& C);         // generic merge function
protected:                                       // local types
  typedef typename List::iterator Itor;          // iterator type
                                                 // overridden functions
  virtual void fromA(const E& a, List& C) = 0;
  virtual void fromBoth(const E& a, const E& b, List& C) = 0;
  virtual void fromB(const E& b, List& C) = 0;
};
```

      Code Fragment 11.9: Definition of the class Merge for generic merging.

    The function merge, which is presented in Code Fragment 11.10 performs the actual merger. It is structurally similar to the list-based merge procedure given in Code Fragment 11.3. Rather than simply taking an element from list A or list B, it invokes one of the virtual functions to perform the appropriate specialized task. The final result is stored in the list C.

```
template <typename E>                            // generic merge
void Merge<E>::merge(List& A, List& B, List& C) {
  Itor pa = A.begin();                           // A's elements
  Itor pb = B.begin();                           // B's elements
  while (pa != A.end() && pb != B.end()) {       // main merging loop
    if (*pa < *pb)
      fromA(*pa++, C);                           // take from A
    else if (*pa == *pb)
      fromBoth(*pa++, *pb++, C);                 // take from both
    else
      fromB(*pb++, C);                           // take from B
  }
  while (pa != A.end()) { fromA(*pa++, C); }     // take rest from A
  while (pb != B.end()) { fromB(*pb++, C); }     // take rest from B
}
```

Code Fragment 11.10: Member function merge which implements generic merging for class Merge.

To convert Merge into a useful class, we provide definitions for the three auxiliary functions, fromA, fromBoth, and fromB. See Code 11.11.

- In class UnionMerge, merge copies every element from *A* and *B* into *C*, but does not duplicate any element.
- In class IntersectMerge, merge copies every element that is in both *A* and *B* into *C*, but "throws away" elements in one set but not in the other.
- In class SubtractMerge, merge copies every element that is in *A* and not in *B* into *C*.

```
template <typename E>                         // set union
class UnionMerge : public Merge<E> {
protected:
  typedef typename Merge<E>::List List;
  virtual void fromA(const E& a, List& C)
    { C.push_back(a); }                       // add a
  virtual void fromBoth(const E& a, const E& b, List& C)
    { C.push_back(a); }                       // add a only
  virtual void fromB(const E& b, List& C)
    { C.push_back(b); }                       // add b
};

template <typename E>                         // set intersection
class IntersectMerge : public Merge<E> {
protected:
  typedef typename Merge<E>::List List;
  virtual void fromA(const E& a, List& C)
    { }                                       // ignore
  virtual void fromBoth(const E& a, const E& b, List& C)
    { C.push_back(a); }                       // add a only
  virtual void fromB(const E& b, List& C)
    { }                                       // ignore
};

template <typename E>                         // set subtraction
class SubtractMerge : public Merge<E> {
protected:
  typedef typename Merge<E>::List List;
  virtual void fromA(const E& a, List& C)
    { C.push_back(a); }                       // add a
  virtual void fromBoth(const E& a, const E& b, List& C)
    { }                                       // ignore
  virtual void fromB(const E& b, List& C)
    { }                                       // ignore
};
```

Code Fragment 11.11: Classes extending the Merge class by specializing the auxiliary functions to perform set union, intersection, and subtraction, respectively.

11.4.3   Partitions with Union-Find Operations

A ***partition*** is a collection of disjoint sets. We define the functions of the partition ADT using position objects (Section 6.2.1), each of which stores an element x. The partition ADT supports the following functions.

> makeSet(x):   Create a singleton set containing the element x and return the position storing x in this set.
>
> union(A, B):   Return the set $A \cup B$, destroying the old A and B.
>
> find(p):   Return the set containing the element in position p.

A simple implementation of a partition with a total of n elements is using a collection of sequences, one for each set, where the sequence for a set A stores set positions as its elements. Each position object stores a variable, *element*, which references its associated element x and allows the execution of the element() function in $O(1)$ time. In addition, we also store a variable, *set*, in each position, which references the sequence storing p, since this sequence is representing the set containing p's element. (See Figure 11.16.) Thus, we can perform operation find(p) in $O(1)$ time, by following the *set* reference for p. Likewise, makeSet also takes $O(1)$ time. Operation union(A, B) requires that we join two sequences into one and update the *set* references of the positions in one of the two. We choose to implement this operation by removing all the positions from the sequence with smaller size, and inserting them in the sequence with larger size. Each time we take a position p from the smaller set s and insert it into the larger set t, we update the *set* reference for p to now point to t. Hence, the operation union(A, B) takes time $O(\min(|A|, |B|))$, which is $O(n)$, because, in the worst case, $|A| = |B| = n/2$. Nevertheless, as shown below, an amortized analysis shows this implementation to be much better than appears from this worst-case analysis.
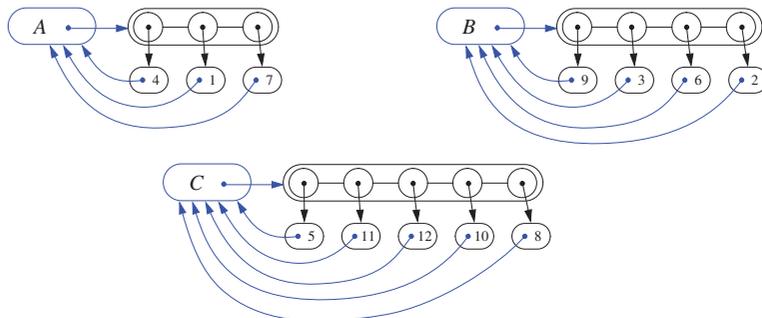
Figure 11.16: Sequence-based implementation of a partition consisting of three sets: $A = \{1, 4, 7\}$, $B = \{2, 3, 6, 9\}$, and $C = \{5, 8, 10, 11, 12\}$.

Performance of the Sequence Implementation

The sequence implementation above is simple, but it is also efficient, as the following theorem shows.

Proposition 11.9: *Performing a series of n makeSet, union, and find operations, using the sequence-based implementation above, starting from an initially empty partition takes $O(n \log n)$ time.*

Justification: We use the accounting method and assume that one cyber-dollar can pay for the time to perform a find operation, a makeSet operation, or the movement of a position object from one sequence to another in a union operation. In the case of a find or makeSet operation, we charge the operation itself 1 cyber-dollar. In the case of a union operation, however, we charge 1 cyber-dollar to each position that we move from one set to another. Note that we charge nothing to the union operations themselves. Clearly, the total charges to find and makeSet operations add up to $O(n)$.

Consider, then, the number of charges made to positions on behalf of union operations. The important observation is that each time we move a position from one set to another, the size of the new set at least doubles. Thus, each position is moved from one set to another at most $\log n$ times; hence, each position can be charged at most $O(\log n)$ times. Since we assume that the partition is initially empty, there are $O(n)$ different elements referenced in the given series of operations, which implies that the total time for all the union operations is $O(n \log n)$. ∎

The amortized running time of an operation in a series of makeSet, union, and find operations, is the total time taken for the series divided by the number of operations. We conclude from the proposition above that, for a partition implemented using sequences, the amortized running time of each operation is $O(\log n)$. Thus, we can summarize the performance of our simple sequence-based partition implementation as follows.

Proposition 11.10: *Using a sequence-based implementation of a partition, in a series of n makeSet, union, and find operations starting from an initially empty partition, the amortized running time of each operation is $O(\log n)$.*

Note that in this sequence-based implementation of a partition, each find operation takes worst-case $O(1)$ time. It is the running time of the union operations that is the computational bottleneck.

In the next section, we describe a tree-based implementation of a partition that does not guarantee constant-time find operations, but has amortized time much better than $O(\log n)$ per union operation.

A Tree-Based Partition Implementation ⋆

An alternative data structure uses a collection of trees to store the n elements in sets, where each tree is associated with a different set. (See Figure 11.17.) In particular, we implement each tree with a linked data structure whose nodes are themselves the set position objects. We still view each position p as being a node having a variable, *element*, referring to its element x, and a variable, *set*, referring to a set containing x, as before. But now we also view each position p as being of the "*set*" data type. Thus, the *set* reference of each position p can point to a position, which could even be p itself. Moreover, we implement this approach so that all the positions and their respective *set* references together define a collection of trees.

We associate each tree with a set. For any position p, if p's *set* reference points back to p, then p is the ***root*** of its tree, and the name of the set containing p is "p" (that is, we use position names as set names in this case). Otherwise, the *set* reference for p points to p's parent in its tree. In either case, the set containing p is the one associated with the root of the tree containing p.
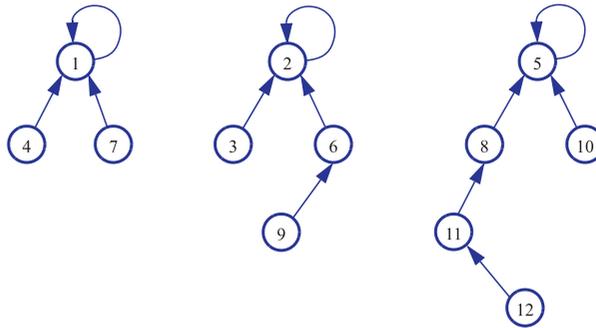
Figure 11.17: Tree-based implementation of a partition consisting of three disjoint sets: $A = \{1, 4, 7\}$, $B = \{2, 3, 6, 9\}$, and $C = \{5, 8, 10, 11, 12\}$.

With this partition data structure, operation union(A, B) is called with position arguments p and q that respectively represent the sets A and B (that is, $A = p$ and $B = q$). We perform this operation by making one of the trees a subtree of the other (Figure 11.18b), which can be done in $O(1)$ time by setting the *set* reference of the root of one tree to point to the root of the other tree. Operation find for a position p is performed by walking up to the root of the tree containing the position p (Figure 11.18a), which takes $O(n)$ time in the worst case.

At first, this implementation may seem to be no better than the sequence-based data structure, but we add the following two simple heuristics to make it run faster.

Union-by-Size: Store, with each position node p, the size of the subtree rooted at p. In a union operation, make the tree of the smaller set become a subtree of the other tree, and update the size field of the root of the resulting tree.
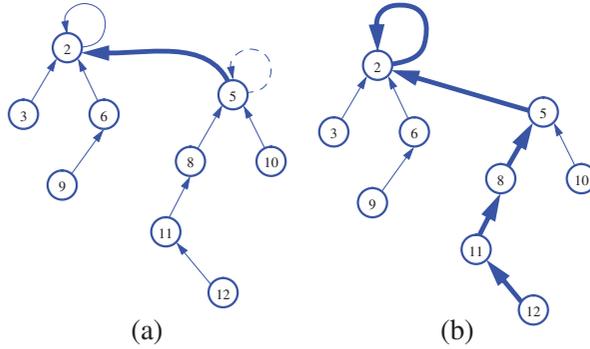
Figure 11.18: Tree-based implementation of a partition: (a) operation union(A,B); (b) operation find(p), where p denotes the position object for element 12.

Path Compression: In a find operation, for each node v that the find visits, reset the parent pointer from v to point to the root. (See Figure 11.19.)
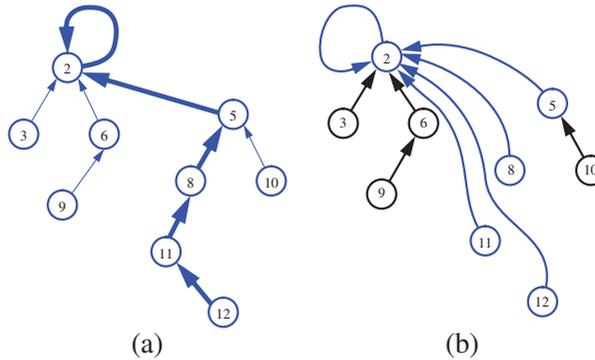
Figure 11.19: Path-compression heuristic: (a) path traversed by operation find on element 12; (b) restructured tree.

A surprising property of this data structure, when implemented using the union-by-size and path-compression heuristics, is that performing a series of n union and find operations takes $O(n \log^* n)$ time, where $\log^* n$ is the **log-star** function, which is the inverse of the **tower-of-twos** function. Intuitively, $\log^* n$ is the number of times that one can iteratively take the logarithm (base 2) of a number before getting a number smaller than 2. Table 11.1 shows a few sample values.

Minimum n	2	$2^2 = 4$	$2^{2^2} = 16$	$2^{2^{2^2}} = 65,536$	$2^{2^{2^{2^2}}} = 2^{65,536}$
$\log^* n$	1	2	3	4	5

Table 11.1: Some values of $\log^* n$ and critical values for its inverse.

11.5   Selection

There are a number of applications in which we are interested in identifying a single element in terms of its rank relative to an ordering of the entire set. Examples include identifying the minimum and maximum elements, but we may also be interested in, say, identifying the *median* element, that is, the element such that half of the other elements are smaller and the remaining half are larger. In general, queries that ask for an element with a given rank are called *order statistics*.

Defining the Selection Problem

In this section, we discuss the general order-statistic problem of selecting the kth smallest element from an unsorted collection of n comparable elements. This is known as the *selection* problem. Of course, we can solve this problem by sorting the collection and then indexing into the sorted sequence at index $k - 1$. Using the best comparison-based sorting algorithms, this approach would take $O(n \log n)$ time, which is obviously an overkill for the cases where $k = 1$ or $k = n$ (or even $k = 2$, $k = 3$, $k = n - 1$, or $k = n - 5$), because we can easily solve the selection problem for these values of k in $O(n)$ time. Thus, a natural question to ask is whether we can achieve an $O(n)$ running time for all values of k (including the interesting case of finding the median, where $k = \lfloor n/2 \rfloor$).

11.5.1   Prune-and-Search

This may come as a small surprise, but we can indeed solve the selection problem in $O(n)$ time for any value of k. Moreover, the technique we use to achieve this result involves an interesting algorithmic design pattern. This design pattern is known as *prune-and-search* or *decrease-and-conquer*. In applying this design pattern, we solve a given problem that is defined on a collection of n objects by pruning away a fraction of the n objects and recursively solving the smaller problem. When we have finally reduced the problem to one defined on a constant-sized collection of objects, then we solve the problem using some brute-force method. Returning back from all the recursive calls completes the construction. In some cases, we can avoid using recursion, in which case we simply iterate the prune-and-search reduction step until we can apply a brute-force method and stop. Incidentally, the binary search method described in Section 9.3.1 is an example of the prune-and-search design pattern.

11.5.2 Randomized Quick-Select

In applying the prune-and-search pattern to the selection problem, we can design a simple and practical method, called ***randomized quick-select***, for finding the kth smallest element in an unordered sequence of n elements on which a total order relation is defined. Randomized quick-select runs in $O(n)$ ***expected*** time, taken over all possible random choices made by the algorithm. This expectation does not depend whatsoever on any randomness assumptions about the input distribution. We note though that randomized quick-select runs in $O(n^2)$ time in the ***worst case***. The justification of this is left as an exercise (Exercise R-11.26). We also provide an exercise (Exercise C-11.32) for modifying randomized quick-select to get a ***deterministic*** selection algorithm that runs in $O(n)$ ***worst-case*** time. The existence of this deterministic algorithm is mostly of theoretical interest, however, since the constant factor hidden by the big-Oh notation is relatively large in this case.

Suppose we are given an unsorted sequence S of n comparable elements together with an integer $k \in [1, n]$. At a high level, the quick-select algorithm for finding the kth smallest element in S is similar in structure to the randomized quick-sort algorithm described in Section 11.2.1. We pick an element x from S at random and use this as a "pivot" to subdivide S into three subsequences L, E, and G, storing the elements of S less than x, equal to x, and greater than x, respectively. This is the prune step. Then, based on the value of k, we determine which of these sets to recur on. Randomized quick-select is described in Code Fragment 11.12.

Algorithm quickSelect(S, k):
    Input: Sequence S of n comparable elements, and an integer $k \in [1, n]$
    Output: The kth smallest element of S
    if $n = 1$ **then**
        return the (first) element of S.
    pick a random (pivot) element x of S and divide S into three sequences:
        • L, storing the elements in S less than x
        • E, storing the elements in S equal to x
        • G, storing the elements in S greater than x.
    if $k \leq |L|$ **then**
        quickSelect(L, k)
    else if $k \leq |L| + |E|$ **then**
        return x {each element in E is equal to x}
    else
        quickSelect($G, k - |L| - |E|$)   {note the new selection parameter}

Code Fragment 11.12: Randomized quick-select algorithm.

11.5.3 Analyzing Randomized Quick-Select

Showing that randomized quick-select runs in $O(n)$ time requires a simple probabilistic argument. The argument is based on the *linearity of expectation*, which states that if X and Y are random variables and c is a number, then

$$E(X+Y) = E(X) + E(Y) \qquad \text{and} \qquad E(cX) = cE(X),$$

where we use $E(\mathcal{Z})$ to denote the expected value of the expression $\mathcal{Z}$.

Let $t(n)$ be the running time of randomized quick-select on a sequence of size n. Since this algorithm depends on random events, its running time, $t(n)$, is a random variable. We want to bound $E(t(n))$, the expected value of $t(n)$. Say that a recursive invocation of our algorithm is "good" if it partitions S so that the size of L and G is at most $3n/4$. Clearly, a recursive call is good with probability $1/2$. Let $g(n)$ denote the number of consecutive recursive calls we make, including the present one, before we get a good one. Then we can characterize $t(n)$ using the following *recurrence equation*

$$t(n) \le bn \cdot g(n) + t(3n/4),$$

where $b \ge 1$ is a constant. Applying the linearity of expectation for $n > 1$, we get

$$E(t(n)) \le E(bn \cdot g(n) + t(3n/4)) = bn \cdot E(g(n)) + E(t(3n/4)).$$

Since a recursive call is good with probability $1/2$, and whether a recursive call is good or not is independent on its parent call being good, the expected value of $g(n)$ is the same as the expected number of times we must flip a fair coin before it comes up "heads." That is, $E(g(n)) = 2$. Thus, if we let $T(n)$ be shorthand for $E(t(n))$, then we can write the case for $n > 1$ as

$$T(n) \le T(3n/4) + 2bn.$$

To convert this relation into a closed form, let us iteratively apply this inequality assuming n is large. So, for example, after two applications,

$$T(n) \le T((3/4)^2 n) + 2b(3/4)n + 2bn.$$

At this point, we should see that the general case is

$$T(n) \le 2bn \cdot \sum_{i=0}^{\lceil \log_{4/3} n \rceil} (3/4)^i.$$

In other words, the expected running time is at most $2bn$ times a geometric sum whose base is a positive number less than 1. Thus, by Proposition 4.5, $T(n)$ is $O(n)$.

Proposition 11.11: *The expected running time of randomized quick-select on a sequence S of size n is $O(n)$, assuming two elements of S can be compared in $O(1)$ time.*

11.6 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-11.1 What is the best algorithm for sorting each of the following: general comparable objects, long character strings, double-precision floating point numbers, 32-bit integers, and bytes? Justify your answer.

R-11.2 Suppose S is a list of n bits, that is, n 0's and 1's. How long will it take to sort S with the merge-sort algorithm? What about quick-sort?

R-11.3 Suppose S is a list of n bits, that is, n 0's and 1's. How long will it take to sort S stably with the bucket-sort algorithm?

R-11.4 Give a complete justification of Proposition 11.1.

R-11.5 In the merge-sort tree shown in Figures 11.2 through 11.4, some edges are drawn as arrows. What is the meaning of a downward arrow? How about an upward arrow?

R-11.6 Give a complete pseudo-code description of the recursive merge-sort algorithm that takes an array as its input and output.

R-11.7 Show that the running time of the merge-sort algorithm on an n-element sequence is $O(n \log n)$, even when n is not a power of 2.

R-11.8 Suppose we are given two n-element sorted sequences A and B that should not be viewed as sets (that is, A and B may contain duplicate entries). Describe an $O(n)$-time method for computing a sequence representing the set $A \cup B$ (with no duplicates).

R-11.9 Show that $(X - A) \cup (X - B) = X - (A \cap B)$, for any three sets X, A, and B.

R-11.10 Suppose we modify the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an n-element sequence as the pivot, we choose the element at index $\lfloor n/2 \rfloor$. What is the running time of this version of quick-sort on a sequence that is already sorted?

R-11.11 Consider a modification of the deterministic version of the quick-sort algorithm where we choose the element at index $\lfloor n/2 \rfloor$ as our pivot. Describe the kind of sequence that would cause this version of quick-sort to run in $\Omega(n^2)$ time.

R-11.12 Show that the best-case running time of quick-sort on a sequence of size n with distinct elements is $O(n \log n)$.

R-11.13 Describe a randomized version of in-place quick-sort in pseudo-code.

R-11.14 Show that the probability that any given input element x belongs to more than $2 \log n$ subproblems in size group i, for randomized quick-sort, is at most $1/n^2$.

R-11.15 Suppose algorithm inPlaceQuickSort (Code Fragment 11.6) is executed on a sequence with duplicate elements. Show that the algorithm still correctly sorts the input sequence, but the result of the divide step may differ from the high-level description given in Section 11.2, and may result in inefficiencies. In particular, what happens in the partition step when there are elements equal to the pivot? Is the sequence E (storing the elements equal to the pivot) actually computed? Does the algorithm recur on the subsequences L and G, or on some other subsequences? What is the running time of the algorithm if all the input elements are equal?

R-11.16 Of the $n!$ possible inputs to a given comparison-based sorting algorithm, what is the absolute maximum number of inputs that could be sorted with just n comparisons?

R-11.17 Bella has a comparison-based sorting algorithm that sorts the first k elements in sequence of size n in $O(n)$ time. Give a big-Oh characterization of the biggest that k can be?

R-11.18 Is the merge-sort algorithm in Section 11.1 stable? Why or why not?

R-11.19 An algorithm that sorts key-value entries by key is said to be ***straggling*** if, any time two entries e_i and e_j have equal keys, but e_i appears before e_j in the input, then the algorithm places e_i after e_j in the output. Describe a change to the merge-sort algorithm in Section 11.1 to make it straggling.

R-11.20 Describe a radix-sort method for lexicographically sorting a sequence S of triplets (k, l, m), where k, l, and m are integers in the range $[0, N-1]$, for some $N \geq 2$. How could this scheme be extended to sequences of d-tuples $(k_1, k_2, \ldots, k_d)$, where each k_i is an integer in the range $[0, N-1]$?

R-11.21 Is the bucket-sort algorithm in-place? Why or why not?

R-11.22 Give an example input list that requires merge-sort and heap-sort to take $O(n \log n)$ time to sort, but insertion-sort runs in $O(n)$ time. What if you reverse this list?

R-11.23 Describe, in pseudo-code, how to perform path compression on a path of length h in $O(h)$ time in a tree-based partition union/find structure.

R-11.24 Edward claims he has a fast way to do path compression in a partition structure, starting at a node v. He puts v into a list L, and starts following parent pointers. Each time he encounters a new node, u, he adds u to L and updates the parent pointer of each node in L to point to u's parent. Show that Edward's algorithm runs in $\Omega(h^2)$ time on a path of length h.

R-11.25 Describe an in-place version of the quick-select algorithm in pseudo-code.

R-11.26 Show that the worst-case running time of quick-select on an n-element sequence is $\Omega(n^2)$.

Creativity

C-11.1 Describe an efficient algorithm for converting a dictionary, D, implemented with a linked list, into a map, M, implemented with a linked list, so that each key in D has an entry in M, and the relative order of entries in M is the same as their relative order in D.

C-11.2 Linda claims to have an algorithm that takes an input sequence S and produces an output sequence T that is a sorting of the n elements in S.
   a. Give an algorithm, isSorted, for testing in $O(n)$ time if T is sorted.
   b. Explain why the algorithm isSorted is not sufficient to prove a particular output T of Linda's algorithm is a sorting of S.
   c. Describe what additional information Linda's algorithm could output so that her algorithm's correctness could be established on any given S and T in $O(n)$ time.

C-11.3 Given two sets A and B represented as sorted sequences, describe an efficient algorithm for computing $A \oplus B$, which is the set of elements that are in A or B, but not in both.

C-11.4 Suppose that we represent sets with balanced search trees. Describe and analyze algorithms for each of the functions in the set ADT, assuming that one of the two sets is much smaller than the other.

C-11.5 Describe and analyze an efficient function for removing all duplicates from a collection A of n elements.

C-11.6 Consider sets whose elements are integers in the range $[0, N-1]$. A popular scheme for representing a set A of this type is by means of a Boolean array, B, where we say that x is in A if and only if $B[x] = $ **true**. Since each cell of B can be represented with a single bit, B is sometimes referred to as a ***bit vector***. Describe and analyze efficient algorithms for performing the functions of the set ADT assuming this representation.

C-11.7 Consider a version of deterministic quick-sort where we pick the median of the d last elements in the input sequence of n elements as our pivot, for a fixed, constant odd number $d \geq 3$. What is the asymptotic worst-case running time of quick-sort in this case?

C-11.8 Another way to analyze randomized quick-sort is to use a ***recurrence equation***. In this case, we let $T(n)$ denote the expected running time of randomized quick-sort, and we observe that, because of the worst-case partitions for good and bad splits, we can write

$$T(n) \leq \frac{1}{2}\left(T(3n/4) + T(n/4)\right) + \frac{1}{2}\left(T(n-1)\right) + bn,$$

where bn is the time needed to partition a list for a given pivot and concatenate the result sublists after the recursive calls return. Show, by induction, that $T(n)$ is $O(n \log n)$.

C-11.9 Modify inPlaceQuickSort (Code Fragment 11.6) to handle the general case efficiently when the input sequence, S, may have duplicate keys.

C-11.10 Describe a nonrecursive, in-place version of the quick-sort algorithm. The algorithm should still be based on the same divide-and-conquer approach, but use an explicit stack to process subproblems.

C-11.11 An ***inverted file*** is a critical data structure for implementing a search engine or the index of a book. Given a document D, which can be viewed as an unordered, numbered list of words, an inverted file is an ordered list of words, L, such that, for each w word in L, we store the indices of the places in D where w appears. Design an efficient algorithm for constructing L from D.

C-11.12 Given an array A of n entries with keys equal to 0 or 1, describe an in-place function for ordering A so that all the 0's are before every 1.

C-11.13 Suppose we are given an n-element sequence S such that each element in S represents a different vote for president, where each vote is given as an integer representing a particular candidate. Design an $O(n \log n)$-time algorithm to see who wins the election S represents, assuming the candidate with the most votes wins (even if there are $O(n)$ candidates).

C-11.14 Consider the voting problem from Exercise C-11.13, but now suppose that we know the number $k < n$ of candidates running. Describe an $O(n \log k)$-time algorithm for determining who wins the election.

C-11.15 Consider the voting problem from Exercise C-11.13, but now suppose a candidate wins only if he or she gets a majority of the votes cast. Design and analyze a fast algorithm for determining the winner if there is one.

C-11.16 Show that any comparison-based sorting algorithm can be made to be stable without affecting its asymptotic running time.

C-11.17 Suppose we are given two sequences A and B of n elements, possibly containing duplicates, on which a total order relation is defined. Describe an efficient algorithm for determining if A and B contain the same set of elements. What is the running time of this method?

C-11.18 Given an array A of n integers in the range $[0, n^2 - 1]$, describe a simple function for sorting A in $O(n)$ time.

C-11.19 Let $S_1, S_2, \ldots, S_k$ be k different sequences whose elements have integer keys in the range $[0, N - 1]$, for some parameter $N \geq 2$. Describe an algorithm running in $O(n + N)$ time for sorting all the sequences (not as a union), where n denotes the total size of all the sequences.

C-11.20 Given a sequence S of n elements, on which a total order relation is defined, describe an efficient function for determining whether there are two equal elements in S. What is the running time of your function?

C-11.21 Let S be a sequence of n elements on which a total order relation is defined. Recall that an **inversion** in S is a pair of elements x and y such that x appears before y in S but $x > y$. Describe an algorithm running in $O(n \log n)$ time for determining the **number** of inversions in S.

C-11.22 Let S be a random permutation of n distinct integers. Argue that the expected running time of insertion-sort on S is $\Omega(n^2)$.
(Hint: Note that half of the elements ranked in the top half of a sorted version of S are expected to be in the first half of S.)

C-11.23 Let A and B be two sequences of n integers each. Given an integer m, describe an $O(n \log n)$-time algorithm for determining if there is an integer a in A and an integer b in B such that $m = a + b$.

C-11.24 Given a set of n integers, describe and analyze a fast method for finding the $\lceil \log n \rceil$ integers closest to the median.

C-11.25 James has a set A of n nuts and a set B of n bolts, such that each nut in A has a unique matching bolt in B. Unfortunately, the nuts in A all look the same, and the bolts in B all look the same as well. The only kind of a comparison that Bob can make is to take a nut-bolt pair (a,b), such that a is in A and b is in B, and test it to see if the threads of a are larger, smaller, or a perfect match with the threads of b. Describe and analyze an efficient algorithm for Bob to match up all of his nuts and bolts.

C-11.26 Show how to use a deterministic $O(n)$-time selection algorithm to sort a sequence of n elements in $O(n \log n)$ **worst-case** time.

C-11.27 Given an unsorted sequence S of n comparable elements, and an integer k, give an $O(n \log k)$ expected-time algorithm for finding the $O(k)$ elements that have rank $\lceil n/k \rceil$, $2\lceil n/k \rceil$, $3\lceil n/k \rceil$, and so on.

C-11.28 Let S be a sequence of n insert and removeMin operations, where all the keys involved are integers in the range $[0, n-1]$. Describe an algorithm running in $O(n \log^* n)$ for determining the answer to each removeMin.

C-11.29 Space aliens have given us a program, alienSplit, that can take a sequence S of n integers and partition S in $O(n)$ time into sequences $S_1, S_2, \ldots, S_k$ of size at most $\lceil n/k \rceil$ each, such that the elements in S_i are less than or equal to every element in S_{i+1}, for $i = 1, 2, \ldots, k-1$, for a fixed number, $k < n$. Show how to use alienSplit to sort S in $O(n \log n / \log k)$ time.

C-11.30 Karen has a new way to do path compression in a tree-based union/find partition data structure starting at a node v. She puts all the nodes that are on the path from v to the root in a set S. Then she scans through S and sets the parent pointer of each node in S to its parent's parent pointer (recall

that the parent pointer of the root points to itself). If this pass changed the value of any node's parent pointer, then she repeats this process, and goes on repeating this process until she makes a scan through S that does not change any node's parent value. Show that Karen's algorithm is correct and analyze its running time for a path of length h.

C-11.31 Let S be a sequence of n integers. Describe a method for printing out all the pairs of inversions in S in $O(n+k)$ time, where k is the number of such inversions.

C-11.32 This problem deals with modification of the quick-select algorithm to make it deterministic yet still run in $O(n)$ time on an n-element sequence. The idea is to modify the way we choose the pivot so that it is chosen deterministically, not randomly, as follows:

> Partition the set S into $\lceil n/5 \rceil$ groups of size 5 each (except possibly for one group). Sort each little set and identify the median element in this set. From this set of $\lceil n/5 \rceil$ "baby" medians, apply the selection algorithm recursively to find the median of the baby medians. Use this element as the pivot and proceed as in the quick-select algorithm.

Show that this deterministic method runs in $O(n)$ time by answering the following questions (please ignore floor and ceiling functions if that simplifies the mathematics, for the asymptotics are the same either way):

a. How many baby medians are less than or equal to the chosen pivot? How many are greater than or equal to the pivot?

b. For each baby median less than or equal to the pivot, how many other elements are less than or equal to the pivot? Is the same true for those greater than or equal to the pivot?

c. Argue why the method for finding the deterministic pivot and using it to partition S takes $O(n)$ time.

d. Based on these estimates, write a recurrence equation to bound the worst-case running time $t(n)$ for this selection algorithm (note that in the worst case there are two recursive calls—one to find the median of the baby medians and one to recur on the larger of L and G).

e. Using this recurrence equation, show by induction that $t(n)$ is $O(n)$.

Projects

P-11.1 Design and implement two versions of the bucket-sort algorithm in C++, one for sorting an array of **char** values and one for sorting an array of **short** values. Experimentally compare the performance of your implementations with the sorting algorithm of the Standard Template Library.

P-11.2 Experimentally compare the performance of in-place quick-sort and a version of quick-sort that is not in-place.

P-11.3 Design and implement a version of the bucket-sort algorithm for sorting a linked list of n entries (for instance, a list of type std::list<int>) with integer keys taken from the range $[0, N-1]$, for $N \geq 2$. The algorithm should run in $O(n+N)$ time.

P-11.4 Implement merge-sort and deterministic quick-sort and perform a series of benchmarking tests to see which one is faster. Your tests should include sequences that are "random" as well as "almost" sorted.

P-11.5 Implement deterministic and randomized versions of the quick-sort algorithm and perform a series of benchmarking tests to see which one is faster. Your tests should include sequences that are very "random" looking as well as ones that are "almost" sorted.

P-11.6 Implement an in-place version of insertion-sort and an in-place version of quick-sort. Perform benchmarking tests to determine the range of values of n where quick-sort is on average better than insertion-sort.

P-11.7 Design and implement an animation for one of the sorting algorithms described in this chapter. Your animation should illustrate the key properties of this algorithm in an intuitive manner.

P-11.8 Implement the randomized quick-sort and quick-select algorithms, and design a series of experiments to test their relative speeds.

P-11.9 Implement an extended set ADT that includes the functions union(B), intersect(B), subtract(B), size(), empty(), plus the functions equals(B), contains(e), insert(e), and remove(e) with obvious meaning.

P-11.10 Implement the tree-based union/find partition data structure with both the union-by-size and path-compression heuristics.

Chapter Notes

Knuth's classic text on *Sorting and Searching* [60] contains an extensive history of the sorting problem and algorithms for solving it. Huang and Langston [48] show how to merge two sorted lists in-place in linear time. Our set ADT is derived from that of Aho, Hopcroft, and Ullman [5]. The standard quick-sort algorithm is due to Hoare [45]. More information about randomization, including Chernoff bounds, can be found in the appendix and the book by Motwani and Raghavan [80]. The quick-sort analysis given in this chapter is a combination of the analysis given in a previous edition of this book and the analysis of Kleinberg and Tardos [55]. Exercise C-11.8 is due to Littman. Gonnet and Baeza-Yates [37] analyze and experimentally compare several sorting algorithms. The term "prune-and-search" comes originally from the computational geometry literature (such as in the work of Clarkson [21] and Megiddo [71, 72]). The term "decrease-and-conquer" is from Levitin [65].

This page intentionally left blank

Chapter
12 Strings and Dynamic Programming

Contents

12.1   String Operations

Document processing is rapidly becoming one of the dominant functions of computers. Computers are used to edit documents, to search documents, to transport documents over the Internet, and to display documents on printers and computer screens. For example, the Internet document formats HTML and XML are primarily text formats, with added tags for multimedia content. Making sense of the many terabytes of information on the Internet requires a considerable amount of text processing.

In addition to having interesting applications, text processing algorithms also highlight some important algorithmic design patterns. In particular, the pattern matching problem gives rise to the **brute-force method**, which is often inefficient but has wide applicability. For text compression, we can apply the **greedy method**, which often allows us to approximate solutions to hard problems, and for some problems (such as in text compression) actually gives rise to optimal algorithms. Finally, in discussing text similarity, we introduce the **dynamic programming** design pattern, which can be applied in some special instances to solve a problem in polynomial time that appears at first to require exponential time to solve.

Text Processing

At the heart of algorithms for processing text are methods for dealing with character strings. Character strings can come from a wide variety of sources, including scientific, linguistic, and Internet applications. Indeed, the following are examples of such strings:

$$P \;=\; \text{``CGTAAACTGCTTTAATCAAACGC''}$$
$$S \;=\; \text{``http://www.wiley.com''}.$$

The first string, P, comes from DNA applications, and the second string, S, is the Internet address (URL) for the publisher of this book.

Several of the typical string processing operations involve breaking large strings into smaller strings. In order to be able to speak about the pieces that result from such operations, we use the term **substring** of an m-character string P to refer to a string of the form $P[i]P[i+1]P[i+2]\cdots P[j]$, for some $0 \le i \le j \le m-1$, that is, the string formed by the characters in P from index i to index j, inclusive. Technically, this means that a string is actually a substring of itself (taking $i=0$ and $j=m-1$), so if we want to rule this out as a possibility, we must restrict the definition to **proper** substrings, which require that either $i>0$ or $j<m-1$.

To simplify the notation for referring to substrings, let us use $P[i..j]$ to denote the substring of P from index i to index j, inclusive. That is,

$$P[i..j] = P[i]P[i+1] \cdots P[j].$$

We use the convention that if $i > j$, then $P[i..j]$ is equal to the **null string**, which has length 0. In addition, in order to distinguish some special kinds of substrings, let us refer to any substring of the form $P[0..i]$, for $0 \le i \le m-1$, as a **prefix** of P, and any substring of the form $P[i..m-1]$, for $0 \le i \le m-1$, as a **suffix** of P. For example, if we again take P to be the string of DNA given above, then "CGTAA" is a prefix of P, "CGC" is a suffix of P, and "TTAATC" is a (proper) substring of P. Note that the null string is a prefix and a suffix of any other string.

To allow for fairly general notions of a character string, we typically do not restrict the characters in T and P to explicitly come from a well-known character set, like the ASCII or Unicode character sets. Instead, we typically use the symbol Σ to denote the character set, or **alphabet**, from which characters can come. Since most document processing algorithms are used in applications where the underlying character set is finite, we usually assume that the size of the alphabet Σ, denoted with $|\Sigma|$, is a fixed constant.

12.1.1 The STL String Class

Recall from Chapter 1 that C++ supports two types of strings. A C-style string is just an array of type **char** terminated by a null character '\0'. By themselves, C-style strings do not support complex string operations. The C++ Standard Template Library (STL) provides a complete string class. This class supports a bewildering number of string operations. We list just a few of them. In the following, let S denote the STL string object on which the operation is being performed, and let Q denote another STL string or a C-style string.

size():	Return the number of characters, n, of S.
empty():	Return true if the string is empty and false otherwise.
operator$[i]$:	Return the character at index i of S, without performing array bounds checking.
at(i):	Return the character at index i of S. An out_of_range exception is thrown if i is out of bounds.
insert(i,Q):	Insert string Q prior to index i in S and return a reference to the result.
append(Q):	Append string Q to the end of S and return a reference to the result.
erase(i,m):	Remove m characters starting at index i and return a reference to the result.

substr(i,m): Return the substring of S of length m starting at index i.

find(Q): If Q is a substring of S, return the index of the beginning of the first occurrence of Q in S, else return n, the length of S.

c_str(): Return a C-style string containing the contents of S.

By default, a string is initialized to the empty string. A string may be initialized from another STL string or from a C-style string. It is not possible, however, to initialize an STL string from a single character. STL strings also support functions that return both forward and backward iterators. All operations that are defined in terms of integer indices have counterparts that are based on iterators.

The STL string class also supports assignment of one string to another. It provides relational operators, such as ==, <, >=, which are performed lexicographically. Strings can be concatenated using +, and we may append one string to another using +=. Strings can be input using >> and output using <<. The function getline(in, S) reads an entire line of input from the input stream *in* and assigns it to the string S.

The STL string class is actually a special case of a more general templated class, called basic_string<T>, which supports all the string operations but allows its elements to be of an arbitrary type, T, not just **char**. The STL string is just a short way of saying basic_string<char>. A "string of integers" could be defined as basic_string<int>.

Example 12.1: *Consider the following series of operations, which are performed on the string $S =$ "abcdefghijklmnop":*

Operation	*Output*
S.size()	16
S.at(5)	'f'
S[5]	'f'
S + "qrs"	"abcdefghijklmnopqrs"
S == "abcdefghijklmnop"	**true**
S.find("ghi")	6
S.substr(4,6)	"efghij"
S.erase(4,6)	"abcdklmnop"
S.insert(1,"xxx")	"axxxbcdklmnop"
S += "xy"	"axxxbcdklmnopxy"
S.append("z")	"axxxbcdklmnopxyz"

With the exception of the find(Q) function, which we discuss in Section 12.3, all the above functions are easily implemented simply by representing the string as an array of characters.

12.2 Dynamic Programming

In this section, we discuss the ***dynamic programming*** algorithm-design technique. This technique is similar to the divide-and-conquer technique (Section 11.1.1), in that it can be applied to a wide variety of different problems. There are few algorithmic techniques that can take problems that seem to require exponential time and produce polynomial-time algorithms to solve them. Dynamic programming is one such technique. In addition, the algorithms that result from applications of the dynamic programming technique are usually quite simple—often needing little more than a few lines of code to describe some nested loops for filling in a table.

12.2.1   Matrix Chain-Product

Rather than starting out with an explanation of the general components of the dynamic programming technique, we begin by giving a classic, concrete example. Suppose we are given a collection of n two-dimensional arrays (matrices) for which we wish to compute the product

$$A = A_0 \cdot A_1 \cdot A_2 \cdots A_{n-1},$$

where A_i is a $d_i \times d_{i+1}$ matrix, for $i = 0, 1, 2, \ldots, n-1$. In the standard matrix multiplication algorithm (which is the one we use), to multiply a $d \times e$-matrix B times an $e \times f$-matrix C, we compute the product, A, as

$$A[i][j] = \sum_{k=0}^{e-1} B[i][k] \cdot C[k][j].$$

This definition implies that matrix multiplication is associative, that is, it implies that $B \cdot (C \cdot D) = (B \cdot C) \cdot D$. Thus, we can parenthesize the expression for A any way we wish and we still end up with the same answer. We do not necessarily perform the same number of primitive (that is, scalar) multiplications in each parenthesization, however, as is illustrated in the following example.

Example 12.2: *Let B be a 2×10-matrix, let C be a 10×50-matrix, and let D be a 50×20-matrix. Computing $B \cdot (C \cdot D)$ requires $2 \cdot 10 \cdot 20 + 10 \cdot 50 \cdot 20 = 10,400$ multiplications, whereas computing $(B \cdot C) \cdot D$ requires $2 \cdot 10 \cdot 50 + 2 \cdot 50 \cdot 20 = 3000$ multiplications.*

The ***matrix chain-product*** problem is to determine the parenthesization of the expression defining the product A that minimizes the total number of scalar multiplications performed. As the example above illustrates, the differences between different solutions can be dramatic, so finding a good solution can result in significant speedups.

Defining Subproblems

Of course, one way to solve the matrix chain-product problem is to simply enumerate all the possible ways of parenthesizing the expression for A and determine the number of multiplications performed by each one. Unfortunately, the set of all different parenthesizations of the expression for A is equal in number to the set of all different binary trees that have n external nodes. This number is exponential in n. Thus, this straightforward ("brute force") algorithm runs in exponential time, for there are an exponential number of ways to parenthesize an associative arithmetic expression.

We can improve the performance achieved by the brute-force algorithm significantly, however, by making a few observations about the nature of the matrix chain-product problem. The first observation is that the problem can be split into *subproblems*. In this case, we can define a number of different subproblems, each of which computes the best parenthesization for some subexpression $A_i \cdot A_{i+1} \cdots A_j$. As a concise notation, we use $N_{i,j}$ to denote the minimum number of multiplications needed to compute this subexpression. Thus, the original matrix chain-product problem can be characterized as that of computing the value of $N_{0,n-1}$. This observation is important, but we need one more in order to apply the dynamic programming technique.

Characterizing Optimal Solutions

The other important observation we can make about the matrix chain-product problem is that it is possible to characterize an optimal solution to a particular subproblem in terms of optimal solutions to its subproblems. We call this property the *subproblem optimality* condition.

In the case of the matrix chain-product problem, we observe that, no matter how we parenthesize a subexpression, there has to be some final matrix multiplication that we perform. That is, a full parenthesization of a subexpression $A_i \cdot A_{i+1} \cdots A_j$ has to be of the form $(A_i \cdots A_k) \cdot (A_{k+1} \cdots A_j)$, for some $k \in \{i, i+1, \ldots, j-1\}$. Moreover, for whichever k is the correct one, the products $(A_i \cdots A_k)$ and $(A_{k+1} \cdots A_j)$ must also be solved optimally. If this were not so, then there would be a global optimal that had one of these subproblems solved suboptimally. But this is impossible, since we could then reduce the total number of multiplications by replacing the current subproblem solution by an optimal solution for the subproblem. This observation implies a way of explicitly defining the optimization problem for $N_{i,j}$ in terms of other optimal subproblem solutions. Namely, we can compute $N_{i,j}$ by considering each place k where we could put the final multiplication and taking the minimum over all such choices.

Designing a Dynamic Programming Algorithm

We can therefore characterize the optimal subproblem solution, $N_{i,j}$, as

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\},$$

where $N_{i,i} = 0$, since no work is needed for a single matrix. That is, $N_{i,j}$ is the minimum, taken over all possible places to perform the final multiplication, of the number of multiplications needed to compute each subexpression plus the number of multiplications needed to perform the final matrix multiplication.

Notice that there is a ***sharing of subproblems*** going on that prevents us from dividing the problem into completely independent subproblems (as we would need to do to apply the divide-and-conquer technique). We can, nevertheless, use the equation for $N_{i,j}$ to derive an efficient algorithm by computing $N_{i,j}$ values in a bottom-up fashion, and storing intermediate solutions in a table of $N_{i,j}$ values. We can begin simply enough by assigning $N_{i,i} = 0$ for $i = 0, 1, \ldots, n-1$. We can then apply the general equation for $N_{i,j}$ to compute $N_{i,i+1}$ values, since they depend only on $N_{i,i}$ and $N_{i+1,i+1}$ values that are available. Given the $N_{i,i+1}$ values, we can then compute the $N_{i,i+2}$ values, and so on. Therefore, we can build $N_{i,j}$ values up from previously computed values until we can finally compute the value of $N_{0,n-1}$, which is the number that we are searching for. The details of this ***dynamic programming*** solution are given in Code Fragment 12.1.

Algorithm MatrixChain($d_0, \ldots, d_n$):
    Input: Sequence $d_0, \ldots, d_n$ of integers
    Output: For $i, j = 0, \ldots, n-1$, the minimum number of multiplications $N_{i,j}$
        needed to compute the product $A_i \cdot A_{i+1} \cdots A_j$, where A_k is a $d_k \times d_{k+1}$ matrix
    for $i \leftarrow 0$ to $n-1$ **do**
       $N_{i,i} \leftarrow 0$
    for $b \leftarrow 1$ to $n-1$ **do**
       for $i \leftarrow 0$ to $n-b-1$ **do**
          $j \leftarrow i + b$
          $N_{i,j} \leftarrow +\infty$
          for $k \leftarrow i$ to $j-1$ **do**
             $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}.$

Code Fragment 12.1: Dynamic programming algorithm for the matrix chain-product problem.

Thus, we can compute $N_{0,n-1}$ with an algorithm that consists primarily of three nested for-loops. The outside loop is executed n times. The loop inside is executed at most n times. And the inner-most loop is also executed at most n times. Therefore, the total running time of this algorithm is $O(n^3)$.

12.2.2   DNA and Text Sequence Alignment

A common text processing problem, which arises in genetics and software engineering, is to test the similarity between two text strings. In a genetics application, the two strings could correspond to two strands of DNA, that we want to compare. Likewise, in a software engineering application, the two strings could come from two versions of source code for the same program. We might want to compare the two versions to determine what changes have been made from one version to the next. Indeed, determining the similarity between two strings is so common that the Unix and Linux operating systems have a built-in program, `diff`, for comparing text files.

Given a string $X = x_0 x_1 x_2 \cdots x_{n-1}$, a **subsequence** of X is any string that is of the form $x_{i_1} x_{i_2} \cdots x_{i_k}$, where $i_j < i_{j+1}$; that is, it is a sequence of characters that are not necessarily contiguous but are nevertheless taken in order from X. For example, the string *AAAG* is a subsequence of the string *CGATAATTGAGA*.

The DNA and text similarity problem we address here is the ***longest common subsequence*** (LCS) problem. In this problem, we are given two character strings, $X = x_0 x_1 x_2 \cdots x_{n-1}$ and $Y = y_0 y_1 y_2 \cdots y_{m-1}$, over some alphabet (such as the alphabet $\{A, C, G, T\}$ common in computational genetics) and are asked to find a longest string S that is a subsequence of both X and Y. One way to solve the longest common subsequence problem is to enumerate all subsequences of X and take the largest one that is also a subsequence of Y. Since each character of X is either in or not in a subsequence, there are potentially 2^n different subsequences of X, each of which requires $O(m)$ time to determine whether it is a subsequence of Y. Thus, this brute-force approach yields an exponential-time algorithm that runs in $O(2^n m)$ time, which is very inefficient. Fortunately, the LCS problem is efficiently solvable using ***dynamic programming***.

The Components of a Dynamic Programming Solution

As mentioned above, the dynamic programming technique is used primarily for ***optimization*** problems, where we wish to find the "best" way of doing something. We can apply the dynamic programming technique in such situations if the problem has certain properties.

Simple Subproblems: There has to be some way of repeatedly breaking the global-optimization problem into subproblems. Moreover, there should be a simple way of defining subproblems with just a few indices, like i, j, k, and so on.

Subproblem Optimization: An optimal solution to the global problem must be a composition of optimal subproblem solutions.

Subproblem Overlap: Optimal solutions to unrelated subproblems can contain subproblems in common.

Applying Dynamic Programming to the LCS Problem

Recall that in the LCS problem, we are given two character strings, X and Y, of length n and m, respectively, and are asked to find a longest string S that is a subsequence of both X and Y. Since X and Y are character strings, we have a natural set of indices with which to define subproblems—indices into the strings X and Y. Let us define a subproblem, therefore, as that of computing the value $L[i,j]$, which we will use to denote the length of a longest string that is a subsequence of both $X[0..i] = x_0 x_1 x_2 \ldots x_i$ and $Y[0..j] = y_0 y_1 y_2 \ldots y_j$. This definition allows us to rewrite $L[i,j]$ in terms of optimal subproblem solutions. This definition depends on which of two cases we are in. (See Figure 12.1.)
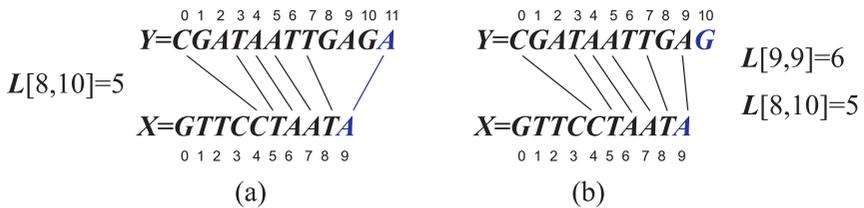
Figure 12.1: The two cases in the longest common subsequence algorithm: (a) $x_i = y_j$; (b) $x_i \neq y_j$. Note that the algorithm stores only the $L[i,j]$ values, not the matches.

- $x_i = y_j$. In this case, we have a match between the last character of $X[0..i]$ and the last character of $Y[0..j]$. We claim that this character belongs to a longest common subsequence of $X[0..i]$ and $Y[0..j]$. To justify this claim, let us suppose it is not true. There has to be some longest common subsequence $x_{i_1} x_{i_2} \ldots x_{i_k} = y_{j_1} y_{j_2} \ldots y_{j_k}$. If $x_{i_k} = x_i$ or $y_{j_k} = y_j$, then we get the same sequence by setting $i_k = i$ and $j_k = j$. Alternately, if $x_{j_k} \neq x_i$, then we can get an even longer common subsequence by adding x_i to the end. Thus, a longest common subsequence of $X[0..i]$ and $Y[0..j]$ ends with x_i. Therefore, we set

$$L[i,j] = L[i-1,j-1]+1 \quad \text{if } x_i = y_j.$$

- $x_i \neq y_j$. In this case, we cannot have a common subsequence that includes both x_i and y_j. That is, we can have a common subsequence end with x_i or one that ends with y_j (or possibly neither), but certainly not both. Therefore, we set

$$L[i,j] = \max\{L[i-1,j], L[i,j-1]\} \quad \text{if } x_i \neq y_j.$$

In order to make both of these equations make sense in the boundary cases when $i = 0$ or $j = 0$, we assign $L[i,-1] = 0$ for $i = -1, 0, 1, \ldots, n-1$ and $L[-1,j] = 0$ for $j = -1, 0, 1, \ldots, m-1$.

The LCS Algorithm

The definition of $L[i, j]$ satisfies subproblem optimization, since we cannot have a longest common subsequence without also having longest common subsequences for the subproblems. Also, it uses subproblem overlap, because a subproblem solution $L[i, j]$ can be used in several other problems (namely, the problems $L[i+1, j]$, $L[i, j+1]$, and $L[i+1, j+1]$). Turning this definition of $L[i, j]$ into an algorithm is actually quite straightforward. We initialize an $(n+1) \times (m+1)$ array, L, for the boundary cases when $i = 0$ or $j = 0$. Namely, we initialize $L[i, -1] = 0$ for $i = -1, 0, 1, \ldots, n-1$ and $L[-1, j] = 0$ for $j = -1, 0, 1, \ldots, m-1$. Then, we iteratively build up values in L until we have $L[n-1, m-1]$, the length of a longest common subsequence of X and Y. We give a pseudo-code description of this algorithm in Code Fragment 12.2.

Algorithm LCS(X,Y):

    Input: Strings X and Y with n and m elements, respectively

    Output: For $i = 0, \ldots, n-1$, $j = 0, \ldots, m-1$, the length $L[i, j]$ of a longest string that is a subsequence of both the string $X[0..i] = x_0 x_1 x_2 \cdots x_i$ and the string $Y[0..j] = y_0 y_1 y_2 \cdots y_j$

    for $i \leftarrow -1$ to $n-1$ **do**

      $L[i, -1] \leftarrow 0$

    for $j \leftarrow 0$ to $m-1$ **do**

      $L[-1, j] \leftarrow 0$

    for $i \leftarrow 0$ to $n-1$ **do**

      for $j \leftarrow 0$ to $m-1$ **do**

        if $x_i = y_j$ **then**

          $L[i, j] \leftarrow L[i-1, j-1] + 1$

        else

          $L[i, j] \leftarrow \max\{L[i-1, j], L[i, j-1]\}$

    return array L

Code Fragment 12.2: Dynamic programming algorithm for the LCS problem.

The running time of the algorithm of Code Fragment 12.2 is easy to analyze, because it is dominated by two nested **for** loops, with the outer one iterating n times and the inner one iterating m times. Since the if-statement and assignment inside the loop each requires $O(1)$ primitive operations, this algorithm runs in $O(nm)$ time. Thus, the dynamic programming technique can be applied to the longest common subsequence problem to improve significantly over the exponential-time brute-force solution to the LCS problem.

Algorithm LCS (Code Fragment 12.2) computes the length of the longest common subsequence (stored in $L[n-1, m-1]$), but not the subsequence itself. As shown in the following proposition, a simple postprocessing step can extract the longest common subsequence from the array L returned by the algorithm.

Proposition 12.3: *Given a string X of n characters and a string Y of m characters, we can find the longest common subsequence of X and Y in $O(nm)$ time.*

Justification: Algorithm LCS computes $L[n-1, m-1]$, the **length** of a longest common subsequence, in $O(nm)$ time. Given the table of $L[i, j]$ values, constructing a longest common subsequence is straightforward. One method is to start from $L[n,m]$ and work back through the table, reconstructing a longest common subsequence from back to front. At any position $L[i, j]$, we can determine whether $x_i = y_j$. If this is true, then we can take x_i as the next character of the subsequence (noting that x_i is **before** the previous character we found, if any), moving next to $L[i-1, j-1]$. If $x_i \neq y_j$, then we can move to the larger of $L[i, j-1]$ and $L[i-1, j]$. (See Figure 12.2.) We stop when we reach a boundary cell (with $i = -1$ or $j = -1$). This method constructs a longest common subsequence in $O(n+m)$ additional time. ∎

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**

0 1 2 3 4 5 6 7 8 9 10 11
Y=*CGATAATTGAGA*

X=*GTTCCTAATA*
0 1 2 3 4 5 6 7 8 9
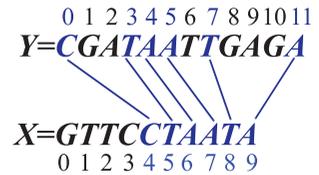
Figure 12.2: The algorithm for constructing a longest common subsequence from the array L.

12.3 Pattern Matching Algorithms

In the classic **pattern matching** problem on strings, we are given a **text** string T of length n and a **pattern** string P of length m, and want to find whether P is a substring of T. The notion of a "match" is that there is a substring of T starting at some index i that matches P, character by character, so that $T[i] = P[0]$, $T[i+1] = P[1]$, ..., $T[i+m-1] = P[m-1]$. That is, $P = T[i..i+m-1]$. Thus, the output from a pattern matching algorithm could either be some indication that the pattern P does not exist in T or an integer indicating the starting index in T of a substring matching P. This is exactly the computation performed by the find function of the STL string interface. Alternatively, one may want to find all the indices where a substring of T matching P begins.

In this section, we present three pattern matching algorithms (with increasing levels of difficulty).

12.3.1 Brute Force

The **brute-force** algorithmic design pattern is a powerful technique for algorithm design when we have something we wish to search for or when we wish to optimize some function. In applying this technique in a general situation we typically enumerate all possible configurations of the inputs involved and pick the best of all these enumerated configurations.

In applying this technique to design the **brute-force pattern matching** algorithm, we derive what is probably the first algorithm that we might think of for solving the pattern matching problem—we simply test all the possible placements of P relative to T. This algorithm, shown in Code Fragment 12.3, is quite simple.

Algorithm BruteForceMatch(T, P):
    Input: Strings T (text) with n characters and P (pattern) with m characters
    Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T

    for $i \leftarrow 0$ **to** $n - m$ {for each candidate index in T} **do**
        $j \leftarrow 0$
        while ($j < m$ **and** $T[i+j] = P[j]$) **do**
            $j \leftarrow j + 1$
        if $j = m$ **then**
            return i
    return "There is no substring of T matching P."

Code Fragment 12.3: Brute-force pattern matching.

Performance

The brute-force pattern matching algorithm could not be simpler. It consists of two nested loops, with the outer loop indexing through all possible starting indices of the pattern in the text, and the inner loop indexing through each character of the pattern, comparing it to its potentially corresponding character in the text. Thus, the correctness of the brute-force pattern matching algorithm follows immediately from this exhaustive search approach.

The running time of brute-force pattern matching in the worst case is not good, however, because, for each candidate index in T, we can perform up to m character comparisons to discover that P does not match T at the current index. Referring to Code Fragment 12.3, we see that the outer **for** loop is executed at most $n - m + 1$ times, and the inner loop is executed at most m times. Thus, the running time of the brute-force method is $O((n - m + 1)m)$, which is simplified as $O(nm)$. Note that when $m = n/2$, this algorithm has quadratic running time $O(n^2)$.

Example 12.4: *Suppose we are given the text string*

$$T = \texttt{"abacaabaccabacabaabb"}$$

and the pattern string

$$P = \texttt{"abacab"}.$$

In Figure 12.3, we illustrate the execution of the brute-force pattern matching algorithm on T and P.
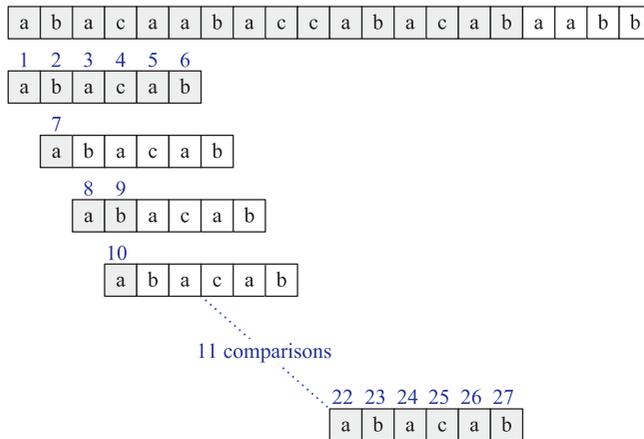
Figure 12.3: Example run of the brute-force pattern matching algorithm. The algorithm performs 27 character comparisons, indicated above with numerical labels.

12.3.2 The Boyer-Moore Algorithm

At first, we might feel that it is always necessary to examine every character in T in order to locate a pattern P as a substring. But this is not always the case. The **Boyer-Moore** (**BM**) pattern matching algorithm, which we study in this section, can sometimes avoid comparisons between P and a sizable fraction of the characters in T. The only caveat is that, whereas the brute-force algorithm can work even with a potentially unbounded alphabet, the BM algorithm assumes the alphabet is of fixed, finite size. It works the fastest when the alphabet is moderately sized and the pattern is relatively long. Thus, the BM algorithm is ideal for searching words in documents. In this section, we describe a simplified version of the original algorithm by Boyer and Moore.

The main idea of the BM algorithm is to improve the running time of the brute-force algorithm by adding two potentially time-saving heuristics. Roughly stated, these heuristics are as follows:

Looking-Glass Heuristic: When testing a possible placement of P against T, begin the comparisons from the end of P and move backward to the front of P.

Character-Jump Heuristic: During the testing of a possible placement of P against T, a mismatch of text character $T[i] = c$ with the corresponding pattern character $P[j]$ is handled as follows. If c is not contained anywhere in P, then shift P completely past $T[i]$ (for it cannot match any character in P). Otherwise, shift P until an occurrence of character c in P gets aligned with $T[i]$.

We formalize these heuristics shortly, but at an intuitive level, they work as an integrated team. The looking-glass heuristic sets up the other heuristic to allow us to avoid comparisons between P and whole groups of characters in T. In this case at least, we can get to the destination faster by going backwards, for if we encounter a mismatch during the consideration of P at a certain location in T, then we are likely to avoid lots of needless comparisons by significantly shifting P relative to T using the character-jump heuristic. The character-jump heuristic pays off big if it can be applied early in the testing of a potential placement of P against T.

Let us therefore get down to the business of defining how the character-jump heuristics can be integrated into a string pattern matching algorithm. To implement this heuristic, we define a function $\mathsf{last}(c)$ that takes a character c from the alphabet and characterizes how far we may shift the pattern P if a character equal to c is found in the text that does not match the pattern. In particular, we define $\mathsf{last}(c)$ as:

- If c is in P, $\mathsf{last}(c)$ is the index of the last (right-most) occurrence of c in P. Otherwise, we conventionally define $\mathsf{last}(c) = -1$.

If characters can be used as indices in arrays, then the last function can be easily implemented as a lookup table. We leave the method for computing this table in $O(m + |\Sigma|)$ time, given P, as a simple exercise (Exercise R-12.8). This last function gives us all the information we need to perform the character-jump heuristic.

In Code Fragment 12.4, we show the BM pattern matching algorithm.

Algorithm BMMatch(T,P):

   Input: Strings T (text) with n characters and P (pattern) with m characters

   Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T

   compute function last

   $i \leftarrow m-1$

   $j \leftarrow m-1$

   repeat

     if $P[j] = T[i]$ **then**

       if $j = 0$ **then**

         return i {a match!}

       else

         $i \leftarrow i-1$

         $j \leftarrow j-1$

     else

       $i \leftarrow i+m-\min(j, 1+\mathsf{last}(T[i]))$ {jump step}

       $j \leftarrow m-1$

   until $i > n-1$

   return "There is no substring of T matching P."

     Code Fragment 12.4: The Boyer-Moore pattern matching algorithm.

The jump step is illustrated in Figure 12.4.
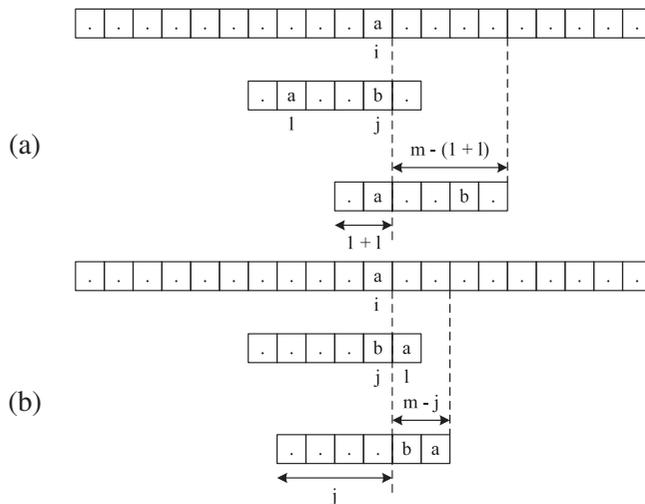
Figure 12.4: The jump step in the algorithm of Code Fragment 12.4, where we let $l = \mathsf{last}(T[i])$. We distinguish two cases: (a) $1+l \le j$, where we shift the pattern by $j-l$ units; (b) $j < 1+l$, where we shift the pattern by one unit.

In Figure 12.5, we illustrate the execution of the Boyer-Moore pattern matching algorithm on an input string similar to Example 12.4.

a	b	a	c	a	a	b	a	d	c	a	b	a	c	a	b	a	a	b	b

1

a	b	a	c	a	b

        4   3   2   13   12 11 10   9   8

a	b	a	c	a	b		a	b	a	c	a	b

                  5   7

a	b	a	c	a	b		a	b	a	c	a	b

                  6

a	b	a	c	a	b
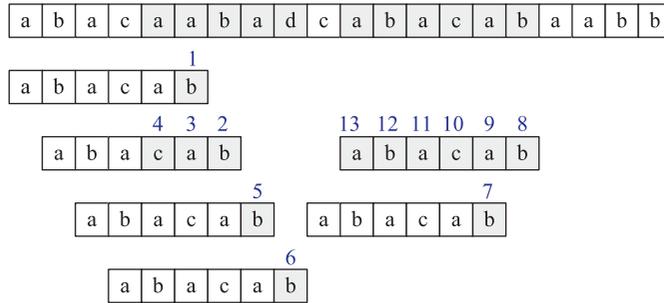
The last(c) function:

c	a	b	c	d
last(c)	4	5	3	-1

Figure 12.5: The BM pattern matching algorithm. The algorithm performs 13 character comparisons, which are indicated with numerical labels.

The correctness of the BM pattern matching algorithm follows from the fact that each time the method makes a shift, it is guaranteed not to "skip" over any possible matches. This is because last(c) indicates the ***last*** occurrence of c in P.

The worst-case running time of the BM algorithm is $O(nm + |\Sigma|)$. Namely, the computation of the last function takes $O(m + |\Sigma|)$ time and the actual search for the pattern takes $O(nm)$ time in the worst case, the same as the brute-force algorithm. An example of a text-pattern pair that achieves the worst case is

$$T = \overbrace{aaaaaa\cdots a}^{n}$$
$$P = b\overbrace{aa\cdots a}^{m-1}.$$

The worst-case performance, however, is unlikely to be achieved for English text because, in this case, the BM algorithm is often able to skip large portions of text. (See Figure 12.6.) Experimental evidence on English text shows that the average number of comparisons done per character is 0.24 for a five-character pattern string.
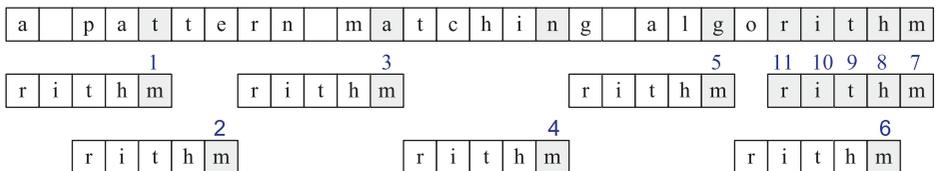
Figure 12.6: An example of a Boyer-Moore execution on English text.

A C++ implementation of the BM pattern matching algorithm, based on an STL vector, is shown in Code Fragment 12.5.

```cpp
/** Simplified version of the Boyer-Moore algorithm. Returns the index of
 *  the leftmost substring of the text matching the pattern, or -1 if none.
 */
int BMmatch(const string& text, const string& pattern) {
  std::vector<int> last = buildLastFunction(pattern);
  int n = text.size();
  int m = pattern.size();
  int i = m − 1;
  if (i > n − 1)                          // pattern longer than text?
    return −1;                            // ...then no match
  int j = m − 1;
  do {
    if (pattern[j] == text[i])
      if (j == 0) return i;               // found a match
      else {                              // looking-glass heuristic
        i−−; j−−;                         // proceed right-to-left
      }
    else {                                // character-jump heuristic
      i = i + m − std::min(j, 1 + last[text[i]]);
      j = m − 1;
    }
  } while (i <= n − 1);
  return −1;                              // no match
}
                                          // construct function last
std::vector<int> buildLastFunction(const string& pattern) {
  const int N_ASCII = 128;                // number of ASCII characters
  int i;
  std::vector<int> last(N_ASCII);         // assume ASCII character set
  for (i = 0; i < N_ASCII; i++)           // initialize array
    last[i] = −1;
  for (i = 0; i < pattern.size(); i++) {
    last[pattern[i]] = i;                 // (implicit cast to ASCII code)
  }
  return last;
}
```

Code Fragment 12.5: C++ implementation of the Boyer-Moore (BM) pattern matching algorithm. The algorithm is expressed by two static functions: Method BMmatch performs the matching and calls the auxiliary function buildLastFunction to compute the last function, expressed by an array indexed by the ASCII code of the character. Method BMmatch indicates the absence of a match by returning the conventional value −1.

We have actually presented a simplified version of the Boyer-Moore (BM) algorithm. The original BM algorithm achieves running time $O(n+m+|\Sigma|)$ by using an alternative shift heuristic to the partially matched text string, whenever it shifts the pattern more than the character-jump heuristic. This alternative shift heuristic is based on applying the main idea from the Knuth-Morris-Pratt pattern matching algorithm, which we discuss next.

12.3.3   The Knuth-Morris-Pratt Algorithm

In studying the worst-case performance of the brute-force and BM pattern matching algorithms on specific instances of the problem, such as that given in Example 12.4, we should notice a major inefficiency. Specifically, we may perform many comparisons while testing a potential placement of the pattern against the text, yet if we discover a pattern character that does not match in the text, then we throw away all the information gained by these comparisons and start over again from scratch with the next incremental placement of the pattern. The Knuth-Morris-Pratt (or "KMP") algorithm discussed in this section, avoids this waste of information and, in so doing, it achieves a running time of $O(n+m)$, which is optimal in the worst case. That is, in the worst case any pattern matching algorithm will have to examine all the characters of the text and all the characters of the pattern at least once.

The Failure Function

The main idea of the KMP algorithm is to preprocess the pattern string P so as to compute a *failure function*, f, that indicates the proper shift of P so that, to the largest extent possible, we can reuse previously performed comparisons. Specifically, the failure function $f(j)$ is defined as the length of the longest prefix of P that is a suffix of $P[1..j]$ (note that we did *not* put $P[0..j]$ here). We also use the convention that $f(0) = 0$. Later, we discuss how to compute the failure function efficiently. The importance of this failure function is that it "encodes" repeated substrings inside the pattern itself.

Example 12.5: *Consider the pattern string $P =$ "abacab" from Example 12.4. The Knuth-Morris-Pratt (KMP) failure function, $f(j)$, for the string P is as shown in the following table:*

j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$f(j)$	0	0	1	0	1	2

The KMP pattern matching algorithm, shown in Code Fragment 12.6, incrementally processes the text string T comparing it to the pattern string P. Each time there is a match, we increment the current indices. On the other hand, if there is a mismatch and we have previously made progress in P, then we consult the failure function to determine the new index in P where we need to continue checking P against T. Otherwise (there was a mismatch and we are at the beginning of P), we simply increment the index for T (and keep the index variable for P at its beginning). We repeat this process until we find a match of P in T or the index for T reaches n, the length of T (indicating that we did not find the pattern P in T).

Algorithm KMPMatch(T, P):
    Input: Strings T (text) with n characters and P (pattern) with m characters
    Output: Starting index of the first substring of T matching P, or an indication
        that P is not a substring of T
    $f \leftarrow$ KMPFailureFunction(P)   {construct the failure function f for P}
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $i < n$ **do**
      if $P[j] = T[i]$ **then**
        if $j = m - 1$ **then**
          return $i - m + 1$ {a match!}
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
      else if $j > 0$ {no match, but we have advanced in P} **then**
        $j \leftarrow f(j-1)$ {j indexes just after prefix of P that must match}
      else
        $i \leftarrow i + 1$
    return "There is no substring of T matching P."

Code Fragment 12.6: The KMP pattern matching algorithm.

The main part of the KMP algorithm is the **while** loop, which performs a comparison between a character in T and a character in P each iteration. Depending upon the outcome of this comparison, the algorithm either moves on to the next characters in T and P, consults the failure function for a new candidate character in P, or starts over with the next index in T. The correctness of this algorithm follows from the definition of the failure function. Any comparisons that are skipped are actually unnecessary, for the failure function guarantees that all the ignored comparisons are redundant—they would involve comparing the same matching characters over again.

| a | b | a | c | a | a | b | a | c | c | a | b | a | c | a | b | a | a | b | b |

1 2 3 4 5 6
| a | b | a | c | a | b |

7
| a | b | a | c | a | b |

no comparison
needed here

8 9 10 11 12
| a | b | a | c | a | b |

13
| a | b | a | c | a | b |

14 15 16 17 18 19
| a | b | a | c | a | b |
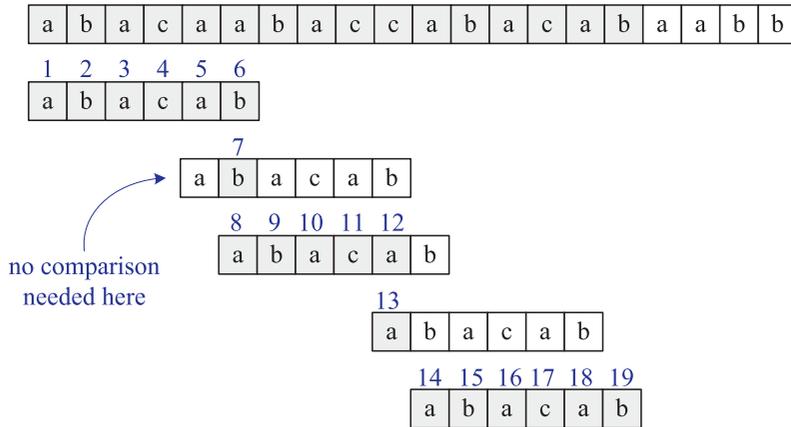
Figure 12.7: The KMP pattern matching algorithm. The failure function f for this pattern is given in Example 12.5. The algorithm performs 19 character comparisons, which are indicated with numerical labels.

In Figure 12.7, we illustrate the execution of the KMP pattern matching algorithm on the same input strings as in Example 12.4. Note the use of the failure function to avoid redoing one of the comparisons between a character of the pattern and a character of the text. Also note that the algorithm performs fewer overall comparisons than the brute-force algorithm run on the same strings (Figure 12.3).

Performance

Excluding the computation of the failure function, the running time of the KMP algorithm is clearly proportional to the number of iterations of the **while** loop. For the sake of analysis, let us define $k = i - j$. Intuitively, k is the total amount by which the pattern P has been shifted with respect to the text T. Note that throughout the execution of the algorithm, we have $k \leq n$. One of the following three cases occurs at each iteration of the loop.

- If $T[i] = P[j]$, then i increases by 1, and k does not change, since j also increases by 1.
- If $T[i] \neq P[j]$ and $j > 0$, then i does not change and k increases by at least 1, since, in this case, k changes from $i - j$ to $i - f(j-1)$, which is an addition of $j - f(j-1)$, which is positive because $f(j-1) < j$.
- If $T[i] \neq P[j]$ and $j = 0$, then i increases by 1 and k increases by 1, since j does not change.

Thus, at each iteration of the loop, either i or k increases by at least 1 (possibly both); hence, the total number of iterations of the **while** loop in the KMP pattern matching algorithm is at most $2n$. Of course, achieving this bound assumes that we have already computed the failure function for P.

Constructing the KMP Failure Function

To construct the failure function, we use the method shown in Code Fragment 12.7, which is a "bootstrapping" process quite similar to the KMPMatch algorithm. We compare the pattern to itself as in the KMP algorithm. Each time we have two characters that match, we set $f(i) = j+1$. Note that since we have $i > j$ throughout the execution of the algorithm, $f(j-1)$ is always defined when we need to use it.

Algorithm KMPFailureFunction(P):
    Input: String P (pattern) with m characters
    Output: The failure function f for P, which maps j to the length of the longest
        prefix of P that is a suffix of $P[1..j]$
    $i \leftarrow 1$
    $j \leftarrow 0$
    $f(0) \leftarrow 0$
    while $i < m$ **do**
      if $P[j] = P[i]$ **then**
        {we have matched $j+1$ characters}
        $f(i) \leftarrow j+1$
        $i \leftarrow i+1$
        $j \leftarrow j+1$
      else if $j > 0$ **then**
        {j indexes just after a prefix of P that must match}
        $j \leftarrow f(j-1)$
      else
        {we have no match here}
        $f(i) \leftarrow 0$
        $i \leftarrow i+1$

Code Fragment 12.7: Computation of the failure function used in the KMP pattern matching algorithm. Note how the algorithm uses the previous values of the failure function to efficiently compute new values.

Algorithm KMPFailureFunction runs in $O(m)$ time. Its analysis is analogous to that of algorithm KMPMatch. Thus, we have:

Proposition 12.6: *The Knuth-Morris-Pratt algorithm performs pattern matching on a text string of length n and a pattern string of length m in $O(n+m)$ time.*

A C++ implementation of the KMP pattern matching algorithm, based on an STL vector, is shown in Code Fragment 12.8.

```
                                                    // KMP algorithm
int KMPmatch(const string& text, const string& pattern) {
  int n = text.size();
  int m = pattern.size();
  std::vector<int> fail = computeFailFunction(pattern);
  int i = 0;                                         // text index
  int j = 0;                                         // pattern index
  while (i < n) {
    if (pattern[j] == text[i]) {
      if (j == m − 1)
        return i − m + 1;                            // found a match
      i++; j++;
    }
    else if (j > 0) j = fail[j − 1];
    else i++;
  }
  return −1;                                         // no match
}

std::vector<int> computeFailFunction(const string& pattern) {
  std::vector<int> fail(pattern.size());
  fail[0] = 0;
  int m = pattern.size();
  int j = 0;
  int i = 1;
  while (i < m) {
    if (pattern[j] == pattern[i]) {                  // j + 1 characters match
      fail[i] = j + 1;
      i++; j++;
    }
    else if (j > 0)                                  // j follows a matching prefix
      j = fail[j − 1];
    else {                                           // no match
      fail[i] = 0;
      i++;
    }
  }
  return fail;
}
```

Code Fragment 12.8: C++ implementation of the KMP pattern matching algorithm. The algorithm is expressed by two static functions. Function KMPmatch performs the matching and calls the auxiliary function computeFailFunction to compute the failure function, expressed by an array. Method KMPmatch indicates the absence of a match by returning the conventional value −1.

12.4 Text Compression and the Greedy Method

In this section, we consider an important text processing task, ***text compression***. In this problem, we are given a string X defined over some alphabet, such as the ASCII or Unicode character sets, and we want to efficiently encode X into a small binary string Y (using only the characters 0 and 1). Text compression is useful in any situation where we are communicating over a low-bandwidth channel, such as a modem line or infrared connection, and we wish to minimize the time needed to transmit our text. Likewise, text compression is also useful for storing collections of large documents more efficiently, in order to allow for a fixed-capacity storage device to contain as many documents as possible.

The method for text compression explored in this section is the ***Huffman code***. Standard encoding schemes, such as the ASCII and Unicode systems, use fixed-length binary strings to encode characters (with 7 bits in the ASCII system and 16 in the Unicode system). A Huffman code, on the other hand, uses a variable-length encoding optimized for the string X. The optimization is based on the use of character ***frequencies***, where we have, for each character c, a count $f(c)$ of the number of times c appears in the string X. The Huffman code saves space over a fixed-length encoding by using short code-word strings to encode high-frequency characters and long code-word strings to encode low-frequency characters.

To encode the string X, we convert each character in X from its fixed-length code word to its variable-length code word, and we concatenate all these code words in order to produce the encoding Y for X. In order to avoid ambiguities, we insist that no code word in our encoding is a prefix of another code word in our encoding. Such a code is called a ***prefix code***, and it simplifies the decoding of Y in order to get back X. (See Figure 12.8.) Even with this restriction, the savings produced by a variable-length prefix code can be significant, particularly if there is a wide variance in character frequencies (as is the case for natural language text in almost every spoken language).

Huffman's algorithm for producing an optimal variable-length prefix code for X is based on the construction of a binary tree T that represents the code. Each node in T, except the root, represents a bit in a code word, with each left child representing a "0" and each right child representing a "1." Each external node v is associated with a specific character, and the code word for that character is defined by the sequence of bits associated with the nodes in the path from the root of T to v. (See Figure 12.8.) Each external node v has a ***frequency***, $f(v)$, which is simply the frequency in X of the character associated with v. In addition, we give each internal node v in T a frequency, $f(v)$, that is the sum of the frequencies of all the external nodes in the subtree rooted at v.
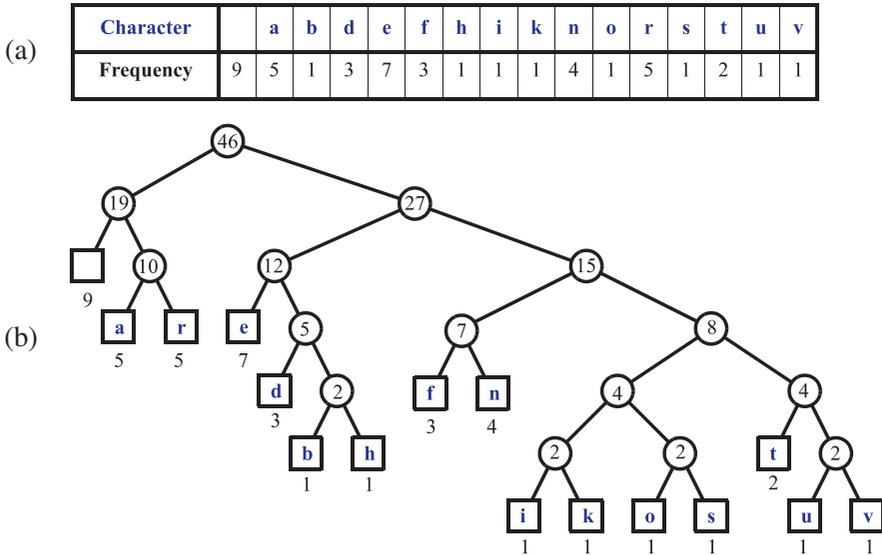
Figure 12.8: An example Huffman code for the input string $X = $ "a fast runner need never be afraid of the dark": (a) frequency of each character of X; (b) Huffman tree T for string X. The code for a character c is obtained by tracing the path from the root of T to the external node where c is stored, and associating a left child with 0 and a right child with 1. For example, the code for "a" is 010, and the code for "f" is 1100.

12.4.1 The Huffman-Coding Algorithm

The Huffman-coding algorithm begins with each of the d distinct characters of the string X to encode being the root node of a single-node binary tree. The algorithm proceeds in a series of rounds. In each round, the algorithm takes the two binary trees with the smallest frequencies and merges them into a single binary tree. It repeats this process until only one tree is left. (See Code Fragment 12.9.)

Each iteration of the **while** loop in Huffman's algorithm can be implemented in $O(\log d)$ time using a priority queue represented with a heap. In addition, each iteration takes two nodes out of Q and adds one in, a process that is repeated $d - 1$ times before exactly one node is left in Q. Thus, this algorithm runs in $O(n + d \log d)$ time. Although a full justification of this algorithm's correctness is beyond our scope, we note that its intuition comes from a simple idea—any optimal code can be converted into an optimal code in which the code words for the two lowest-frequency characters, a and b, differ only in their last bit. Repeating the argument for a string with a and b replaced by a character c, gives the following.

Proposition 12.7: *Huffman's algorithm constructs an optimal prefix code for a string of length n with d distinct characters in $O(n + d \log d)$ time.*

Algorithm Huffman(X):

   Input: String X of length n with d distinct characters

   Output: Coding tree for X

   Compute the frequency $f(c)$ of each character c of X.

   Initialize a priority queue Q.

   for each character c in X **do**

      Create a single-node binary tree T storing c.

      Insert T into Q with key $f(c)$.

   while Q.size() > 1 **do**

      $f_1 \leftarrow Q$.min()

      $T_1 \leftarrow Q$.removeMin()

      $f_2 \leftarrow Q$.min()

      $T_2 \leftarrow Q$.removeMin()

      Create a new binary tree T with left subtree T_1 and right subtree T_2.

      Insert T into Q with key $f_1 + f_2$.

   return tree Q.removeMin()

Code Fragment 12.9: Huffman-coding algorithm.

12.4.2 The Greedy Method

Huffman's algorithm for building an optimal encoding is an example application of an algorithmic design pattern called the ***greedy method***. This design pattern is applied to optimization problems, where we are trying to construct some structure while minimizing or maximizing some property of that structure.

The general formula for the greedy method pattern is almost as simple as that for the brute-force method. In order to solve a given optimization problem using the greedy method, we proceed by a sequence of choices. The sequence starts from some well-understood starting condition, and computes the cost for that initial condition. The pattern then asks that we iteratively make additional choices by identifying the decision that achieves the best cost improvement from all of the choices that are currently possible. This approach does not always lead to an optimal solution.

But there are several problems that it does work for, and such problems are said to possess the ***greedy-choice*** property. This is the property that a global optimal condition can be reached by a series of locally optimal choices (that is, choices that are each the current best from among the possibilities available at the time), starting from a well-defined starting condition. The problem of computing an optimal variable-length prefix code is just one example of a problem that possesses the greedy-choice property.

12.5   Tries

The pattern matching algorithms presented in the previous section speed up the search in a text by preprocessing the pattern (to compute the failure function in the KMP algorithm or the last function in the BM algorithm). In this section, we take a complementary approach, namely, we present string searching algorithms that preprocess the text. This approach is suitable for applications where a series of queries is performed on a fixed text, so that the initial cost of preprocessing the text is compensated by a speedup in each subsequent query (for example, a Web site that offers pattern matching in Shakespeare's *Hamlet* or a search engine that offers Web pages on the *Hamlet* topic).

A **trie** (pronounced "try") is a tree-based data structure for storing strings in order to support fast pattern matching. The main application for tries is in information retrieval. Indeed, the name "trie" comes from the word "re*trie*val." In an information retrieval application, such as a search for a certain DNA sequence in a genomic database, we are given a collection S of strings, all defined using the same alphabet. The primary query operations that tries support are pattern matching and **prefix matching**. The latter operation involves being given a string X, and looking for all the strings in S that contain X as a prefix.

12.5.1   Standard Tries

Let S be a set of s strings from alphabet Σ such that no string in S is a prefix of another string. A **standard trie** for S is an ordered tree T with the following properties (see Figure 12.9):

- Each node of T, except the root, is labeled with a character of Σ.
- The ordering of the children of an internal node of T is determined by a canonical ordering of the alphabet Σ.
- T has s external nodes, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to an external node v of T yields the string of S associated with v.

Thus, a trie T represents the strings of S with paths from the root to the external nodes of T. Note the importance of assuming that no string in S is a prefix of another string. This ensures that each string of S is uniquely associated with an external node of T. We can always satisfy this assumption by adding a special character that is not in the original alphabet Σ at the end of each string.

An internal node in a standard trie T can have anywhere between 1 and d children, where d is the size of the alphabet. There is an edge going from the root r to one of its children for each character that is first in some string in the collection S. In addition, a path from the root of T to an internal node v at depth i corresponds to
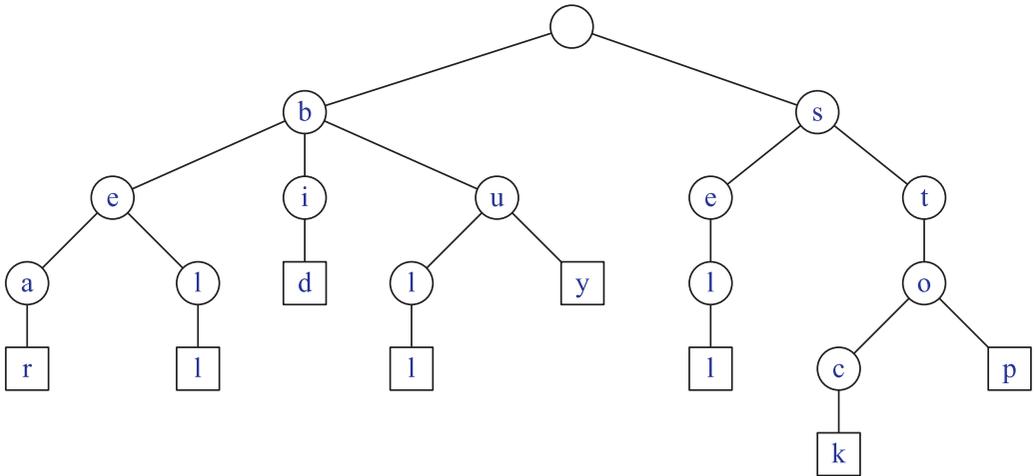
Figure 12.9: Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

an i-character prefix $X[0..i-1]$ of a string X of S. In fact, for each character c that can follow the prefix $X[0..i-1]$ in a string of the set S, there is a child of v labeled with character c. In this way, a trie concisely stores the common prefixes that exist among a set of strings.

If there are only two characters in the alphabet, then the trie is essentially a binary tree, with some internal nodes possibly having only one child (that is, it may be an improper binary tree). In general, if there are d characters in the alphabet, then the trie will be a multi-way tree where each internal node has between 1 and d children. In addition, there are likely to be several internal nodes in a standard trie that have fewer than d children. For example, the trie shown in Figure 12.9 has several internal nodes with only one child. We can implement a trie with a tree storing characters at its nodes.

The following proposition provides some important structural properties of a standard trie.

Proposition 12.8: *A standard trie storing a collection S of s strings of total length n from an alphabet of size d has the following properties:*

- *Every internal node of T has at most d children*
- *T has s external nodes*
- *The height of T is equal to the length of the longest string in S*
- *The number of nodes of T is $O(n)$*

The worst case for the number of nodes of a trie occurs when no two strings share a common nonempty prefix; that is, except for the root, all internal nodes have one child.

A trie T for a set S of strings can be used to implement a dictionary whose keys are the strings of S. Namely, we perform a search in T for a string X by tracing down from the root the path indicated by the characters in X. If this path can be traced and terminates at an external node, then we know X is in the dictionary. For example, in the trie in Figure 12.9, tracing the path for "bull" ends up at an external node. If the path cannot be traced or the path can be traced but terminates at an internal node, then X is not in the dictionary. In the example in Figure 12.9, the path for "bet" cannot be traced and the path for "be" ends at an internal node. Neither such word is in the dictionary. Note that in this implementation of a dictionary, single characters are compared instead of the entire string (key). It is easy to see that the running time of the search for a string of size m is $O(dm)$, where d is the size of the alphabet. Indeed, we visit at most $m+1$ nodes of T and we spend $O(d)$ time at each node. For some alphabets, we may be able to improve the time spent at a node to be $O(1)$ or $O(\log d)$ by using a dictionary of characters implemented in a hash table or search table. However, since d is a constant in most applications, we can stick with the simple approach that takes $O(d)$ time per node visited.

From the discussion above, it follows that we can use a trie to perform a special type of pattern matching, called ***word matching***, where we want to determine whether a given pattern matches one of the words of the text exactly. (See Figure 12.10.) Word matching differs from standard pattern matching since the pattern cannot match an arbitrary substring of the text, but only one of its words. Using a trie, word matching for a pattern of length m takes $O(dm)$ time, where d is the size of the alphabet, independent of the size of the text. If the alphabet has constant size (as is the case for text in natural languages and DNA strings), a query takes $O(m)$ time, proportional to the size of the pattern. A simple extension of this scheme supports prefix matching queries. However, arbitrary occurrences of the pattern in the text (for example, the pattern is a proper suffix of a word or spans two words) cannot be efficiently performed.

To construct a standard trie for a set S of strings, we can use an incremental algorithm that inserts the strings one at a time. Recall the assumption that no string of S is a prefix of another string. To insert a string X into the current trie T, we first try to trace the path associated with X in T. Since X is not already in T and no string in S is a prefix of another string, we stop tracing the path at an ***internal*** node v of T before reaching the end of X. We then create a new chain of node descendents of v to store the remaining characters of X. The time to insert X is $O(dm)$, where m is the length of X and d is the size of the alphabet. Thus, constructing the entire trie for set S takes $O(dn)$ time, where n is the total length of the strings of S.

s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!	
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46

b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!	
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68

h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88

(a)
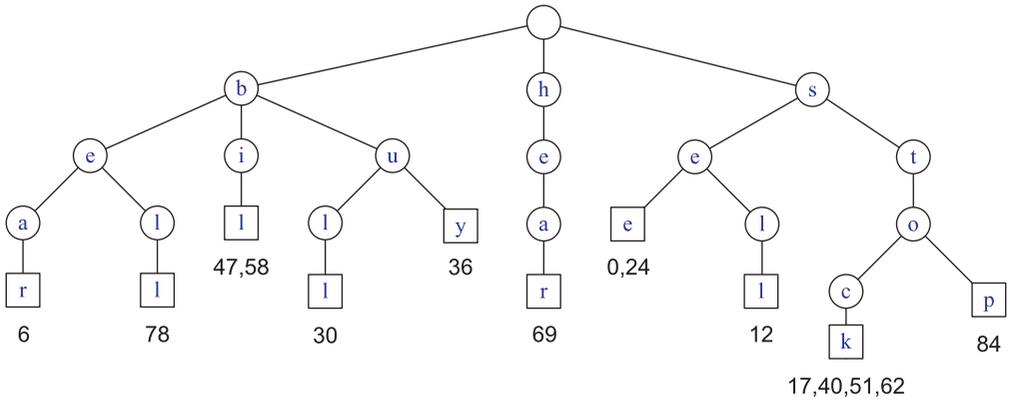
(b)

Figure 12.10: Word matching and prefix matching with a standard trie: (a) text to be searched; (b) standard trie for the words in the text (articles and prepositions, which are also known as **stop words**, excluded), with external nodes augmented with indications of the word positions.

There is a potential space inefficiency in the standard trie that has prompted the development of the **compressed trie**, which is also known (for historical reasons) as the **Patricia trie**. Namely, there are potentially a lot of nodes in the standard trie that have only one child, and the existence of such nodes is a waste. We discuss the compressed trie next.

12.5.2 Compressed Tries

A *compressed trie* is similar to a standard trie but it ensures that each internal node in the trie has at least two children. It enforces this rule by compressing chains of single-child nodes into individual edges. (See Figure 12.11.) Let T be a standard trie. We say that an internal node v of T is *redundant* if v has one child and is not the root. For example, the trie of Figure 12.9 has eight redundant nodes. Let us also say that a chain of $k \geq 2$ edges

$$(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k),$$

is *redundant* if:

- v_i is redundant for $i = 1, \ldots, k-1$
- v_0 and v_k are not redundant

We can transform T into a compressed trie by replacing each redundant chain $(v_0, v_1) \cdots (v_{k-1}, v_k)$ of $k \geq 2$ edges into a single edge (v_0, v_k), relabeling v_k with the concatenation of the labels of nodes $v_1, \ldots, v_k$.
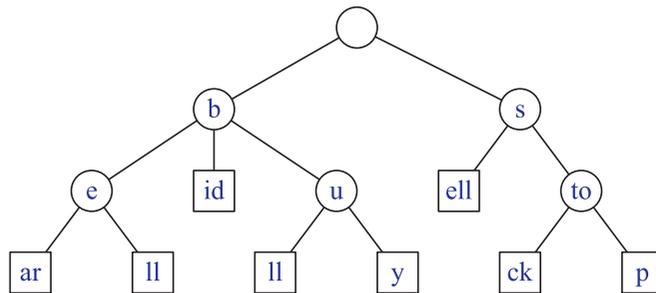
Figure 12.11: Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}. Compare this with the standard trie shown in Figure 12.9.

Thus, nodes in a compressed trie are labeled with strings, which are substrings of strings in the collection, rather than with individual characters. The advantage of a compressed trie over a standard trie is that the number of nodes of the compressed trie is proportional to the number of strings and not to their total length, as shown in the following proposition (compare with Proposition 12.8).

Proposition 12.9: *A compressed trie storing a collection S of s strings from an alphabet of size d has the following properties:*

- *Every internal node of T has at least two children and most d children*
- *T has s external nodes*
- *The number of nodes of T is $O(s)$*

The attentive reader may wonder whether the compression of paths provides any significant advantage, since it is offset by a corresponding expansion of the node labels. Indeed, a compressed trie is truly advantageous only when it is used as an *auxiliary* index structure over a collection of strings already stored in a primary structure, and is not required to actually store all the characters of the strings in the collection.

Suppose, for example, that the collection S of strings is an array of strings $S[0]$, $S[1]$, …, $S[s-1]$. Instead of storing the label X of a node explicitly, we represent it implicitly by a triplet of integers (i, j, k), such that $X = S[i][j..k]$; that is, X is the substring of $S[i]$ consisting of the characters from the jth to the kth included. (See the example in Figure 12.12. Also compare with the standard trie of Figure 12.10.)

	0	1	2	3	4
$S[0] =$	s	e	e		
$S[1] =$	b	e	a	r	
$S[2] =$	s	e	l	l	
$S[3] =$	s	t	o	c	k

	0	1	2	3
$S[4] =$	b	u	l	l
$S[5] =$	b	u	y	
$S[6] =$	b	i	d	

	0	1	2	3
$S[7] =$	h	e	a	r
$S[8] =$	b	e	l	l
$S[9] =$	s	t	o	p

(a)

Tree (compressed trie):

- Root
  - 1, 0, 0
    - 1, 1, 1
      - 1, 2, 3
      - 8, 2, 3
    - 6, 1, 2
  - 7, 0 3
    - 4, 1, 1
      - 4, 2, 3
      - 5, 2, 2
  - 0, 0, 0
    - 0, 1, 1
      - 0, 2, 2
      - 2, 2, 3
    - 3, 1, 2
      - 3, 3, 4
      - 9, 3, 3
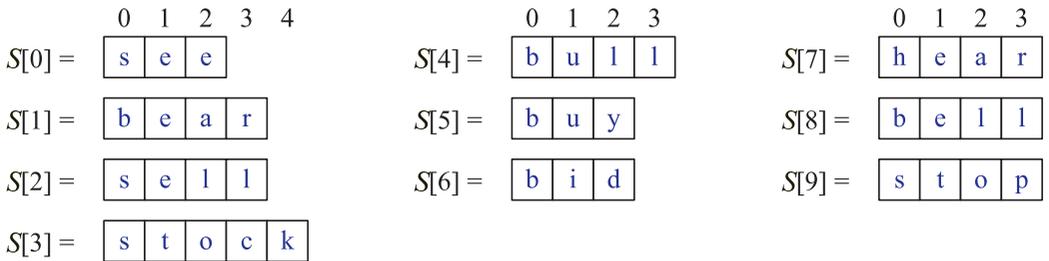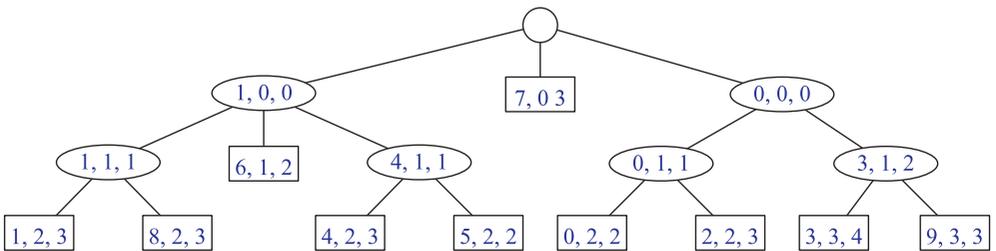
(b)

Figure 12.12: (a) Collection S of strings stored in an array. (b) Compact representation of the compressed trie for S.

This additional compression scheme allows us to reduce the total space for the trie itself from $O(n)$ for the standard trie to $O(s)$ for the compressed trie, where n is the total length of the strings in S and s is the number of strings in S. We must still store the different strings in S, of course, but we nevertheless reduce the space for the trie.

12.5.3 Suffix Tries

One of the primary applications for tries is for the case when the strings in the collection S are all the suffixes of a string X. Such a trie is called the ***suffix trie*** (also known as a ***suffix tree*** or ***position tree***) of string X. For example, Figure 12.13(a) shows the suffix trie for the eight suffixes of string "minimize." For a suffix trie, the compact representation presented in the previous section can be further simplified. Namely, the label of each vertex is a pair (i, j) indicating the string $X[i..j]$. (See Figure 12.13(b).) To satisfy the rule that no suffix of X is a prefix of another suffix, we can add a special character, denoted with $, that is not in the original alphabet Σ at the end of X (and thus to every suffix). That is, if string X has length n, we build a trie for the set of n strings $X[i..n-1]$\$, for $i = 0, \ldots, n-1$.

Saving Space

Using a suffix trie allows us to save space over a standard trie by using several space compression techniques, including those used for the compressed trie.

The advantage of the compact representation of tries now becomes apparent for suffix tries. Since the total length of the suffixes of a string X of length n is

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2},$$

storing all the suffixes of X explicitly would take $O(n^2)$ space. Even so, the suffix trie represents these strings implicitly in $O(n)$ space, as formally stated in the following proposition.

Proposition 12.10: *The compact representation of a suffix trie T for a string X of length n uses $O(n)$ space.*

Construction

We can construct the suffix trie for a string of length n with an incremental algorithm like the one given in Section 12.5.1. This construction takes $O(dn^2)$ time because the total length of the suffixes is quadratic in n. However, the (compact) suffix trie for a string of length n can be constructed in $O(n)$ time with a specialized algorithm, different from the one for general tries. This linear-time construction algorithm is fairly complex, however, and is not reported here. Still, we can take advantage of the existence of this fast construction algorithm when we want to use a suffix trie to solve other problems.
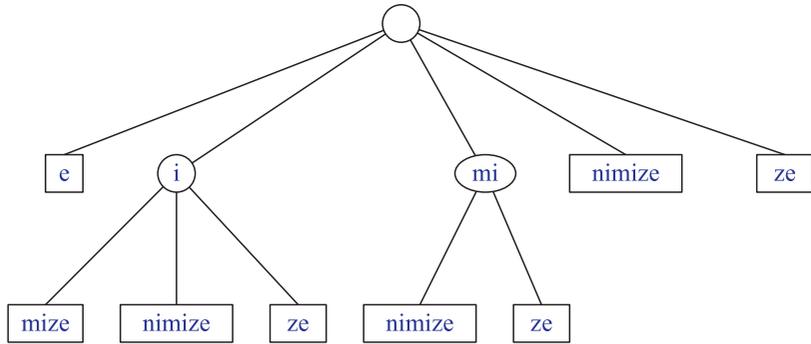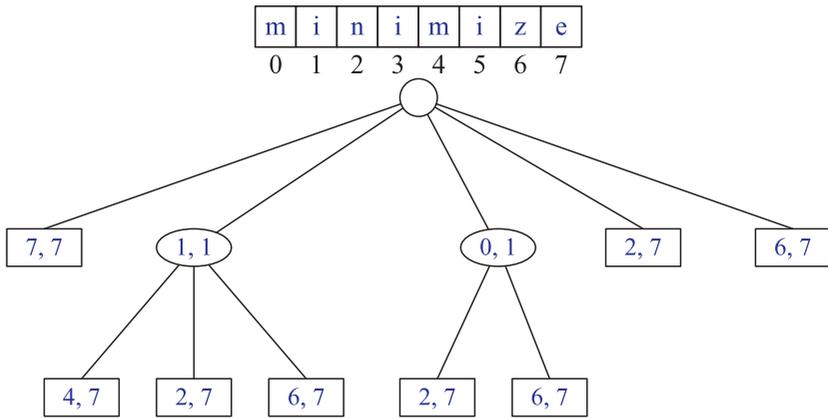
Figure 12.13: (a) Suffix trie T for the string $X = $ ''minimize''. (b) Compact representation of T, where pair (i, j) denotes $X[i..j]$.

Using a Suffix Trie

The suffix trie T for a string X can be used to efficiently perform pattern matching queries on text X. Namely, we can determine whether a pattern P is a substring of X by trying to trace a path associated with P in T. P is a substring of X if and only if such a path can be traced. The search down the trie T assumes that nodes in T store some additional information, with respect to the compact representation of the suffix trie:

> If node v has label (i, j) and Y is the string of length y associated with the path from the root to v (included), then $X[j - y + 1..j] = Y$.

This property ensures that we can easily compute the start index of the pattern in the text when a match occurs.

12.5.4   Search Engines

The World Wide Web contains a huge collection of text documents (Web pages). Information about these pages are gathered by a program called a ***Web crawler***, which then stores this information in a special dictionary database. A Web ***search engine*** allows users to retrieve relevant information from this database, thereby identifying relevant pages on the Web containing given keywords. In this section, we present a simplified model of a search engine.

Inverted Files

The core information stored by a search engine is a dictionary, called an ***inverted index*** or ***inverted file***, storing key-value pairs (w, L), where w is a word and L is a collection of pages containing word w. The keys (words) in this dictionary are called ***index terms*** and should be a set of vocabulary entries and proper nouns as large as possible. The elements in this dictionary are called ***occurrence lists*** and should cover as many Web pages as possible.

We can efficiently implement an inverted index with a data structure consisting of:

1. An array storing the occurrence lists of the terms (in no particular order)
2. A compressed trie for the set of index terms, where each external node stores the index of the occurrence list of the associated term.

The reason for storing the occurrence lists outside the trie is to keep the size of the trie data structure sufficiently small to fit in internal memory. Instead, because of their large total size, the occurrence lists have to be stored on disk.

With our data structure, a query for a single keyword is similar to a word matching query (Section 12.5.1). Namely, we find the keyword in the trie and we return the associated occurrence list.

When multiple keywords are given and the desired output are the pages containing ***all*** the given keywords, we retrieve the occurrence list of each keyword using the trie and return their intersection. To facilitate the intersection computation, each occurrence list should be implemented with a sequence sorted by address or with a dictionary (see, for example, the generic merge computation discussed in Section 11.4).

In addition to the basic task of returning a list of pages containing given keywords, search engines provide an important additional service by ***ranking*** the pages returned by relevance. Devising fast and accurate ranking algorithms for search engines is a major challenge for computer researchers and electronic commerce companies.

12.6 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-12.1 What is the best way to multiply a chain of matrices with dimensions that are 10×5, 5×2, 2×20, 20×12, 12×4, and 4×60? Show your work.

R-12.2 Design an efficient algorithm for the matrix chain multiplication problem that outputs a fully parenthesized expression for how to multiply the matrices in the chain using the minimum number of operations.

R-12.3 List the prefixes of the string $P = $ "aaabbaaa" that are also suffixes of P.

R-12.4 Draw a figure illustrating the comparisons done by brute-force pattern matching for the text "aaabaadaabaaa" and pattern "aabaaa".

R-12.5 Repeat the previous problem for the BM pattern matching algorithm, not counting the comparisons made to compute the last(c) function.

R-12.6 Repeat the previous problem for the KMP pattern matching algorithm, not counting the comparisons made to compute the failure function.

R-12.7 Compute a table representing the last function used in the BM pattern matching algorithm for the pattern string

"the quick brown fox jumped over a lazy cat"

assuming the following alphabet (which starts with the space character):

$$\Sigma = \{\_,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z\}.$$

R-12.8 Assuming that the characters in alphabet Σ can be enumerated and can be used to index arrays, give an $O(m + |\Sigma|)$-time method for constructing the last function from an m-length pattern string P.

R-12.9 Compute a table representing the KMP failure function for the pattern string "cgtacgttcgtac".

R-12.10 Draw a standard trie for the following set of strings:

$$\{abab, baba, ccccc, bbaaaa, caa, bbaacc, cbcc, cbca\}.$$

R-12.11 Draw a compressed trie for the set of strings given in Exercise R-12.10.

R-12.12 Draw the compact representation of the suffix trie for the string

> `"minimize minime"`.

R-12.13 What is the longest prefix of the string `"cgtacgttcgtacg"` that is also a suffix of this string?

R-12.14 Draw the frequency array and Huffman tree for the following string:

> `"dogs do not spot hot pots or cats"`.

R-12.15 Show the longest common subsequence array L for the two strings

$$X \;=\; \text{"skullandbones"}$$
$$Y \;=\; \text{"lullabybabies"}.$$

What is a longest common subsequence between these strings?

Creativity

C-12.1 A native Australian named Anatjari wishes to cross a desert carrying only a single water bottle. He has a map that marks all the watering holes along the way. Assuming he can walk k miles on one bottle of water, design an efficient algorithm for determining where Anatjari should refill his bottle in order to make as few stops as possible. Argue why your algorithm is correct.

C-12.2 Describe an efficient greedy algorithm for making change for a specified value using a minimum number of coins, assuming there are four denominations of coins, called quarters, dimes, nickels, and pennies, with values 25, 10, 5, and 1, respectively. Argue why your algorithm is correct.

C-12.3 Give an example set of denominations of coins so that a greedy change-making algorithm will not use the minimum number of coins.

C-12.4 In the ***art gallery guarding*** problem we are given a line L that represents a long hallway in an art gallery. We are also given a set $X = \{x_0, x_1, \ldots, x_{n-1}\}$ of real numbers that specify the positions of paintings in this hallway. Suppose that a single guard can protect all the paintings within distance at most 1 of his or her position (on both sides). Design an algorithm for finding a placement of guards that uses the minimum number of guards to guard all the paintings with positions in X.

C-12.5 Let P be a convex polygon, a ***triangulation*** of P is an addition of diagonals connecting the vertices of P so that each interior face is a triangle. The ***weight*** of a triangulation is the sum of the lengths of the diagonals.

Assuming that we can compute lengths and add and compare them in constant time, give an efficient algorithm for computing a minimum-weight triangulation of P.

C-12.6 Give an example of a text T of length n and a pattern P of length m that force the brute-force pattern matching algorithm to have a running time that is $\Omega(nm)$.

C-12.7 Give a justification of why the KMPFailureFunction function (Code Fragment 12.7) runs in $O(m)$ time on a pattern of length m.

C-12.8 Show how to modify the KMP string pattern matching algorithm so as to find *every* occurrence of a pattern string P that appears as a substring in T, while still running in $O(n+m)$ time. (Be sure to catch even those matches that overlap.)

C-12.9 Let T be a text of length n, and let P be a pattern of length m. Describe an $O(n+m)$-time method for finding the longest prefix of P that is a substring of T.

C-12.10 Say that a pattern P of length m is a *circular* substring of a text T of length n if there is an index $0 \le i < m$, such that $P = T[n-m+i..n-1] + T[0..i-1]$, that is, if P is a (normal) substring of T or P is equal to the concatenation of a suffix of T and a prefix of T. Give an $O(n+m)$-time algorithm for determining whether P is a circular substring of T.

C-12.11 The KMP pattern matching algorithm can be modified to run faster on binary strings by redefining the failure function as

$$f(j) = \text{the largest } k < j \text{ such that } P[0..k-2]\widehat{p_k} \text{ is a suffix of } P[1..j],$$

where $\widehat{p_k}$ denotes the complement of the kth bit of P. Describe how to modify the KMP algorithm to be able to take advantage of this new failure function and also give a function for computing this failure function. Show that this function makes at most n comparisons between the text and the pattern (as opposed to the $2n$ comparisons needed by the standard KMP algorithm given in Section 12.3.3).

C-12.12 Modify the simplified BM algorithm presented in this chapter using ideas from the KMP algorithm so that it runs in $O(n+m)$ time.

C-12.13 Given a string X of length n and a string Y of length m, describe an $O(n+m)$-time algorithm for finding the longest prefix of X that is a suffix of Y.

C-12.14 Give an efficient algorithm for deleting a string from a standard trie and analyze its running time.

C-12.15 Give an efficient algorithm for deleting a string from a compressed trie and analyze its running time.

C-12.16 Describe an algorithm for constructing the compact representation of a suffix trie, given its noncompact representation, and analyze its running time.

C-12.17 Let T be a text string of length n. Describe an $O(n)$-time method for finding the longest prefix of T that is a substring of the reversal of T.

C-12.18 Describe an efficient algorithm to find the longest palindrome that is a suffix of a string T of length n. Recall that a **palindrome** is a string that is equal to its reversal. What is the running time of your method?

C-12.19 Given a sequence $S = (x_0, x_1, x_2, \ldots, x_{n-1})$ of numbers, describe an $O(n^2)$-time algorithm for finding a longest subsequence $T = (x_{i_0}, x_{i_1}, x_{i_2}, \ldots, x_{i_{k-1}})$ of numbers, such that $i_j < i_{j+1}$ and $x_{i_j} > x_{i_{j+1}}$. That is, T is a longest decreasing subsequence of S.

C-12.20 Define the **edit distance** between two strings X and Y of length n and m, respectively, to be the number of edits that it takes to change X into Y. An edit consists of a character insertion, a character deletion, or a character replacement. For example, the strings "algorithm" and "rhythm" have edit distance 6. Design an $O(nm)$-time algorithm for computing the edit distance between X and Y.

C-12.21 Design a greedy algorithm for making change after someone buys some candy costing x cents and the customer gives the clerk \$1. Your algorithm should try to minimize the number of coins returned.

    a. Show that your greedy algorithm returns the minimum number of coins if the coins have denominations \$0.25, \$0.10, \$0.05, and \$0.01.

    b. Give a set of denominations for which your algorithm may not return the minimum number of coins. Include an example where your algorithm fails.

C-12.22 Give an efficient algorithm for determining if a pattern P is a subsequence (not substring) of a text T. What is the running time of your algorithm?

C-12.23 Let x and y be strings of length n and m respectively. Define $B(i, j)$ to be the length of the longest common substring of the suffix of length i in x and the suffix of length j in y. Design an $O(nm)$-time algorithm for computing all the values of $B(i, j)$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

C-12.24 Raji has just won a contest that allows her to take n pieces of candy out of a candy store for free. Raji is old enough to realize that some candy is expensive, while other candy is relatively cheap, costing much less. The jars of candy are numbered 0, 1, ..., $m - 1$, so that jar j has n_j pieces in it, with a price of c_j per piece. Design an $O(n + m)$-time algorithm that allows Raji to maximize the value of the pieces of candy she takes for her winnings. Show that your algorithm produces the maximum value for Raji.

C-12.25 Let three integer arrays, A, B, and C, be given, each of size n. Given an arbitrary integer x, design an $O(n^2 \log n)$-time algorithm to determine if there exist numbers, a in A, b in B, and c in C, such that $x = a + b + c$.

C-12.26 Give an $O(n^2)$-time algorithm for the previous problem.

Projects

P-12.1 Implement the LCS algorithm and use it to compute the best sequence alignment between some DNA strings that you can get online from GenBank.

P-12.2 Perform an experimental analysis, using documents found on the Internet, of the efficiency (number of character comparisons performed) of the brute-force and KMP pattern matching algorithms for varying-length patterns.

P-12.3 Perform an experimental analysis, using documents found on the Internet, of the efficiency (number of character comparisons performed) of the brute-force and BM pattern matching algorithms for varying-length patterns.

P-12.4 Perform an experimental comparison of the relative speeds of the brute-force, KMP, and BM pattern matching algorithms. Document the time taken for coding up each of these algorithms as well as their relative running times on documents found on the Internet that are then searched using varying-length patterns.

P-12.5 Implement a compression and decompression scheme that is based on Huffman coding.

P-12.6 Create a class that implements a standard trie for a set of ASCII strings. The class should have a constructor that takes as an argument a list of strings, and the class should have a method that tests whether a given string is stored in the trie.

P-12.7 Create a class that implements a compressed trie for a set of ASCII strings. The class should have a constructor that takes as an argument a list of strings, and the class should have a function that tests whether a given string is stored in the trie.

P-12.8 Create a class that implements a prefix trie for an ASCII string. The class should have a constructor that takes as an argument a string and a function for pattern matching on the string.

P-12.9 Implement the simplified search engine described in Section 12.5.4 for the pages of a small Web site. Use all the words in the pages of the site as index terms, excluding stop words such as articles, prepositions, and pronouns.

P-12.10 Implement a search engine for the pages of a small Web site by adding a page-ranking feature to the simplified search engine described in Section 12.5.4. Your page-ranking feature should return the most relevant pages first. Use all the words in the pages of the site as index terms, excluding stop words, such as articles, prepositions, and pronouns.

P-12.11 Write a program that takes two character strings (which could be, for example, representations of DNA strands) and computes their edit distance, showing the corresponding pieces. (See Exercise C-12.20.)

Chapter Notes

The KMP algorithm is described by Knuth, Morris, and Pratt in their journal article [61], and Boyer and Moore describe their algorithm in a journal article published the same year [14]. In their article, however, Knuth *et al.* [61] also prove that the BM algorithm runs in linear time. More recently, Cole [22] shows that the BM algorithm makes at most $3n$ character comparisons in the worst case, and this bound is tight. All of the algorithms discussed above are also discussed in the book chapter by Aho [3], although in a more theoretical framework, including the methods for regular-expression pattern matching. The reader interested in further study of string pattern matching algorithms is referred to the book by Stephen [90] and the book chapters by Aho [3] and Crochemore and Lecroq [26].

The trie was invented by Morrison [79] and is discussed extensively in the classic *Sorting and Searching* book by Knuth [60]. The name "Patricia" is short for "Practical Algorithm to Retrieve Information Coded in Alphanumeric" [79]. McCreight [69] shows how to construct suffix tries in linear time. An introduction to the field of information retrieval, which includes a discussion of search engines for the Web, is provided in the book by Baeza-Yates and Ribeiro-Neto [7].

Chapter

13

Graph Algorithms

Contents

13.1   Graphs

A ***graph*** is a way of representing relationships that exist between pairs of objects. That is, a graph is a set of objects, called vertices, together with a collection of pairwise connections between them. This notion of a "graph" should not be confused with bar charts and function plots, as these kinds of "graphs" are unrelated to the topic of this chapter. Graphs have applications in a host of different domains, including mapping, transportation, electrical engineering, and computer networks.

Viewed abstractly, a ***graph*** G is simply a set V of ***vertices*** and a collection E of pairs of vertices from V, called ***edges***. Thus, a graph is a way of representing connections or relationships between pairs of objects from some set V. Some books use different terminology for graphs and refer to what we call vertices as ***nodes*** and what we call edges as ***arcs***. We use the terms "vertices" and "edges."

Edges in a graph are either ***directed*** or ***undirected***. An edge (u,v) is said to be ***directed*** from u to v if the pair (u,v) is ordered, with u preceding v. An edge (u,v) is said to be ***undirected*** if the pair (u,v) is not ordered. Undirected edges are sometimes denoted with set notation, as $\{u,v\}$, but for simplicity we use the pair notation (u,v), noting that in the undirected case (u,v) is the same as (v,u). Graphs are typically visualized by drawing the vertices as ovals or rectangles and the edges as segments or curves connecting pairs of ovals and rectangles. The following are some examples of directed and undirected graphs.

Example 13.1: *We can visualize collaborations among the researchers of a certain discipline by constructing a graph whose vertices are associated with the researchers themselves, and whose edges connect pairs of vertices associated with researchers who have coauthored a paper or book. (See Figure 13.1.) Such edges are undirected because coauthorship is a* **symmetric** *relation; that is, if A has coauthored something with B, then B necessarily has coauthored something with A.*
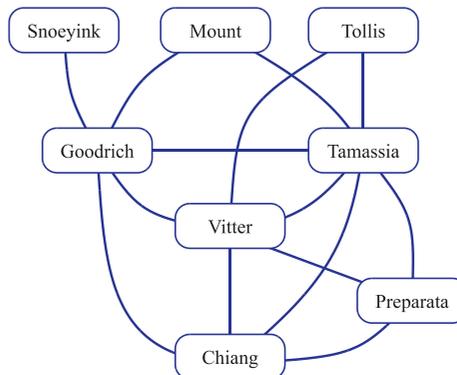
Figure 13.1: Graph of coauthorship among some authors.

Example 13.2: *An object-oriented program can be associated with a graph whose vertices represent the classes defined in the program and whose edges indicate inheritance between classes. There is an edge from a vertex v to a vertex u if the class for v extends the class for u. Such edges are directed because the inheritance relation only goes in one direction (that is, it is* **asymmetric***).*

If all the edges in a graph are undirected, then we say the graph is an ***undirected graph***. Likewise, a ***directed graph***, also called a ***digraph***, is a graph whose edges are all directed. A graph that has both directed and undirected edges is often called a ***mixed graph***. Note that an undirected or mixed graph can be converted into a directed graph by replacing every undirected edge (u, v) by the pair of directed edges (u, v) and (v, u). It is often useful, however, to keep undirected and mixed graphs represented as they are, for such graphs have several applications, such as that of the following example.

Example 13.3: *A city map can be modeled by a graph whose vertices are intersections or dead ends, and whose edges are stretches of streets without intersections. This graph has both undirected edges, which correspond to stretches of two-way streets, and directed edges, which correspond to stretches of one-way streets. Thus, in this way, a graph modeling a city map is a mixed graph.*

Example 13.4: *Physical examples of graphs are present in the electrical wiring and plumbing networks of a building. Such networks can be modeled as graphs, where each connector, fixture, or outlet is viewed as a vertex, and each uninterrupted stretch of wire or pipe is viewed as an edge. Such graphs are actually components of much larger graphs, namely the local power and water distribution networks. Depending on the specific aspects of these graphs that we are interested in, we may consider their edges as undirected or directed, because, in principle, water can flow in a pipe and current can flow in a wire in either direction.*

The two vertices joined by an edge are called the ***end vertices*** (or ***endpoints***) of the edge. If an edge is directed, its first endpoint is its ***origin*** and the other is the ***destination*** of the edge. Two vertices u and v are said to be ***adjacent*** if there is an edge whose end vertices are u and v. An edge is said to be ***incident*** on a vertex if the vertex is one of the edge's endpoints. The ***outgoing edges*** of a vertex are the directed edges whose origin is that vertex. The ***incoming edges*** of a vertex are the directed edges whose destination is that vertex. The ***degree*** of a vertex v, denoted $\deg(v)$, is the number of incident edges of v. The ***in-degree*** and ***out-degree*** of a vertex v are the number of the incoming and outgoing edges of v, and are denoted $\text{indeg}(v)$ and $\text{outdeg}(v)$, respectively.

Example 13.5: *We can study air transportation by constructing a graph G, called*
a flight network *, whose vertices are associated with airports, and whose edges
are associated with flights. (See Figure 13.2.) In graph G, the edges are directed
because a given flight has a specific travel direction (from the origin airport to the
destination airport). The endpoints of an edge e in G correspond respectively to the
origin and destination for the flight corresponding to e. Two airports are adjacent
in G if there is a flight that flies between them, and an edge e is incident upon a
vertex v in G if the flight for e flies to or from the airport for v. The outgoing edges
of a vertex v correspond to the outbound flights from v's airport, and the incoming
edges correspond to the inbound flights to v's airport. Finally, the in-degree of a
vertex v of G corresponds to the number of inbound flights to v's airport, and the
out-degree of a vertex v in G corresponds to the number of outbound flights.*

The definition of a graph refers to the group of edges as a ***collection***, not a ***set***,
thus allowing for two undirected edges to have the same end vertices, and for two
directed edges to have the same origin and the same destination. Such edges are
called ***parallel edges*** or ***multiple edges***. Parallel edges can be in a flight network
(Example 13.5), in which case multiple edges between the same pair of vertices
could indicate different flights operating on the same route at different times of the
day. Another special type of edge is one that connects a vertex to itself. Namely, we
say that an edge (undirected or directed) is a ***self-loop*** if its two endpoints coincide.
A self-loop may occur in a graph associated with a city map (Example 13.3), where
it would correspond to a "circle" (a curving street that returns to its starting point).

With few exceptions, graphs do not have parallel edges or self-loops. Such
graphs are said to be ***simple***. Thus, we can usually say that the edges of a simple
graph are a ***set*** of vertex pairs (and not just a collection). Throughout this chapter,
we assume that a graph is simple unless otherwise specified.
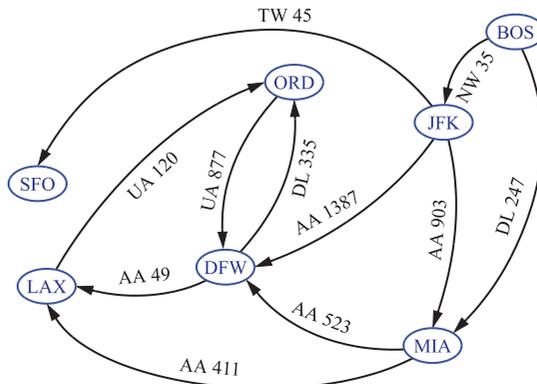
Figure 13.2: Example of a directed graph representing a flight network. The end-
points of edge UA 120 are LAX and ORD; hence, LAX and ORD are adjacent.
The in-degree of DFW is 3, and the out-degree of DFW is 2.

In the propositions that follow, we explore a few important properties of graphs.

Proposition 13.6: *If G is a graph with m edges, then*

$$\sum_{v \text{ in } G} \deg(v) = 2m.$$

Justification: An edge (u, v) is counted twice in the summation above; once by its endpoint u and once by its endpoint v. Thus, the total contribution of the edges to the degrees of the vertices is twice the number of edges. ∎

Proposition 13.7: *If G is a directed graph with m edges, then*

$$\sum_{v \text{ in } G} indeg(v) = \sum_{v \text{ in } G} outdeg(v) = m.$$

Justification: In a directed graph, an edge (u, v) contributes one unit to the out-degree of its origin u and one unit to the in-degree of its destination v. Thus, the total contribution of the edges to the out-degrees of the vertices is equal to the number of edges, and similarly for the out-degrees. ∎

We next show that a simple graph with n vertices has $O(n^2)$ edges.

Proposition 13.8: *Let G be a simple graph with n vertices and m edges. If G is undirected, then* $m \le n(n-1)/2$, *and if G is directed, then* $m \le n(n-1)$.

Justification: Suppose that G is undirected. Since no two edges can have the same endpoints and there are no self-loops, the maximum degree of a vertex in G is $n-1$ in this case. Thus, by Proposition 13.6, $2m \le n(n-1)$. Now suppose that G is directed. Since no two edges can have the same origin and destination, and there are no self-loops, the maximum in-degree of a vertex in G is $n-1$ in this case. Thus, by Proposition 13.7, $m \le n(n-1)$. ∎

A ***path*** is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex. A ***cycle*** is a path with at least one edge that has the same start and end vertices. We say that a path is ***simple*** if each vertex in the path is distinct, and we say that a cycle is ***simple*** if each vertex in the cycle is distinct, except for the first and last one. A ***directed path*** is a path such that all edges are directed and are traversed along their direction. A ***directed cycle*** is similarly defined. For example, in Figure 13.2, (BOS, NW 35, JFK, AA 1387, DFW) is in a directed simple path, and (LAX, UA 120, ORD, UA 877, DFW, AA 49, LAX) is a directed simple cycle. If a path P or cycle C is a simple graph, we may omit the edges in P or C, as these are well defined, in which case P is a list of adjacent vertices and C is a cycle of adjacent vertices.

Example 13.9: *Given a graph G representing a city map (see Example 13.3), we can model a couple driving to dinner at a recommended restaurant as traversing a path though G. If they know the way, and don't accidentally go through the same intersection twice, then they traverse a simple path in G. Likewise, we can model the entire trip the couple takes, from their home to the restaurant and back, as a cycle. If they go home from the restaurant in a completely different way than how they went, not even going through the same intersection twice, then their entire round trip is a simple cycle. Finally, if they travel along one-way streets for their entire trip, we can model their night out as a directed cycle.*

A *subgraph* of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G, respectively. For example, in the flight network of Figure 13.2, vertices BOS, JFK, and MIA, and edges AA 903 and DL 247 form a subgraph. A *spanning subgraph* of G is a subgraph of G that contains all the vertices of the graph G. A graph is *connected* if, for any two vertices, there is a path between them. If a graph G is not connected, its maximal connected subgraphs are called the *connected components* of G. A *forest* is a graph without cycles. A *tree* is a connected forest, that is, a connected graph without cycles. Note that this definition of a tree is somewhat different from the one given in Chapter 7. Namely, in the context of graphs, a tree has no root. Whenever there is ambiguity, the trees of Chapter 7 should be referred to as *rooted trees*, while the trees of this chapter should be referred to as *free trees*. The connected components of a forest are (free) trees. A *spanning tree* of a graph is a spanning subgraph that is a (free) tree.

Example 13.10: *Perhaps the most talked about graph today is the Internet, which can be viewed as a graph whose vertices are computers and whose (undirected) edges are communication connections between pairs of computers on the Internet. The computers and the connections between them in a single domain, like* wiley.com, *form a subgraph of the Internet. If this subgraph is connected, then two users on computers in this domain can send e-mail to one another without having their information packets ever leave their domain. Suppose the edges of this subgraph form a spanning tree. This implies that, if even a single connection goes down (for example, because someone pulls a communication cable out of the back of a computer in this domain), then this subgraph will no longer be connected.*

There are a number of simple properties of trees, forests, and connected graphs.

Proposition 13.11: *Let G be an undirected graph with n vertices and m edges.*

- *If G is connected, then $m \geq n - 1$*
- *If G is a tree, then $m = n - 1$*
- *If G is a forest, then $m \leq n - 1$*

13.1.1 The Graph ADT

In this section, we introduce a simplified graph abstract data type (ADT), which is suitable for undirected graphs, that is, graphs whose edges are all undirected. Additional functions for dealing with directed edges are discussed in Section 13.4.

As an abstract data type, a graph is a collection of elements that are stored at the graph's *positions*—its vertices and edges. Hence, we can store elements in a graph at either its edges or its vertices (or both). The graph ADT defines two types, Vertex and Edge. It also provides two list types for storing lists of vertices and edges, called VertexList and EdgeList, respectively.

Each Vertex object *u* supports the following operations, which provide access to the vertex's element and information regarding incident edges and adjacent vertices.

operator*(): Return the element associated with *u*.

incidentEdges(): Return an edge list of the edges incident on *u*.

isAdjacentTo(*v*): Test whether vertices *u* and *v* are adjacent.

Each Edge object *e* supports the following operations, which provide access to the edge's end vertices and information regarding the edge's incidence relationships.

operator*(): Return the element associated with *e*.

endVertices(): Return a vertex list containing *e*'s end vertices.

opposite(*v*): Return the end vertex of edge *e* distinct from vertex *v*; an error occurs if *e* is not incident on *v*.

isAdjacentTo(*f*): Test whether edges *e* and *f* are adjacent.

isIncidentOn(*v*): Test whether *e* is incident on *v*.

Finally, the full graph ADT consists of the following operations, which provide access to the lists of vertices and edges, and provide functions for modifying the graph.

vertices(): Return a vertex list of all the vertices of the graph.

edges(): Return an edge list of all the edges of the graph.

insertVertex(*x*): Insert and return a new vertex storing element *x*.

insertEdge(*v*, *w*, *x*): Insert and return a new undirected edge with end vertices *v* and *w* and storing element *x*.

eraseVertex(*v*): Remove vertex *v* and all its incident edges.

eraseEdge(*e*): Remove edge *e*.

The VertexList and EdgeList classes support the standard list operations, as described in Chapter 6. In particular, we assume that each provides an iterator (Section 6.2.1), which we call VertexItor and EdgeItor, respectively. They also provide functions begin and end, which return iterators to the beginning and end of their respective lists.

13.2 Data Structures for Graphs

In this section, we discuss three popular ways of representing graphs, which are usually referred to as the *edge list* structure, the *adjacency list* structure, and the *adjacency matrix*. In all three representations, we use a collection to store the vertices of the graph. Regarding the edges, there is a fundamental difference between the first two structures and the latter. The edge list structure and the adjacency list structure only store the edges actually present in the graph, while the adjacency matrix stores a placeholder for every pair of vertices (whether there is an edge between them or not). As we will explain in this section, this difference implies that, for a graph G with n vertices and m edges, an edge list or adjacency list representation uses $O(n+m)$ space, whereas an adjacency matrix representation uses $O(n^2)$ space.

13.2.1 The Edge List Structure

The *edge list* structure is possibly the simplest, though not the most efficient, representation of a graph G. In this representation, a vertex v of G storing an element x is explicitly represented by a vertex object. All such vertex objects are stored in a collection V, such as a vector or node list. If V is a vector, for example, then we naturally think of the vertices as being numbered.

Vertex Objects

The vertex object for a vertex v storing element x has member variables for:
- A copy of x
- The position (or entry) of the vertex-object in collection V

The distinguishing feature of the edge list structure is not how it represents vertices, but the way in which it represents edges. In this structure, an edge e of G storing an element x is explicitly represented by an edge object. The edge objects are stored in a collection E, which would typically be a vector or node list.

Edge Objects

The edge object for an edge e storing element x has member variables for:

- A copy of x
- The vertex positions associated with the endpoint vertices of e
- The position (or entry) of the edge-object in collection E

Visualizing the Edge List Structure

We illustrate an example of the edge list structure for a graph G in Figure 13.3.
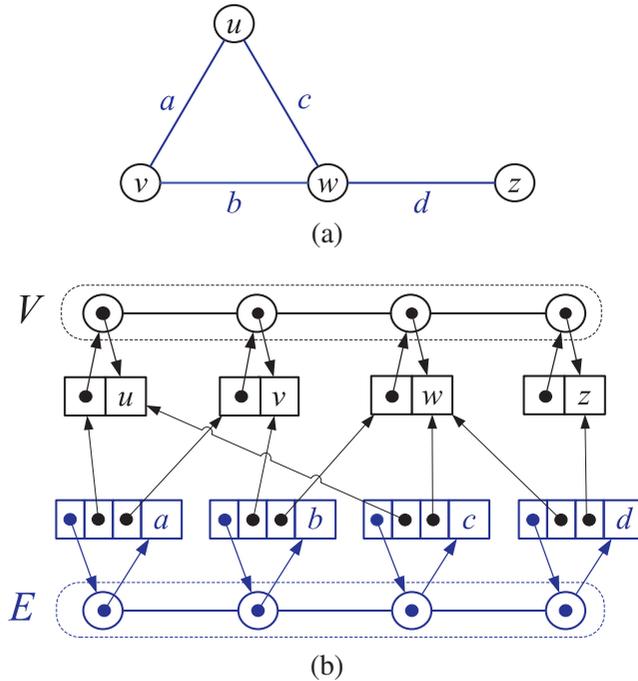
Figure 13.3: (a) A graph G. (b) Schematic representation of the edge list structure for G. We visualize the elements stored in the vertex and edge objects with the element names, instead of with actual references to the element objects.

The reason this structure is called the ***edge list structure*** is that the simplest and most common implementation of the edge collection E is by using a list. Even so, in order to be able to conveniently search for specific objects associated with edges, we may wish to implement E with a dictionary (whose entries store the element as the key and the edge as the value) in spite of our calling this the "edge list." We may also want to implement the collection V by using a dictionary for the same reason. Still, in keeping with tradition, we call this structure the edge list structure.

The main feature of the edge list structure is that it provides direct access from edges to the vertices they are incident upon. This allows us to define simple algorithms for functions $e.\mathsf{endVertices}()$ and $e.\mathsf{opposite}(v)$.

One method that is inefficient for the edge list structure is that of accessing the edges that are incident upon a vertex. Determining this set of vertices requires an exhaustive inspection of all the edge objects in the collection E. That is, in order to determine which edges are incident to a vertex v, we must examine all the edges in the edge list and check, for each one, if it happens to be incident to v. Thus, function v.incidentEdges() runs in time proportional to the number of edges in the graph, not in time proportional to the degree of vertex v. In fact, even to check if two vertices v and w are adjacent by the v.isAdjacentTo(w) function, requires that we search the entire edge collection looking for an edge with end vertices v and w. Moreover, since removing a vertex involves removing all of its incident edges, function eraseVertex also requires a complete search of the edge collection E.

Table 13.1 summarizes the performance of the edge list structure implementation of a graph under the assumption that collections V and E are realized with doubly linked lists (Section 3.3).

Operation	Time
vertices	$O(n)$
edges	$O(m)$
endVertices, opposite	$O(1)$
incidentEdges, isAdjacentTo	$O(m)$
isIncidentOn	$O(1)$
insertVertex, insertEdge, eraseEdge,	$O(1)$
eraseVertex	$O(m)$

Table 13.1: Running times of the functions of a graph implemented with the edge list structure. The space used is $O(n+m)$, where n is the number of vertices and m is the number of edges.

Details for selected functions of the graph ADT are as follows:

- Methods vertices() and edges() are implemented by using the iterators for V and E, respectively, to enumerate the elements of the lists.

- Methods incidentEdges and isAdjacentTo all take $O(m)$ time, since to determine which edges are incident upon a vertex v we must inspect all edges.

- Since the collections V and E are lists implemented with a doubly linked list, we can insert vertices, and insert and remove edges, in $O(1)$ time.

- The update function eraseVertex(v) takes $O(m)$ time, since it requires that we inspect all the edges to find and remove those incident upon v.

Thus, the edge list representation is simple but has significant limitations.

13.2.2 The Adjacency List Structure

The ***adjacency list*** structure for a graph G adds extra information to the edge list structure that supports direct access to the incident edges (and thus to the adjacent vertices) of each vertex. This approach allows us to use the adjacency list structure to implement several functions of the graph ADT much faster than what is possible with the edge list structure, even though both of these two representations use an amount of space proportional to the number of vertices and edges in the graph. The adjacency list structure includes all the structural components of the edge list structure plus the following:

- A vertex object v holds a reference to a collection $I(v)$, called the ***incidence collection*** of v, whose elements store references to the edges incident on v.
- The edge object for an edge e with end vertices v and w holds references to the positions (or entries) associated with edge e in the incidence collections $I(v)$ and $I(w)$.

Traditionally, the incidence collection $I(v)$ for a vertex v is a list, which is why we call this way of representing a graph the ***adjacency list*** structure. The adjacency list structure provides direct access both from the edges to the vertices and from the vertices to their incident edges. We illustrate the adjacency list structure of a graph in Figure 13.4.
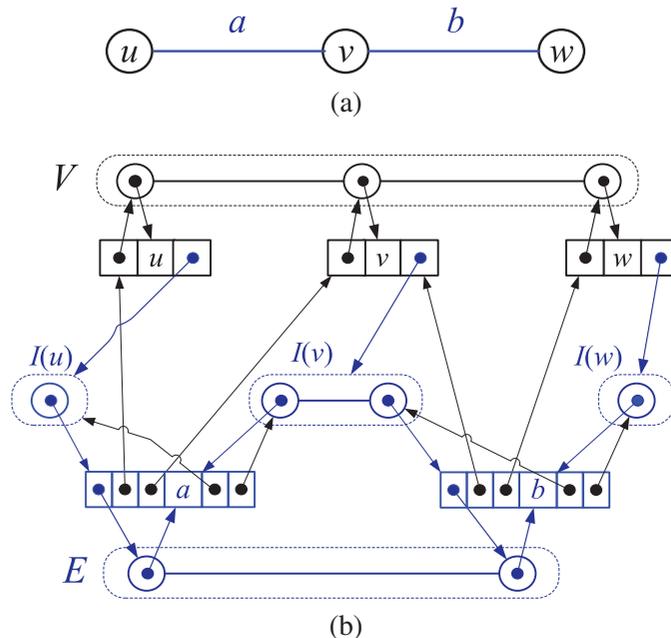
Figure 13.4: (a) A graph G. (b) Schematic representation of the adjacency list structure of G. As in Figure 13.3, we visualize the elements of collections with names.

Performance of the Adjacency List Structure

All of the functions of the graph ADT that can be implemented with the edge list structure in $O(1)$ time can also be implemented in $O(1)$ time with the adjacency list structure, using essentially the same algorithms. In addition, being able to provide access between vertices and edges in both directions allows us to speed up the performance of a number of graph functions by using an adjacency list structure instead of an edge list structure. Table 13.2 summarizes the performance of the adjacency list structure implementation of a graph, assuming that collections V and E and the incidence collections of the vertices are all implemented with doubly linked lists. For a vertex v, the space used by the incidence collection of v is proportional to the degree of v, that is, it is $O(\deg(v))$. Thus, by Proposition 13.6, the space used by the adjacency list structure is $O(n+m)$.

Operation	Time
vertices	$O(n)$
edges	$O(m)$
endVertices, opposite	$O(1)$
v.incidentEdges()	$O(\deg(v))$
v.isAdjacentTo(w)	$O(\min(\deg(v), \deg(w)))$
isIncidentOn	$O(1)$
insertVertex, insertEdge, eraseEdge,	$O(1)$
eraseVertex(v)	$O(\deg(v))$

Table 13.2: Running times of the functions of a graph implemented with the adjacency list structure. The space used is $O(n+m)$, where n is the number of vertices and m is the number of edges.

In contrast to the edge-list way of doing things, the adjacency list structure provides improved running times for the following functions:

- Methods vertices() and edges() are implemented by using the iterators for V and E, respectively, to enumerate the elements of the lists.

- Method v.incidentEdges() takes time proportional to the number of incident vertices of v, that is, $O(\deg(v))$ time.

- Method v.isAdjacentTo(w) can be performed by inspecting either the incidence collection of v or that of w. By choosing the smaller of the two, we get $O(\min(\deg(v), \deg(w)))$ running time.

- Method eraseVertex(v) takes $O(\deg(v))$ time.

13.2.3 The Adjacency Matrix Structure

Like the adjacency list structure, the ***adjacency matrix*** structure of a graph also extends the edge list structure with an additional component. In this case, we augment the edge list with a matrix (a two-dimensional array) A that allows us to determine adjacencies between pairs of vertices in constant time. In the adjacency matrix representation, we think of the vertices as being the integers in the set $\{0, 1, \ldots, n-1\}$ and the edges as being pairs of such integers. This allows us to store references to edges in the cells of a two-dimensional $n \times n$ array A. Specifically, the adjacency matrix representation extends the edge list structure as follows (see Figure 13.5):

- A vertex object v stores a distinct integer i in the range $0, 1, \ldots, n-1$, called the ***index*** of v.
- We keep a two-dimensional $n \times n$ array A such that the cell $A[i, j]$ holds a reference to the edge (v, w), if it exists, where v is the vertex with index i and w is the vertex with index j. If there is no such edge, then $A[i, j] = null$.
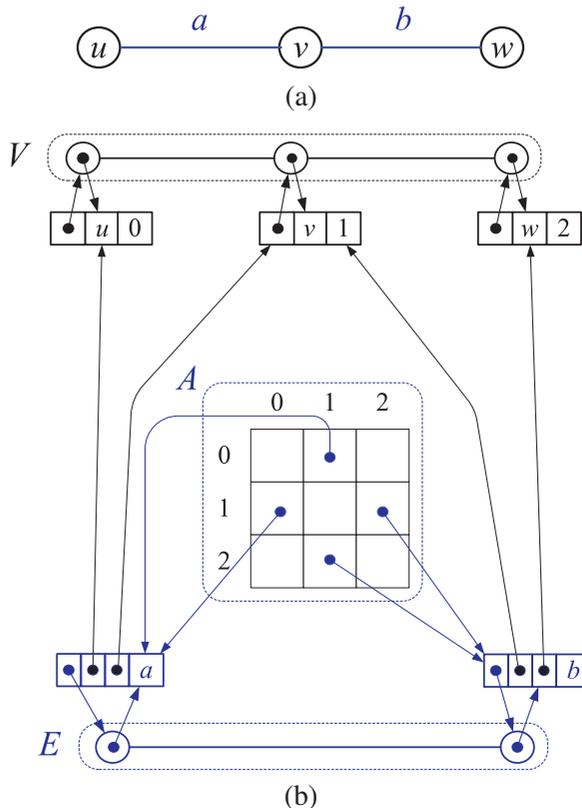
Figure 13.5: (a) A graph G without parallel edges. (b) Schematic representation of the simplified adjacency matrix structure for G.

Performance of the Adjacency Matrix Structure

For graphs with parallel edges, the adjacency matrix representation must be extended so that, instead of having $A[i, j]$ storing a pointer to an associated edge (v, w), it must store a pointer to an incidence collection $I(v, w)$, which stores all the edges from v to w. Since most of the graphs we consider are simple, we do not consider this complication here.

The adjacency matrix A allows us to perform $v.\text{isAdjacentTo}(w)$ in $O(1)$ time. This is done by accessing vertices v and w to determine their respective indices i and j, and then testing whether $A[i, j]$ is null. The efficiency of isAdjacentTo is counteracted by an increase in space usage, however, which is now $O(n^2)$, and in the running time of other functions. For example, function $v.\text{incidentEdges}()$ now requires that we examine an entire row or column of array A and thus runs in $O(n)$ time. Moreover, any vertex insertions or deletions now require creating a whole new array A, of larger or smaller size, respectively, which takes $O(n^2)$ time.

Table 13.3 summarizes the performance of the adjacency matrix structure implementation of a graph. From this table, we observe that the adjacency list structure is superior to the adjacency matrix in space, and is superior in time for all functions except for the isAdjacentTo function.

Operation	Time
vertices	$O(n)$
edges	$O(n^2)$
endVertices, opposite	$O(1)$
isAdjacentTo, isIncidentOn	$O(1)$
incidentEdges	$O(n)$
insertEdge, eraseEdge,	$O(1)$
insertVertex, eraseVertex	$O(n^2)$

Table 13.3: Running times for a graph implemented with an adjacency matrix.

Historically, Boolean adjacency matrices were the first representations used for graphs (so that $A[i, j] = $ **true** if and only if (i, j) is an edge). We should not find this fact surprising, however, for the adjacency matrix has a natural appeal as a mathematical structure (for example, an undirected graph has a symmetric adjacency matrix). The adjacency list structure came later, with its natural appeal in computing due to its faster methods for most algorithms (many algorithms do not use function isAdjacentTo) and its space efficiency.

Most of the graph algorithms we examine run efficiently when acting upon a graph stored using the adjacency list representation. In some cases, however, a trade-off occurs, where graphs with few edges are most efficiently processed with an adjacency list structure, and graphs with many edges are most efficiently processed with an adjacency matrix structure.

13.3 Graph Traversals

Greek mythology tells of an elaborate labyrinth that was built to house the monstrous Minotaur, which was part bull and part man. This labyrinth was so complex that neither beast nor human could escape it. No human, that is, until the Greek hero, Theseus, with the help of the king's daughter, Ariadne, decided to implement a ***graph-traversal*** algorithm. Theseus fastened a ball of thread to the door of the labyrinth and unwound it as he traversed the twisting passages in search of the monster. Theseus obviously knew about good algorithm design, because, after finding and defeating the beast, Theseus easily followed the string back out of the labyrinth to the loving arms of Ariadne. Formally, a ***traversal*** is a systematic procedure for exploring a graph by examining all of its vertices and edges.

13.3.1 Depth-First Search

The first traversal algorithm we consider in this section is ***depth-first search*** (DFS) in an undirected graph. Depth-first search is useful for testing a number of properties of graphs, including whether there is a path from one vertex to another and whether or not a graph is connected.

Depth-first search in an undirected graph G is analogous to wandering in a labyrinth with a string and a can of paint without getting lost. We begin at a specific starting vertex s in G, which we initialize by fixing one end of our string to s and painting s as "visited." The vertex s is now our "current" vertex—call our current vertex u. We then traverse G by considering an (arbitrary) edge (u, v) incident to the current vertex u. If the edge (u, v) leads us to an already visited (that is, painted) vertex v, we immediately return to vertex u. If, on the other hand, (u, v) leads to an unvisited vertex v, then we unroll our string, and go to v. We then paint v as "visited," and make it the current vertex, repeating the computation above. Eventually, we get to a "dead end," that is, a current vertex u such that all the edges incident on u lead to vertices already visited. Thus, taking any edge incident on u causes us to return to u. To get out of this impasse, we roll our string back up, backtracking along the edge that brought us to u, going back to a previously visited vertex v. We then make v our current vertex and repeat the computation above for any edges incident upon v that we have not looked at before. If all of v's incident edges lead to visited vertices, then we again roll up our string and backtrack to the vertex we came from to get to v, and repeat the procedure at that vertex. Thus, we continue to backtrack along the path that we have traced so far until we find a vertex that has yet unexplored edges, take one such edge, and continue the traversal. The process terminates when our backtracking leads us back to the start vertex s, and there are no more unexplored edges incident on s.

This simple process traverses all the edges of *G*. (See Figure 13.6.)
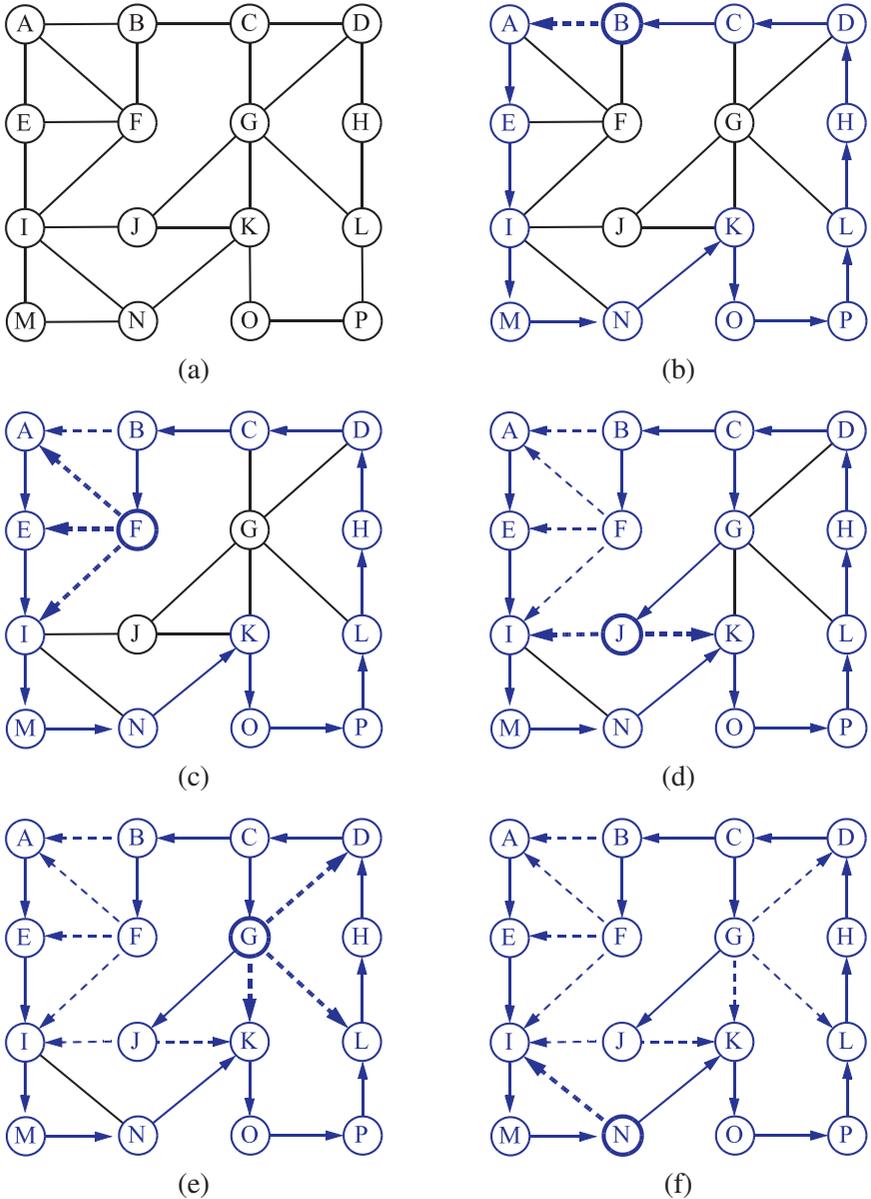
Figure 13.6: Example of depth-first search traversal on a graph starting at vertex *A*. Discovery edges are shown with solid lines and back edges are shown with dashed lines: (a) input graph; (b) path of discovery edges traced from A until back edge (B,A) is hit; (c) reaching F, which is a dead end; (d) after backtracking to C, resuming with edge (C,G), and hitting another dead end, J; (e) after backtracking to G; (f) after backtracking to N.

Discovery Edges and Back Edges

We can visualize a DFS traversal by orienting the edges along the direction in which they are explored during the traversal, distinguishing the edges used to discover new vertices, called *discovery edges*, or *tree edges*, from those that lead to already visited vertices, called *back edges*. (See Figure 13.6(f).) In the analogy above, discovery edges are the edges where we unroll our string when we traverse them, and back edges are the edges where we immediately return without unrolling any string. As we will see, the discovery edges form a spanning tree of the connected component of the starting vertex s. We call the edges not in this tree "back edges" because, assuming that the tree is rooted at the start vertex, each such edge leads back from a vertex in this tree to one of its ancestors in the tree.

The pseudo-code for a DFS traversal starting at a vertex v follows our analogy with string and paint. We use recursion to implement the string analogy, and we assume that we have a mechanism (the paint analogy) to determine if a vertex or edge has been explored or not, and to label the edges as discovery edges or back edges. This mechanism will require additional space and may affect the running time of the algorithm. A pseudo-code description of the recursive DFS algorithm is given in Code Fragment 13.1.

Algorithm DFS(G, v):
    Input: A graph G and a vertex v of G
    Output: A labeling of the edges in the connected component of v as discovery
        edges and back edges
   label v as visited
   for all edges e in v.incidentEdges() **do**
     if edge e is unvisited **then**
       $w \leftarrow e$.opposite(v)
       if vertex w is unexplored **then**
         label e as a discovery edge
         recursively call DFS(G, w)
      else
        label e as a back edge

Code Fragment 13.1: The DFS algorithm.

There are a number of observations that we can make about the depth-first search algorithm, many of which derive from the way the DFS algorithm partitions the edges of the undirected graph G into two groups, the discovery edges and the back edges. For example, since back edges always connect a vertex v to a previously visited vertex u, each back edge implies a cycle in G, consisting of the discovery edges from u to v plus the back edge (u, v).

Proposition 13.12: *Let G be an undirected graph on which a DFS traversal starting at a vertex s has been performed. Then the traversal visits all vertices in the connected component of s, and the discovery edges form a spanning tree of the connected component of s.*

Justification: Suppose there is at least one vertex *v* in *s*'s connected component not visited, and let *w* be the first unvisited vertex on some path from *s* to *v* (we may have *v* = *w*). Since *w* is the first unvisited vertex on this path, it has a neighbor *u* that was visited. But when we visited *u*, we must have considered the edge (u, w); hence, it cannot be correct that *w* is unvisited. Therefore, there are no unvisited vertices in *s*'s connected component.

Since we only mark edges when we go to unvisited vertices, we will never form a cycle with discovery edges, that is, discovery edges form a tree. Moreover, this is a spanning tree because, as we have just seen, the depth-first search visits each vertex in the connected component of *s*.   ■

In terms of its running time, depth-first search is an efficient method for traversing a graph. Note that DFS is called exactly once on each vertex, and that every edge is examined exactly twice, once from each of its end vertices. Thus, if n_s vertices and m_s edges are in the connected component of vertex *s*, a DFS starting at *s* runs in $O(n_s + m_s)$ time, provided the following conditions are satisfied:

- The graph is represented by a data structure such that creating and iterating through the list generated by *v*.incidentEdges() takes $O(\text{degree}(v))$ time, and *e*.opposite(*v*) takes $O(1)$ time. The adjacency list structure is one such structure, but the adjacency matrix structure is not.
- We have a way to "mark" a vertex or edge as explored, and to test if a vertex or edge has been explored in $O(1)$ time. We discuss ways of implementing DFS to achieve this goal in the next section.

Given the assumptions above, we can solve a number of interesting problems.

Proposition 13.13: *Let G be a graph with n vertices and m edges represented with an adjacency list. A DFS traversal of G can be performed in $O(n + m)$ time, and can be used to solve the following problems in $O(n + m)$ time:*

- *Testing whether G is connected*
- *Computing a spanning tree of G, if G is connected*
- *Computing the connected components of G*
- *Computing a path between two given vertices of G, if it exists*
- *Computing a cycle in G, or reporting that G has no cycles*

The justification of Proposition 13.13 is based on algorithms that use slightly modified versions of the DFS algorithm as subroutines.

13.3.2 Implementing Depth-First Search

As we have mentioned above, the data structure we use to represent a graph impacts the performance of the DFS algorithm. For example, an adjacency list can be used to yield a running time of $O(n + m)$ for traversing a graph with n vertices and m edges. Using an adjacency matrix, on the other hand, would result in a running time of $O(n^2)$, since each of the n calls to the incidentEdges function would take $O(n)$ time. If the graph is **dense**, that is, it has close to $O(n^2)$ edges, then the difference between these two choices is minor, as they both would run in $O(n^2)$ time. But if the graph is **sparse**, that is, it has close to $O(n)$ edges, then the adjacency matrix approach would be much slower than the adjacency list approach.

Another important implementation detail deals with the way vertices and edges are represented. In particular, we need to have a way of marking vertices and edges as visited or not. There are two simple solutions, but each has drawbacks.

- We can build our vertex and edge objects to contain a *visited* field, which can be used by the DFS algorithm for marking. This approach is quite simple, and supports constant-time marking and unmarking, but it assumes that we are designing our graph with DFS in mind, which will not always be valid. Furthermore, this approach needlessly restricts DFS to graphs with vertices having a *visited* field. Thus, if we want a generic DFS algorithm that can take any graph as input, this approach has limitations.
- We can use an auxiliary hash table to store all the explored vertices and edges during the DFS algorithm. This scheme is general, in that it does not require any special fields in the positions of the graph. But this approach does not achieve worst-case constant time for marking and unmarking of vertices edges. Instead, such a hash table only supports the mark (insert) and test (find) operations in constant *expected* time (see Section 9.2).

Fortunately, there is a middle ground between these two extremes.

The Decorator Pattern

Marking the explored vertices in a DFS traversal is an example of the **decorator** software engineering design pattern. This pattern is used to add **decorations** (also called **attributes**) to existing objects. Each decoration is identified by a **key** identifying this decoration and by a **value** associated with the key. The use of decorations is motivated by the need of some algorithms and data structures to add extra variables, or temporary scratch data, to objects that do not normally have such variables. Hence, a decoration is a key-value pair that can be dynamically attached to an object. In our DFS example, we would like to have "decorable" vertices and edges with a *visited* decoration and a Boolean value.

Making Graph Vertices Decorable

We can realize the decorator pattern for any position by allowing it to be decorated. This allows us to add labels to vertices and edges, without requiring that we know in advance the kinds of labels that we will need. We say that an object is ***decorable*** if it supports the following functions:

$set(a,x)$: Set the value of attribute a to x.

$get(a)$: Return the value of attribute a.

We assume that Vertex and Edge objects of our graph ADT are decorable, where attribute keys are strings and attribute values are pointers to a generic object class, called Object.

As an example of how this works, suppose that we want to mark vertices as being either visited or not visited by a search procedure. To implement this, we could create two new instances of the Object class, and store pointers to these objects in two variables, say *yes* and *no*. The values of these objects are unimportant to us—all we require is the ability to distinguish between them. Let v be an object of type Decorator. To indicate that v is visited we invoke v.set("visited", *yes*) and to indicate that it was not visited we invoke v.set("visited", *no*). In order to test the value of this decorator, we invoke v.get("visited") and test to see whether the result is *yes* or *no*. This is shown in the following code fragment.

```
Object* yes = new Object;          // decorator values
Object* no = new Object;
Decorator v;                       // a decorable object
// ...
v.set("visited", yes);             // set "visited" attribute
// ...
if (v.get("visited") == yes) cout << "v was visited";
else cout << "v was not visited";
```

In Code Fragment 13.2, we present a C++ implementation of class Decorator. It works by creating an STL map (Section 9.1.3), whose keys are strings and whose values are of type Object*.

```
class Decorator {
private:                                    // member data
  std::map<string,Object*> map;             // the map
public:
  Object* get(const string& a)              // get value of attribute
    { return map[a]; }
  void set(const string& a, Object* d)      // set value
    { map[a] = d; }
};
```

Code Fragment 13.2: A C++ implementation of class Decorator.

DFS Traversal using Decorable Positions

Using decorable positions, the complete DFS traversal algorithm can be described in more detail, as shown in Code Fragment 13.3. We create an attribute named "status" in which to record the status information about vertices and edges. This attribute may take on one of four possible values: *unvisited*, *visited*, *discovery*, and *back*. Initially, all attribute values are assumed to have been set to *unvisited*. On termination, edges will be labeled as *discovery* or *back*, depending on whether they are discovery edges or back edges.

Algorithm DFS(*G*, *v*):
    Input: A graph *G* with decorable vertices and edges, a vertex *v* of *G*, such that all vertices and edges have been decorated with the status value of *unvisited*
    Output: A decoration of the vertices of the connected component of *v* with the value *visited* and of the edges in the connected component of *v* with values *discovery* and *back*, according to a depth-first search traversal of *G*
  v.set("status", *visited*)
  for all edges *e* in *v*.incidentEdges() **do**
    if *e*.get("status") = *unvisited* **then**
      w ← *e*.opposite(*v*)
      if *w*.get("status") = *unvisited* **then**
        e.set("status", *discovered*)
        DFS(*G*, *w*)
      else
        e.set("status", *back*)

Code Fragment 13.3: DFS on a graph with decorable edges and vertices.

13.3.3   A Generic DFS Implementation in C++

In this section, we present a C++ implementation of a generic depth-first search traversal by means of a class, called DFS. This class defines a recursive member function, dfsTraversal(*v*), which performs a DFS traversal of the graph starting at vertex *v*. The behavior of the traversal function can be specialized for a particular application by redefining a number of functions, which are invoked in response to various events that arise in the traversal.

We assume that the vertices and edges are decorable positions and use decorations to determine whether vertices and edges have been visited. The generic DFS class contains the following virtual functions, which may be overridden by concrete subclasses to affect a desired behavior:

- startVisit(*v*): called at the start of the visit of *v*

- traverseDiscovery(e, v): called when a discovery edge *e* out of *v* is traversed
- traverseBack(e, v): called when a back edge *e* out of *v* is traversed
- isDone(): called to determine whether to end the traversal early
- finishVisit(v): called when we are finished exploring from *v*

The class DFS is presented in Code Fragment 13.4. The class is templated with the graph type *G*. It begins with a number of convenience type definitions to allow us to access elements of the underlying graph type more succinctly. We have omitted some of the type definitions, such as VertexList, EdgeList, VertexItor, and EdgeItor. This is followed by the member data of the class, which consists of a reference to the graph, the vertex at which the depth-first traversal begins, and two decorator objects *yes* and *no*, which will be used in decorating vertices and edges. Their actual values are irrelevant, as long as we can distinguish one from the other.

```
template <typename G>
class DFS {                                  // generic DFS
protected:                                    // local types
  typedef typename G::Vertex Vertex;          // vertex position
  typedef typename G::Edge Edge;              // edge position
  // ...insert other typename shortcuts here
protected:                                    // member data
  const G& graph;                             // the graph
  Vertex start;                               // start vertex
  Object *yes, *no;                           // decorator values
protected:                                    // member functions
  DFS(const G& g);                            // constructor
  void initialize();                          // initialize a new DFS
  void dfsTraversal(const Vertex& v);         // recursive DFS utility
                                              // overridden functions
  virtual void startVisit(const Vertex& v) {} // arrived at v
                                              // discovery edge e
  virtual void traverseDiscovery(const Edge& e, const Vertex& from) {}
                                              // back edge e
  virtual void traverseBack(const Edge& e, const Vertex& from) {}
  virtual void finishVisit(const Vertex& v) {} // finished with v
  virtual bool isDone() const { return false; } // finished?
  // ...insert marking utilities here
};
```

Code Fragment 13.4: A generic implementation of depth-first search.

The class's member functions are all protected. They are invoked only by public members of the derived subclasses. These member functions include a constructor, an initialization function, and the generic DFS traversal function. There are a number of virtual functions corresponding to each of the above operations, which are overridden by subclasses of class DFS.

We specify whether vertices and edges have been visited during the traversal through calls to the marking utility functions visit, unvisit, and isVisited, which are shown in Code Fragment 13.5.

```
protected:                                    // marking utilities
  void visit(const Vertex& v)        { v.set("visited", yes); }
  void visit(const Edge& e)          { e.set("visited", yes); }
  void unvisit(const Vertex& v)      { v.set("visited", no); }
  void unvisit(const Edge& e)        { e.set("visited", no); }
  bool isVisited(const Vertex& v)    { return v.get("visited") == yes; }
  bool isVisited(const Edge& e)      { return e.get("visited") == yes; }
```

Code Fragment 13.5: Vertex and edge marking utilities, which are part of DFS.

In Code Fragment 13.6, we present the class constructor and a function that performs initializations before the DFS traversal is performed. (We present the external member functions using the condensed notation, which we introduced in Section 9.2.7.) The constructor initializes the graph reference and allocates the generic objects *yes* and *no*, which are used by the marking utilities. (Eventually, these are deallocated by the class destructor, which we have omitted.) The initialization function marks all vertices and edges as "unvisited."

```
/* DFS⟨G⟩ :: */                               // constructor
  DFS(const G& g)
    : graph(g), yes(new Object), no(new Object) {}

/* DFS⟨G⟩ :: */                               // initialize a new DFS
  void initialize() {
    VertexList verts = graph.vertices();
    for (VertexItor pv = verts.begin(); pv != verts.end(); ++pv)
      unvisit(*pv);                           // mark vertices unvisited
    EdgeList edges = graph.edges();
    for (EdgeItor pe = edges.begin(); pe != edges.end(); ++pe)
      unvisit(*pe);                           // mark edges unvisited
  }
```

Code Fragment 13.6: The class constructor for DFS and the initialization function.

The recursive DFS traversal function is presented in Code Fragment 13.7. The function follows the same structure as presented in Code Fragment 13.3. Note, however, the introduction of calls to the virtual functions startVisit, isDone, traverseDiscovery, traverseBack, and finishVisit. These have not yet been specified, but their definitions determine the concrete behavior of the traversal process.

```
/* DFS⟨G⟩ :: */                                           // recursive traversal
  void dfsTraversal(const Vertex& v) {
    startVisit(v);    visit(v);                           // visit v and mark visited
    EdgeList incident = v.incidentEdges();
    EdgeItor pe = incident.begin();
    while (!isDone() && pe != incident.end()) {           // visit v's incident edges
      Edge e = *pe++;
      if (!isVisited(e)) {                                // discovery edge?
        visit(e);                                         // mark it visited
        Vertex w = e.opposite(v);                         // get opposing vertex
        if (!isVisited(w)) {                              // unexplored?
          traverseDiscovery(e, v);                        // let's discover it
          if (!isDone()) dfsTraversal(w);                 // continue traversal
        }
        else traverseBack(e, v);                          // process back edge
      }
    }
    if (!isDone()) finishVisit(v);                        // finished with v
  }
```

Code Fragment 13.7: The function dfsTraversal, which implements a generic DFS traversal.

Using the Template Method Pattern for DFS

In the remainder of this section, we illustrate a number of concrete applications of our generic DFS traversal. In order to do anything interesting, we must extend DFS and redefine some of its auxiliary functions. This is an example of the template method pattern (see Section 7.3.7), in which a generic computation mechanism is specialized by providing concrete implementations of certain generic steps.

Our first example is the class Components, which counts the number of connected components of a graph. The class is presented in Code Fragment 13.8. Observe that this class is derived from DFS, and so inherits its members. The Components class contains a single data member, which is a counter *nComponents* of the number of connected components it has encountered.

```
template <typename G>
class Components : public DFS<G> {            // count components
private:
  int nComponents;                            // num of components
public:
  Components(const G& g): DFS<G>(g) {}        // constructor
  int operator()();                           // count components
};
```

Code Fragment 13.8: The class Components, which extends the DFS class in order to count the number of components of a graph by overloading the "()" operator.

The class provides a simple constructor, which simply invokes the constructor for the base class by passing it the graph. In order to define the component counting function, we have overloaded the "()" operator. This overloaded function returns the number of components. This means that, given a graph G, we can declare a Components objects and invoke it as follows:

```
Components components(G);    // declare a Components object
int nc = components();       // compute the number of components
```

The function that computes the number of connected components is shown in Code Fragment 13.9. After initializing (by invoking the DFS member function initialize and setting the component counter to zero), it iterates through the vertices of the graph. Whenever it finds an unvisited vertex, it invokes the DFS traversal on this vertex and increments the component counter. The DFS traversal returns only after every vertex of this connected component has been visited (Proposition 13.12). Therefore, any unvisited vertex must lie in a different component. By repeating this on every unvisited vertex, eventually every connected component will be found and counted.

```
/* Components⟨G⟩ :: */                        // count components
  int operator()() {
    initialize();                             // initialize DFS
    nComponents = 0;                          // init vertex count
    VertexList verts = graph.vertices();
    for (VertexItor pv = verts.begin(); pv != verts.end(); ++pv) {
      if (!isVisited(*pv)) {                  // not yet visited?
        dfsTraversal(*pv);                    // visit
        nComponents++;                        // one more component
      }
    }
    return nComponents;
  }
```

Code Fragment 13.9: The overloaded () operator for class Components, which computes the number of connected components of a graph.

Our next example is class FindPath, which finds a path between a given source vertex and target vertex. The class is presented in Code Fragment 13.10. The class's member data consists of the vertex list forming the path (*path*), the target vertex (*target*), and a boolean variable indicating whether the search is complete (*done*). Like before, the principal path finding function has been defined by overloading the "()" operator. This function is given the source vertex s and target vertex t, and returns the path as a list of vertices from s to t. If there is no such path, an empty list is returned. Also like before, to use this class, we first create a new FindPath object, say *findPath*, and then we invoke *findPath*(s,t), for the desired source vertex s and target vertex t.

```
template <typename G>
class FindPath : public DFS<G> {                    // find a path by DFS
private:                                             // local data
  VertexList path;                                  // the path
  Vertex target;                                    // the target vertex
  bool done;                                        // is target found?
protected:                                          // overridden functions
  void startVisit(const Vertex& v);                 // visit vertex
  void finishVisit(const Vertex& v);                // finished with vertex
  bool isDone() const;                              // done yet?
public:
  FindPath(const G& g) : DFS<G>(g) {}               // constructor
                                                    // find path from s to t
  VertexList operator()(const Vertex& s, const Vertex& t);
};
```

Code Fragment 13.10: The class FindPath, which extends the DFS class in order to compute a path from source vertex *s* to target vertex *t*.

The path function is presented in Code Fragment 13.11. After initializing search and the member data, it invokes the recursive DFS traversal. On termination, the vertex list containing the path is returned.

```
/* FindPath⟨G⟩ :: */                                // find path from s to t
  VertexList operator()(const Vertex& s, const Vertex& t) {
    initialize();                                   // initialize DFS
    path.clear();                                   // clear the path
    target = t;                                     // save the target
    done = false;
    dfsTraversal(s);                                // traverse starting at s
    return path;                                    // return the path
  }
```

Code Fragment 13.11: The overloaded () operator for class FindPath, which returns a path from *s* to *t*.

The approach is to perform a depth-first search traversal beginning at the source vertex. We maintain the path of discovery edges from the source to the current vertex. When we encounter an unexplored vertex, we add it to the end of the path. This is processed by overriding the function startVisit. When we finish processing a vertex, we remove it from the path. This is done by overriding function finishVisit. Thus, at any point, the vertex list consists of vertices along the path of the DFS tree from the source to the current vertex. The traversal is terminated when the target vertex is encountered. This is done by setting the boolean flag *done*. We override the function isDone to check for this event. These overridden functions are shown in Code Fragment 13.12.

```
/* FindPath⟨G⟩ :: */                        // visit vertex
  void startVisit(const Vertex& v) {
    path.push_back(v);                      // insert into path
    if (v == target) done = true;           // reached target vertex
  }
/* FindPath⟨G⟩ :: */                        // finished with vertex
  void finishVisit(const Vertex& v)
    { if (!done) path.pop_back(); }         // remove last vertex
/* FindPath⟨G⟩ :: */                        // done yet?
  bool isDone() const
    { return done; }
```

Code Fragment 13.12: The overridden functions used by class FindPath.

Our final example is class FindCycle, which finds a cycle in the connected component of a given start vertex *s*. (The cycle need not contain *s*.) The class is given in Code Fragment 13.13. Its member data consists of the edge list containing the cycle (*cycle*), the first vertex of the cycle (*cycleStart*), and a boolean variable that indicates whether the search is complete (*done*). The cycle function is given by overloading the "()" operator and passing in the start vertex *s*. It returns the cycle as a list of edges. If there is no such cycle, an empty list is returned.

```
template <typename G>
class FindCycle : public DFS<G> {
private:                                    // local data
  EdgeList cycle;                           // cycle storage
  Vertex cycleStart;                        // start of cycle
  bool done;                                // cycle detected?
protected:                                  // overridden functions
  void traverseDiscovery(const Edge& e, const Vertex& from);
  void traverseBack(const Edge& e, const Vertex& from);
  void finishVisit(const Vertex& v);        // finished with vertex
  bool isDone() const;                      // done yet?
public:
  FindCycle(const G& g) : DFS<G>(g) {}      // constructor
  EdgeList operator()(const Vertex& s);     // find a cycle
};
```

Code Fragment 13.13: The class FindCycle, which extends the DFS class in order to compute a cycle in the connected component of a given start vertex *s*.

The cycle finding function is presented in Code Fragment 13.14. After initializing search and the member data, it invokes the recursive DFS traversal. As we show below, upon termination, the edge list consists of an initial prefix of edges from *s* to the vertex *cycleStart*, followed by the edges of the cycle. We traverse the edges of the cycle and remove each until reaching the first edge that is incident to *cycleStart*.

```
/* FindCycle⟨G⟩ :: */                              // find a cycle
  EdgeList operator()(const Vertex& s) {
    initialize();                                  // initialize DFS
    cycle = EdgeList(); done = false;              // initialize members
    dfsTraversal(s);                               // do the search
    if (!cycle.empty() && s != cycleStart) {       // found a cycle?
      EdgeItor pe = cycle.begin();
      while (pe != cycle.end()) {                  // search for prefix
        if ((pe++)->isIncidentOn(cycleStart)) break; // last edge of prefix?
      }
      cycle.erase(cycle.begin(), pe);              // remove prefix
    }
    return cycle;                                  // return the cycle
  }
```

Code Fragment 13.14: The function of class FindCycle, which computes the cycle.

Our approach is based on performing a DFS traversal and storing the discovery edges in the edge list. As shown in Code Fragment 13.15, in traverseDiscovery, we push the current edge onto the edge list. In the function traverseBack, when we encounter a back edge we complete a cycle and set the boolean flag *done* to true to indicate that the cycle has been detected. We also set the variable *cycleStart* to the vertex on the opposite side of the back edge. When backing out of a vertex, as shown in the function finishVisit, we pop the most recent edge off the edge list, unless the cycle has been found.

```
/* FindCycle⟨G⟩ :: */                              // discovery edge e
  void traverseDiscovery(const Edge& e, const Vertex& from)
    { if (!done) cycle.push_back(e); }             // add edge to list
/* FindCycle⟨G⟩ :: */                              // back edge e
  void traverseBack(const Edge& e, const Vertex& from) {
    if (!done) {                                   // no cycle yet?
      done = true;                                 // cycle now detected
      cycle.push_back(e);                          // insert final edge
      cycleStart = e.opposite(from);               // save starting vertex
    }
  }
/* FindCycle⟨G⟩ :: */                              // finished with vertex
  void finishVisit(const Vertex& v) {
    if (!cycle.empty() && !done)                   // not building a cycle?
      cycle.pop_back();                            // remove this edge
  }
/* FindCycle⟨G⟩ :: */                              // done yet?
  bool isDone() const
    { return done; }
```

Code Fragment 13.15: The overridden functions used by class FindCycle.

13.3.4 Polymorphic Objects and Decorator Values ⋆

Programming decorations that support multiple value types poses an interesting problem in C++. To illustrate the problem, let us suppose that we wish to implement an algorithm that associates a string with a persons name and an integer containing the persons current age with each vertex of a social network. Given a vertex *v*, we would like to associate it with two decorators, one for name and one for age.

```
v.set("name", "Bob");
v.set("age", 23);
```

C++'s strong type checking does not allow this, however, since we must specify the type of the attribute value, either string or int, but not both.

To solve this problem, we create a ***polymorphic*** value type (Section 2.2.2). We define a generic class, called Object, and we derive subclasses from this. Each subclass is specialized to store a single value of a particular type, for example, bool, char, int, or string. To make this more concrete, let us consider a simple example for just two types, int and string. It is straightforward to generalize this to other types, even user-defined types.

Our generic Object class is shown in Code Fragment 13.16. It has no data members, but it supports two member functions, intValue and stringValue. The first returns the value from an integer subclass and the second returns the value from a string subclass. An attempt to extract a string value from an integer object or an integer value from a string argument results in an exception being thrown.

```
class Object {                                        // generic object
public:
  int     intValue()    const throw(BadCast);
  string  stringValue() const throw(BadCast);
};
```

Code Fragment 13.16: A generic class, called Object, for storing a polymorphic object of type int or string.

Next, we derive two concrete subclasses from Object. The first, called Integer, stores a single integer, and the second, called String, stores a single STL string. These are shown in Code Fragment 13.17. In addition to a simple constructor, they each provide a member function getValue, which returns the stored value.

Finally, we define the member functions intValue and stringValue of class Object. We show only intValue in Code Fragment 13.18 (stringValue is analogous). This function assumes that the underlying object is a pointer to an Integer. It attempts to dynamically cast itself to an Integer pointer. If successful, it returns the resulting integer value. If not, an exception is thrown.

```
class Integer : public Object {          class String : public Object {
private:                                 private:
  int value;                               string value;
public:                                  public:
  Integer(int v = 0) : value(v) { }        String(string v = "") : value(v) { }
  int getValue() const                     string getValue() const
      { return value; }                        { return value; }
};                                       };
```

Code Fragment 13.17: Concrete subclasses, Integer and String, for storing a single integer and a single string, respectively.

```
int Object::intValue() const throw(BadCast) {       // cast to Integer
  const Integer* p = dynamic_cast<const Integer*>(this);
  if (p == NULL) throw BadCast("Illegal attempt to cast to Integer");
  return p->getValue();
}
```

Code Fragment 13.18: The member function intValue of class Object, which returns the underlying integer value.

To show how to apply this useful polymorphic object, let us return to our earlier example. Recall that *v* is a vertex to which we want to assign two attributes, a name and an age. We create new entities, the first of type String and the second of type Integer. We initialize each with the desired value. Because these are subclasses of Object, we may store these entities as decorators as shown in Code Fragment 13.19.

```
Decorator v;                                   // a decorable object
v.set("name", new String("Bob"));              // store name as "Bob"
v.set("age", new Integer(23));                 // store age as 23
// ...
string n = v.get("name")->stringValue();       // n = "Bob"
int a = v.get("age")->intValue();              // a = 23
```

Code Fragment 13.19: Example use of Object with a polymorphic dictionary.

When we extract the values of these decorators, we make use of the fact that we know that the name is a string, and the age is an integer. Thus, we may apply the appropriate function, stringValue or intValue, to extract the desired attribute value. This example shows the usefulness of polymorphic behavior of objects in C++.

13.3.5 Breadth-First Search

In this section, we consider a different traversal algorithm, called ***breadth-first search*** (BFS). Like DFS, BFS traverses a connected component of a graph, and in so doing it defines a useful spanning tree. BFS is less "adventurous" than DFS, however. Instead of wandering the graph, BFS proceeds in rounds and subdivides the vertices into ***levels***. BFS can also be thought of as a traversal using a string and paint, with BFS unrolling the string in a more conservative manner.

BFS starts at vertex s, which is at level 0 and defines the "anchor" for our string. In the first round, we let out the string the length of one edge and we visit all the vertices we can reach without unrolling the string any farther. In this case, we visit, and paint as "visited," the vertices adjacent to the start vertex s—these vertices are placed into level 1. In the second round, we unroll the string the length of two edges and we visit all the new vertices we can reach without unrolling our string any farther. These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2, and so on. The BFS traversal terminates when every vertex has been visited.

Pseudo-code for a BFS starting at a vertex s is shown in Code Fragment 13.20. We use auxiliary space to label edges, mark visited vertices, and store collections associated with levels. That is, the collections L_0, L_1, L_2, and so on, store the vertices that are in level 0, level 1, level 2, and so on. These collections could, for example, be implemented as queues. They also allow BFS to be nonrecursive.

Algorithm BFS(s):

    initialize collection L_0 to contain vertex s
    $i \leftarrow 0$
    while L_i is not empty **do**
      create collection L_{i+1} to initially be empty
      for all vertices v in L_i **do**
        for all edges e in v.incidentEdges() **do**
          if edge e is unexplored **then**
            $w \leftarrow e$.opposite(v)
            if vertex w is unexplored **then**
              label e as a discovery edge
              insert w into L_{i+1}
            else
              label e as a cross edge
    $i \leftarrow i+1$

          Code Fragment 13.20: The BFS algorithm.

We illustrate a BFS traversal in Figure 13.7.
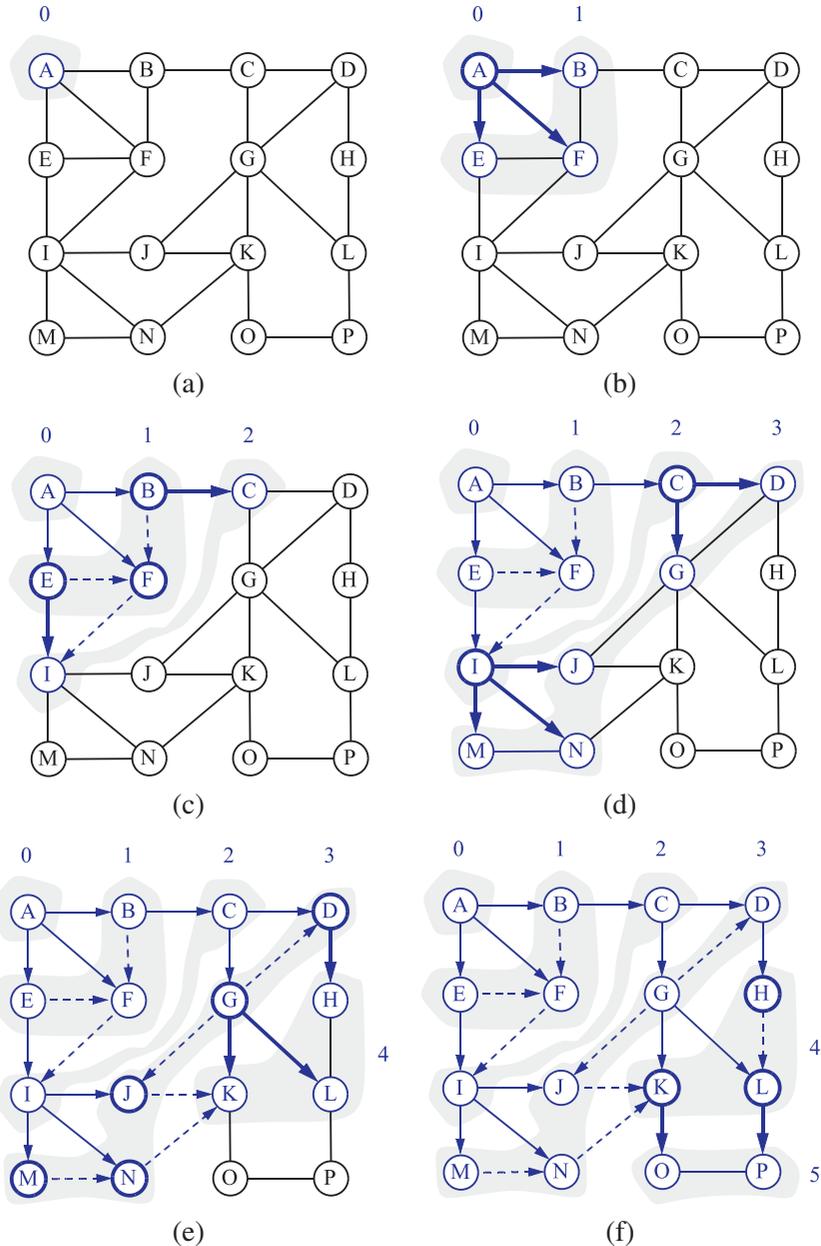
Figure 13.7: Example of breadth-first search traversal, where the edges incident on a vertex are explored by the alphabetical order of the adjacent vertices. The discovery edges are shown with solid lines and the cross edges are shown with dashed lines: (a) graph before the traversal; (b) discovery of level 1; (c) discovery of level 2; (d) discovery of level 3; (e) discovery of level 4; (f) discovery of level 5.

One of the nice properties of the BFS approach is that, in performing the BFS traversal, we can label each vertex by the length of a shortest path (in terms of the number of edges) from the start vertex s. In particular, if vertex v is placed into level i by a BFS starting at vertex s, then the length of a shortest path from s to v is i.

As with DFS, we can visualize the BFS traversal by orienting the edges along the direction in which they are explored during the traversal, and by distinguishing the edges used to discover new vertices, called ***discovery edges***, from those that lead to already visited vertices, called ***cross edges***. (See Figure 13.7(f).) As with the DFS, the discovery edges form a spanning tree, which in this case we call the BFS tree. We do not call the nontree edges "back edges" in this case, however, because none of them connects a vertex to one of its ancestors. Every nontree edge connects a vertex v to another vertex that is neither v's ancestor nor its descendent.

The BFS traversal algorithm has a number of interesting properties, some of which we explore in the proposition that follows.

Proposition 13.14: *Let G be an undirected graph on which a BFS traversal starting at vertex s has been performed. Then*

- *The traversal visits all vertices in the connected component of s*
- *The discovery-edges form a spanning tree T, which we call the BFS tree, of the connected component of s*
- *For each vertex v at level i, the path of the BFS tree T between s and v has i edges, and any other path of G between s and v has at least i edges*
- *If (u,v) is an edge that is not in the BFS tree, then the level numbers of u and v differ by at most 1*

We leave the justification of this proposition as an exercise (Exercise C-13.13). The analysis of the running time of BFS is similar to the one of DFS, which implies the following.

Proposition 13.15: *Let G be a graph with n vertices and m edges represented with the adjacency list structure. A BFS traversal of G takes $O(n+m)$ time. Also, there exist $O(n+m)$-time algorithms based on BFS for the following problems:*

- *Testing whether G is connected*
- *Computing a spanning tree of G, if G is connected*
- *Computing the connected components of G*
- *Given a start vertex s of G, computing, for every vertex v of G, a path with the minimum number of edges between s and v, or reporting that no such path exists*
- *Computing a cycle in G, or reporting that G has no cycles*

13.4   Directed Graphs

In this section, we consider issues that are specific to directed graphs. Recall that a directed graph (***digraph***), is a graph whose edges are all directed.

Methods Dealing with Directed Edges

In order to allow some or all the edges in a graph to be directed, we add the following functions to the graph ADT.

e.isDirected(): Test whether edge e is directed.

e.origin(): Return the origin vertex of edge e.

e.dest(): Return the destination vertex of edge e.

insertDirectedEdge(v, w, x): Insert and return a new directed edge with origin v and destination w and storing element x.

Also, if an edge e is directed, the function e.endVertices() should return a vertex list whose first element is the origin of e, and whose second element is the destination of e. The running time for the functions e.isDirected(), e.origin(), and e.dest() should be $O(1)$, and the running time of the function insertDirectedEdge(v, w, x) should match that of undirected edge insertion.

Reachability

One of the most fundamental issues with directed graphs is the notion of ***reachability***, which deals with determining which vertices can be reached by a path in a directed graph. A traversal in a directed graph always goes along directed paths, that is, paths where all the edges are traversed according to their respective directions. Given vertices u and v of a digraph G, we say that u ***reaches*** v (and v is ***reachable*** from u) if G has a directed path from u to v. We also say that a vertex v reaches an edge (w, z) if v reaches the origin vertex w of the edge.

A digraph G is ***strongly connected*** if for any two vertices u and v of G, u reaches v and v reaches u. A ***directed cycle*** of G is a cycle where all the edges are traversed according to their respective directions. (Note that G may have a cycle consisting of two edges with opposite direction between the same pair of vertices.) A digraph G is ***acyclic*** if it has no directed cycles. (See Figure 13.8 for some examples.)

The ***transitive closure*** of a digraph G is the digraph G^* such that the vertices of G^* are the same as the vertices of G, and G^* has an edge (u, v), whenever G has a directed path from u to v. That is, we define G^* by starting with the digraph G and adding in an extra edge (u, v) for each u and v such that v is reachable from u (and there isn't already an edge (u, v) in G).
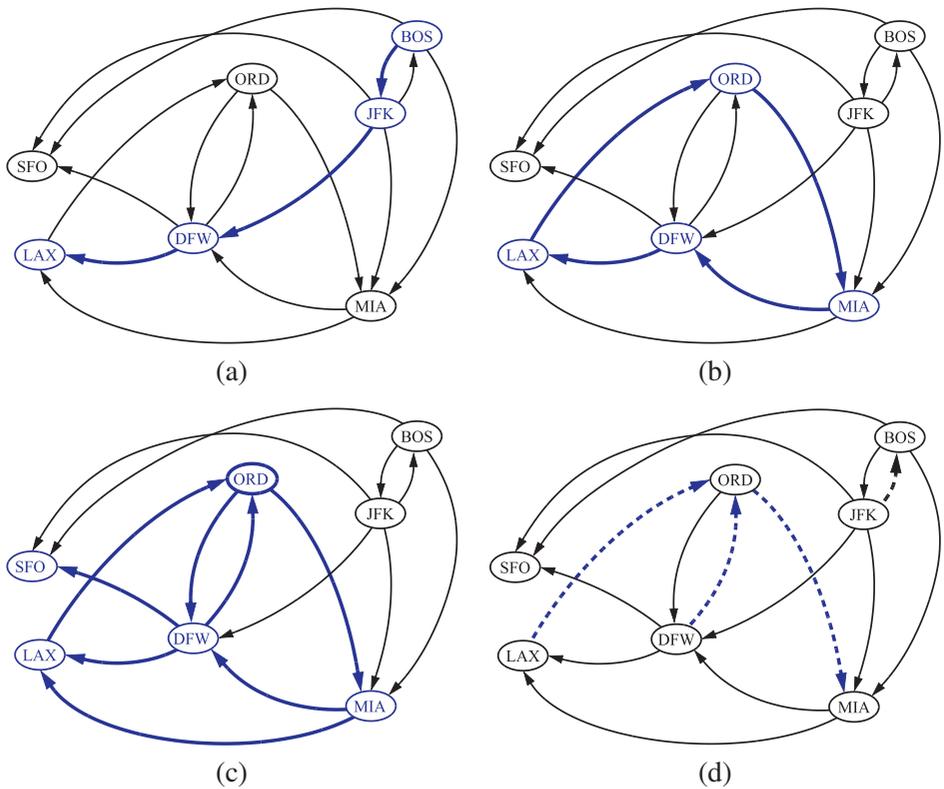
Figure 13.8: Examples of reachability in a digraph: (a) a directed path from BOS to LAX is drawn in blue; (b) a directed cycle (ORD, MIA, DFW, LAX, ORD) is shown in blue; its vertices induce a strongly connected subgraph; (c) the subgraph of the vertices and edges reachable from ORD is shown in blue; (d) removing the dashed blue edges gives an acyclic digraph.

Interesting problems that deal with reachability in a digraph G include the following:

- Given vertices u and v, determine whether u reaches v

- Find all the vertices of G that are reachable from a given vertex s

- Determine whether G is strongly connected

- Determine whether G is acyclic

- Compute the transitive closure G^* of G

In the remainder of this section, we explore some efficient algorithms for solving these problems.

13.4.1 Traversing a Digraph

As with undirected graphs, we can explore a digraph in a systematic way with methods akin to the depth-first search (DFS) and breadth-first search (BFS) algorithms defined previously for undirected graphs (Sections 13.3.1 and 13.3.5). Such explorations can be used, for example, to answer reachability questions. The directed depth-first search and breadth-first search methods we develop in this section for performing such explorations are very similar to their undirected counterparts. In fact, the only real difference is that the directed depth-first search and breadth-first search methods only traverse edges according to their respective directions.

The directed version of DFS starting at a vertex v can be described by the recursive algorithm in Code Fragment 13.21. (See Figure 13.9.)

Algorithm DirectedDFS(v):

    Mark vertex v as visited.
    for each outgoing edge (v, w) of v **do**
        if vertex w has not been visited **then**
            Recursively call DirectedDFS(w).

Code Fragment 13.21: The DirectedDFS algorithm.
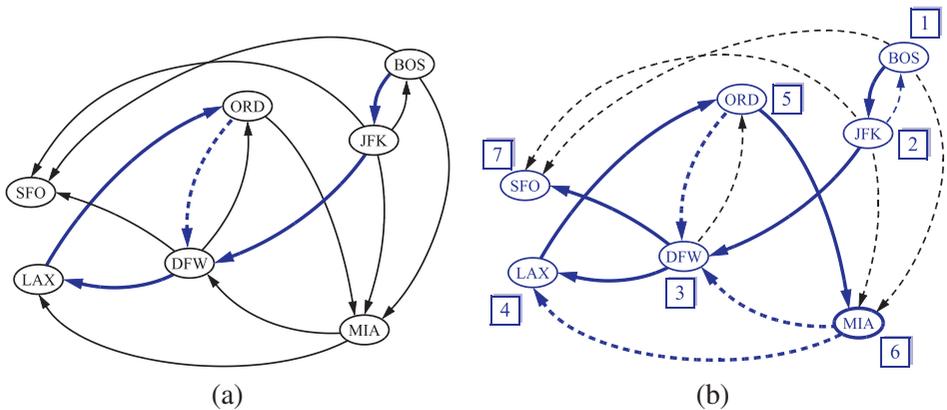
Figure 13.9: DFS in a digraph starting at vertex BOS: (a) intermediate step, where, for the first time, an already visited vertex (DFW) is reached; (b) the completed DFS. The tree edges are shown with solid blue lines, the back edges are shown with dashed blue lines, and the forward and cross edges are shown with dashed black lines. The order in which the vertices are visited is indicated by a label next to each vertex. The edge (ORD, DFW) is a back edge, but (DFW, ORD) is a forward edge. Edge (BOS, SFO) is a forward edge, and (SFO, LAX) is a cross edge.

A DFS on a digraph G partitions the edges of G reachable from the starting vertex into *tree edges* or *discovery edges*, which lead us to discover a new vertex, and *nontree edges*, which take us to a previously visited vertex. The tree edges form a tree rooted at the starting vertex, called the *depth-first search* tree. There are three kinds of nontree edges:

- *Back edges*, which connect a vertex to an ancestor in the DFS tree
- *Forward edges*, which connect a vertex to a descendent in the DFS tree
- *Cross edges*, which connect a vertex to a vertex that is neither its ancestor nor its descendent

Refer back to Figure 13.9(b) to see an example of each type of nontree edge.

Proposition 13.16: *Let G be a digraph. Depth-first search on G starting at a vertex s visits all the vertices of G that are reachable from s. Also, the DFS tree contains directed paths from s to every vertex reachable from s.*

Justification: Let V_s be the subset of vertices of G visited by DFS starting at vertex s. We want to show that V_s contains s and every vertex reachable from s belongs to V_s. Suppose now, for the sake of a contradiction, that there is a vertex w reachable from s that is not in V_s. Consider a directed path from s to w, and let (u, v) be the first edge on such a path taking us out of V_s, that is, u is in V_s but v is not in V_s. When DFS reaches u, it explores all the outgoing edges of u, and thus must also reach vertex v via edge (u, v). Hence, v should be in V_s, and we have obtained a contradiction. Therefore, V_s must contain every vertex reachable from s. ∎

Analyzing the running time of the directed DFS method is analogous to that for its undirected counterpart. In particular, a recursive call is made for each vertex exactly once, and each edge is traversed exactly once (from its origin). Hence, if n_s vertices and m_s edges are reachable from vertex s, a directed DFS starting at s runs in $O(n_s + m_s)$ time, provided the digraph is represented with a data structure that supports constant-time vertex and edge methods. The adjacency list structure satisfies this requirement, for example.

By Proposition 13.16, we can use DFS to find all the vertices reachable from a given vertex, and hence to find the transitive closure of G. That is, we can perform a DFS, starting from each vertex v of G, to see which vertices w are reachable from v, adding an edge (v, w) to the transitive closure for each such w. Likewise, by repeatedly traversing digraph G with a DFS, starting in turn at each vertex, we can easily test whether G is strongly connected. That is, G is strongly connected if each DFS visits all the vertices of G.

Thus, we may immediately derive the proposition that follows.

Proposition 13.17: *Let G be a digraph with n vertices and m edges. The following problems can be solved by an algorithm that traverses G n times using DFS, runs in $O(n(n+m))$ time, and uses $O(n)$ auxiliary space:*

- *Computing, for each vertex v of G, the subgraph reachable from v*
- *Testing whether G is strongly connected*
- *Computing the transitive closure G^* of G*

Testing for Strong Connectivity

Actually, we can determine if a directed graph G is strongly connected much faster than this, just by using two depth-first searches. We begin by performing a DFS of our directed graph G starting at an arbitrary vertex s. If there is any vertex of G that is not visited by this DFS and is not reachable from s, then the graph is not strongly connected. So, if this first DFS visits each vertex of G, then we reverse all the edges of G (using the reverseDirection function) and perform another DFS starting at s in this "reverse" graph. If every vertex of G is visited by this second DFS, then the graph is strongly connected because each of the vertices visited in this DFS can reach s. Since this algorithm makes just two DFS traversals of G, it runs in $O(n+m)$ time.

Directed Breadth-First Search

As with DFS, we can extend breadth-first search (BFS) to work for directed graphs. The algorithm still visits vertices level by level and partitions the set of edges into ***tree edges*** (or ***discovery edges***). Together these form a directed ***breadth-first search*** tree rooted at the start vertex and ***nontree edges***. Unlike the directed DFS method, however, the directed BFS method only leaves two kinds of nontree edges: ***back edges***, which connect a vertex to one of its ancestors, and ***cross edges***, which connect a vertex to another vertex that is neither its ancestor nor its descendent. There are no forward edges, which is a fact we explore in an exercise (Exercise C-13.9).

13.4.2 Transitive Closure

In this section, we explore an alternative technique for computing the transitive closure of a digraph. Let G be a digraph with n vertices and m edges. We compute the transitive closure of G in a series of rounds. We initialize $G_0 = G$. We also arbitrarily number the vertices of G as $v_1, v_2, \ldots, v_n$. We then begin the computation of the rounds, beginning with round 1. In a generic round k, we construct digraph G_k starting with $G_k = G_{k-1}$ and add to G_k the directed edge (v_i, v_j) if digraph G_{k-1} contains both the edges (v_i, v_k) and (v_k, v_j). In this way, we enforce a simple rule embodied in the proposition that follows.

Proposition 13.18: *For $i = 1, \ldots, n$, digraph G_k has an edge (v_i, v_j) if and only if digraph G has a directed path from v_i to v_j, whose intermediate vertices (if any) are in the set $\{v_1, \ldots, v_k\}$. In particular, G_n is equal to G^*, the transitive closure of G.*

Proposition 13.18 suggests a simple algorithm for computing the transitive closure of G that is based on the series of rounds we described above. This algorithm is known as the ***Floyd-Warshall algorithm***, and its pseudo-code is given in Code Fragment 13.22. From this pseudo-code, we can easily analyze the running time of the Floyd-Warshall algorithm assuming that the data structure representing G supports functions isAdjacentTo and insertDirectedEdge in $O(1)$ time. The main loop is executed n times and the inner loop considers each of $O(n^2)$ pairs of vertices, performing a constant-time computation for each one. Thus, the total running time of the Floyd-Warshall algorithm is $O(n^3)$.

Algorithm FloydWarshall(G):
    Input: A digraph G with n vertices
    Output: The transitive closure G^* of G
    let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of G
    $G_0 \leftarrow G$
    for $k \leftarrow 1$ to n **do**
        $G_k \leftarrow G_{k-1}$
        for all i, j in $\{1, \ldots, n\}$ with $i \neq j$ and $i, j \neq k$ **do**
            if both edges (v_i, v_k) and (v_k, v_j) are in G_{k-1} **then**
                add edge (v_i, v_j) to G_k (if it is not already present)
    return G_n

Code Fragment 13.22: Pseudo-code for the Floyd-Warshall algorithm. This algorithm computes the transitive closure G^* of G by incrementally computing a series of digraphs $G_0, G_1, \ldots, G_n$, where $k = 1, \ldots, n$.

This description is actually an example of an algorithmic design pattern known as dynamic programming, which is discussed in more detail in Section 12.2. From the description and analysis above, we may immediately derive the following proposition.

Proposition 13.19: *Let G be a digraph with n vertices, and let G be represented by a data structure that supports lookup and update of adjacency information in $O(1)$ time. Then the Floyd-Warshall algorithm computes the transitive closure G^* of G in $O(n^3)$ time.*

We illustrate an example run of the Floyd-Warshall algorithm in Figure 13.10.
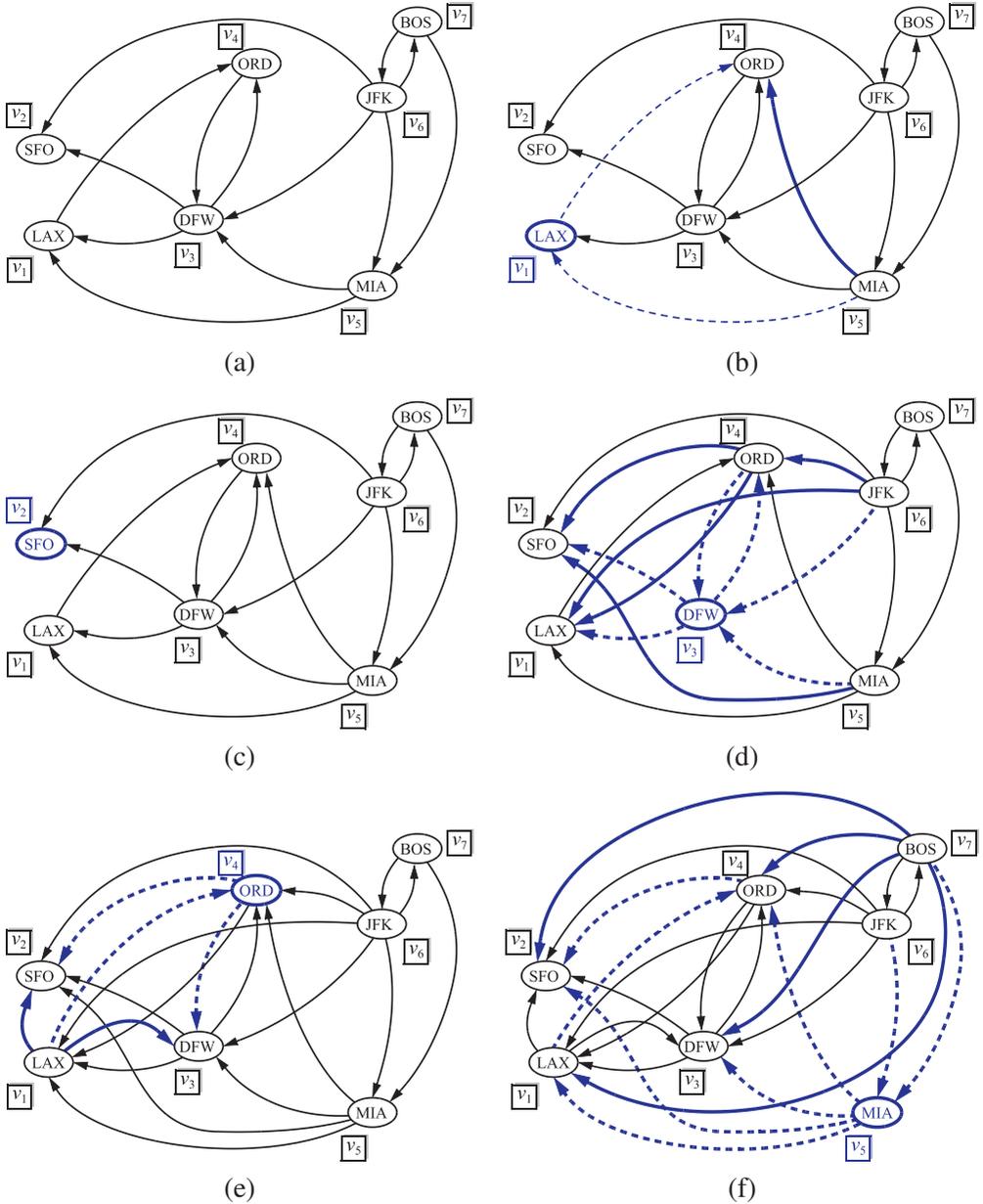
Figure 13.10: Sequence of digraphs computed by the Floyd-Warshall algorithm: (a) initial digraph $G = G_0$ and numbering of the vertices; (b) digraph G_1; (c) G_2; (d) G_3; (e) G_4; (f) G_5. (Note that $G_5 = G_6 = G_7$.) If digraph G_{k-1} has the edges (v_i, v_k) and (v_k, v_j), but not the edge (v_i, v_j). In the drawing of digraph G_k, we show edges (v_i, v_k) and (v_k, v_j) with dashed blue lines, and edge (v_i, v_j) with a thick blue line.

Performance of the Floyd-Warshall Algorithm

The running time of the Floyd-Warshall algorithm might appear to be slower than performing a DFS of a directed graph from each of its vertices, but this depends upon the representation of the graph. If a graph is represented using an adjacency matrix, then running the DFS method once on a directed graph G takes $O(n^2)$ time (we explore the reason for this in Exercise R-13.9). Thus, running DFS n times takes $O(n^3)$ time, which is no better than a single execution of the Floyd-Warshall algorithm, but the Floyd-Warshall algorithm would be much simpler to implement. Nevertheless, if the graph is represented using an adjacency list structure, then running the DFS algorithm n times would take $O(n(n+m))$ time to compute the transitive closure. Even so, if the graph is *dense*, that is, if it has $\Omega(n^2)$ edges, then this approach still runs in $O(n^3)$ time and is more complicated than a single instance of the Floyd-Warshall algorithm. The only case where repeatedly calling the DFS method is better is when the graph is not dense and is represented using an adjacency list structure.

13.4.3 Directed Acyclic Graphs

Directed graphs without directed cycles are encountered in many applications. Such a digraph is often referred to as a ***directed acyclic graph***, or ***DAG***, for short. Applications of such graphs include the following:

- Inheritance between classes of a C++ program
- Prerequisites between courses of a degree program
- Scheduling constraints between the tasks of a project

Example 13.20: *In order to manage a large project, it is convenient to break it up into a collection of smaller tasks. The tasks, however, are rarely independent, because scheduling constraints exist between them. (For example, in a house building project, the task of ordering nails obviously precedes the task of nailing shingles to the roof deck.) Clearly, scheduling constraints cannot have circularities, because they would make the project impossible. (For example, in order to get a job you need to have work experience, but in order to get work experience you need to have a job.) The scheduling constraints impose restrictions on the order in which the tasks can be executed. Namely, if a constraint says that task a must be completed before task b is started, then a must precede b in the order of execution of the tasks. Thus, if we model a feasible set of tasks as vertices of a directed graph, and we place a directed edge from v to w whenever the task for v must be executed before the task for w, then we define a directed acyclic graph.*

The example above motivates the following definition. Let G be a digraph with n vertices. A ***topological ordering*** of G is an ordering $v_1, \ldots, v_n$ of the vertices of G such that for every edge (v_i, v_j) of G, $i < j$. That is, a topological ordering is an ordering such that any directed path in G traverses vertices in increasing order. (See Figure 13.11.) Note that a digraph may have more than one topological ordering.
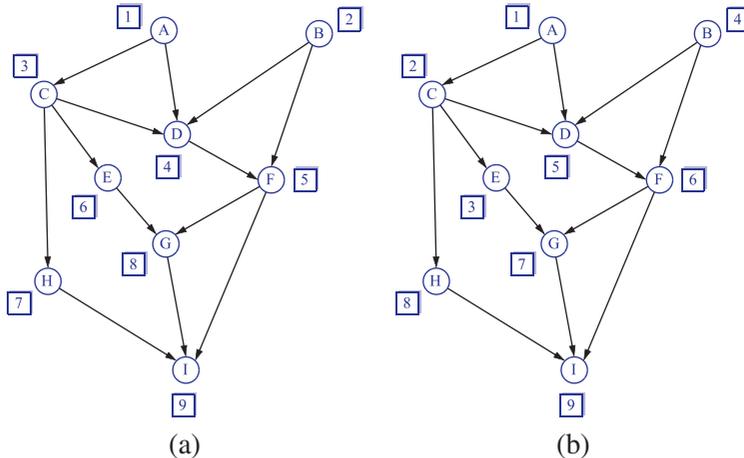
Figure 13.11: Two topological orderings of the same acyclic digraph.

Proposition 13.21: *G has a topological ordering if and only if it is acyclic.*

Justification: The necessity (the "only if" part of the statement) is easy to demonstrate. Suppose G is topologically ordered. Assume, for the sake of a contradiction, that G has a cycle consisting of edges $(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \ldots, (v_{i_{k-1}}, v_{i_0})$. Because of the topological ordering, we must have $i_0 < i_1 < \cdots < i_{k-1} < i_0$, which is clearly impossible. Thus, G must be acyclic.

We now argue the sufficiency of the condition (the "if" part). Suppose G is acyclic. We give an algorithmic description of how to build a topological ordering for G. Since G is acyclic, G must have a vertex with no incoming edges (that is, with in-degree 0). Let v_1 be such a vertex. Indeed, if v_1 did not exist, then in tracing a directed path from an arbitrary start vertex we would eventually encounter a previously visited vertex, thus contradicting the acyclicity of G. If we remove v_1 from G, together with its outgoing edges, the resulting digraph is still acyclic. Hence, the resulting digraph also has a vertex with no incoming edges, and we let v_2 be such a vertex. By repeating this process until the digraph becomes empty, we obtain an ordering $v_1, \ldots, v_n$ of the vertices of G. Because of the construction above, if (v_i, v_j) is an edge of G, then v_i must be deleted before v_j can be deleted, and thus $i < j$. Thus, $v_1, \ldots, v_n$ is a topological ordering. ■

Proposition 13.21's justification suggests an algorithm (Code Fragment 13.23), called ***topological sorting***, for computing a topological ordering of a digraph.

Algorithm TopologicalSort(*G*):

    Input: A digraph *G* with *n* vertices

    Output: A topological ordering $v_1, \ldots, v_n$ of *G*

    S ← an initially empty stack.

    for all *u* in *G*.vertices() **do**

      Let incounter(*u*) be the in-degree of *u*.

      if incounter(*u*) = 0 **then**

        S.push(*u*)

    i ← 1

    while !*S*.empty() **do**

      u ← *S*.pop()

      Let *u* be vertex number *i* in the topological ordering.

      i ← *i* + 1

      for all outgoing edges (*u*, *w*) of *u* **do**

        incounter(*w*) ← incounter(*w*) − 1

        if incounter(*w*) = 0 **then**

          S.push(*w*)

Code Fragment 13.23: Pseudo-code for the topological sorting algorithm. (We show an example application of this algorithm in Figure 13.12.)

Proposition 13.22: *Let G be a digraph with n vertices and m edges. The topological sorting algorithm runs in $O(n+m)$ time using $O(n)$ auxiliary space, and either computes a topological ordering of G or fails to number some vertices, which indicates that G has a directed cycle.*

Justification: The initial computation of in-degrees and setup of the incounter variables can be done with a simple traversal of the graph, which takes $O(n+m)$ time. We use the decorator pattern to associate counter attributes with the vertices. Say that a vertex *u* is ***visited*** by the topological sorting algorithm when *u* is removed from the stack *S*. A vertex *u* can be visited only when incounter(*u*) = 0, which implies that all its predecessors (vertices with outgoing edges into *u*) were previously visited. As a consequence, any vertex that is on a directed cycle will never be visited, and any other vertex will be visited exactly once. The algorithm traverses all the outgoing edges of each visited vertex once, so its running time is proportional to the number of outgoing edges of the visited vertices. Therefore, the algorithm runs in $O(n+m)$ time. Regarding the space usage, observe that the stack *S* and the incounter variables attached to the vertices use $O(n)$ space. ∎

As a side effect, the topological sorting algorithm of Code Fragment 13.23 also tests whether the input digraph *G* is acyclic. Indeed, if the algorithm terminates without ordering all the vertices, then the subgraph of the vertices that have not been ordered must contain a directed cycle.
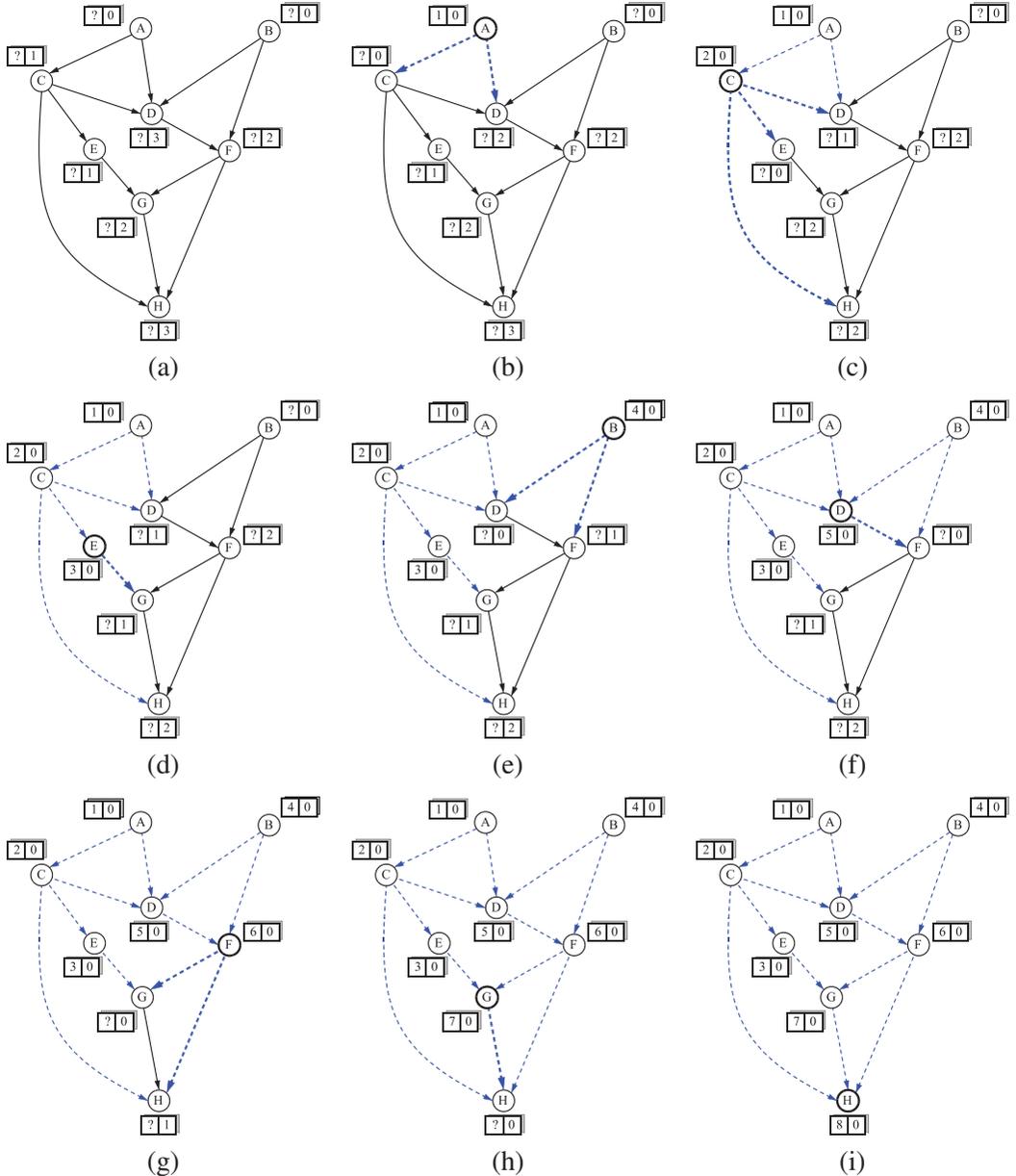
Figure 13.12: Example of a run of algorithm TopologicalSort (Code Fragment 13.23): (a) initial configuration; (b–i) after each while-loop iteration. The vertex labels show the vertex number and the current incounter value. The edges traversed are shown with dashed blue arrows. Thick lines denote the vertex and edges examined in the current iteration.

13.5 Shortest Paths

As we saw in Section 13.3.5, the breadth-first search strategy can be used to find a shortest path from some starting vertex to every other vertex in a connected graph. This approach makes sense in cases where each edge is as good as any other, but there are many situations where this approach is not appropriate. For example, we might be using a graph to represent a computer network (such as the Internet), and we might be interested in finding the fastest way to route a data packet between two computers. In this case, it is probably not appropriate for all the edges to be equal to each other, since some connections in a computer network are typically much faster than others (for example, some edges might represent slow phone-line connections while others might represent high-speed, fiber-optic connections). Likewise, we might want to use a graph to represent the roads between cities, and we might be interested in finding the fastest way to travel cross country. In this case, it is again probably not appropriate for all the edges to be equal to each other, because some inter-city distances will likely be much larger than others. Thus, it is natural to consider graphs whose edges are not weighted equally.

13.5.1 Weighted Graphs

A *weighted graph* is a graph that has a numeric (for example, integer) label $w(e)$ associated with each edge e, called the *weight* of edge e. We show an example of a weighted graph in Figure 13.13.
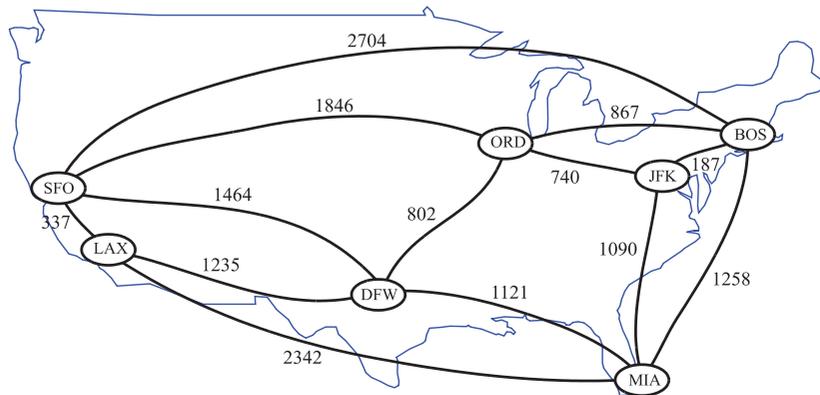
Figure 13.13: A weighted graph whose vertices represent major U.S. airports and whose edge weights represent distances in miles. This graph has a path from JFK to LAX of total weight 2,777 (going through ORD and DFW). This is the minimum weight path in the graph from JFK to LAX.

Defining Shortest Paths in a Weighted Graph

Let G be a weighted graph. The **length** (or weight) of a path is the sum of the weights of the edges of P. That is, if $P = ((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$, then the length of P, denoted $w(P)$, is defined as

$$w(P) = \sum_{i=0}^{k-1} w((v_i, v_{i+1})).$$

The **distance** from a vertex v to a vertex u in G, denoted $d(v, u)$, is the length of a minimum length path (also called **shortest path**) from v to u, if such a path exists.

People often use the convention that $d(v, u) = +\infty$ if there is no path at all from v to u in G. Even if there is a path from v to u in G, the distance from v to u may not be defined, however, if there is a cycle in G whose total weight is negative. For example, suppose vertices in G represent cities, and the weights of edges in G represent how much money it costs to go from one city to another. If someone were willing to actually pay us to go from say JFK to ORD, then the "cost" of the edge (JFK, ORD) would be negative. If someone else were willing to pay us to go from ORD to JFK, then there would be a negative-weight cycle in G and distances would no longer be defined. That is, anyone could now build a path (with cycles) in G from any city A to another city B that first goes to JFK and then cycles as many times as he or she likes from JFK to ORD and back, before going on to B. The existence of such paths would allow us to build arbitrarily low negative-cost paths (and, in this case, make a fortune in the process). But distances cannot be arbitrarily low negative numbers. Thus, any time we use edge weights to represent distances, we must be careful not to introduce any negative-weight cycles.

Suppose we are given a weighted graph G, and we are asked to find a shortest path from some vertex v to each other vertex in G, viewing the weights on the edges as distances. In this section, we explore efficient ways of finding all such shortest paths, if they exist. The first algorithm we discuss is for the simple, yet common, case when all the edge weights in G are nonnegative (that is, $w(e) \geq 0$ for each edge e of G); hence, we know in advance that there are no negative-weight cycles in G. Recall that the special case of computing a shortest path when all weights are equal to one was solved with the BFS traversal algorithm presented in Section 13.3.5.

There is an interesting approach for solving this **single-source** problem based on the **greedy method** design pattern (Section 12.4.2). Recall that in this pattern we solve the problem at hand by repeatedly selecting the best choice from among those available in each iteration. This paradigm can often be used in situations where we are trying to optimize some cost function over a collection of objects. We can add objects to our collection, one at a time, always picking the next one that optimizes the function from among those yet to be chosen.

13.5.2   Dijkstra's Algorithm

The main idea behind applying the greedy method pattern to the single-source, shortest-path problem is to perform a "weighted" breadth-first search starting at v. In particular, we can use the greedy method to develop an algorithm that iteratively grows a "cloud" of vertices out of v, with the vertices entering the cloud in order of their distances from v. Thus, in each iteration, the next vertex chosen is the vertex outside the cloud that is closest to v. The algorithm terminates when no more vertices are outside the cloud, at which point we have a shortest path from v to every other vertex of G. This approach is a simple, but nevertheless powerful, example of the greedy method design pattern.

A Greedy Method for Finding Shortest Paths

Applying the greedy method to the single-source, shortest-path problem, results in an algorithm known as ***Dijkstra's algorithm***. When applied to other graph problems, however, the greedy method may not necessarily find the best solution (such as in the so-called ***traveling salesman problem***, in which we wish to find the shortest path that visits all the vertices in a graph exactly once). Nevertheless, there are a number of situations in which the greedy method allows us to compute the best solution. In this chapter, we discuss two such situations: computing shortest paths and constructing a minimum spanning tree.

In order to simplify the description of Dijkstra's algorithm, we assume, in the following, that the input graph G is undirected (that is, all its edges are undirected) and simple (that is, it has no self-loops and no parallel edges). Hence, we denote the edges of G as unordered vertex pairs (u, z).

In Dijkstra's algorithm for finding shortest paths, the cost function we are trying to optimize in our application of the greedy method is also the function that we are trying to compute—the shortest path distance. This may at first seem like circular reasoning until we realize that we can actually implement this approach by using a "bootstrapping" trick, consisting of using an approximation to the distance function we are trying to compute, which in the end is equal to the true distance.

Edge Relaxation

Let us define a label $D[u]$ for each vertex u in V, which we use to approximate the distance in G from v to u. The meaning of these labels is that $D[u]$ always stores the length of the best path we have found ***so far*** from v to u. Initially, $D[v] = 0$ and $D[u] = +\infty$ for each $u \neq v$, and we define the set C (which is our "***cloud***" of vertices) to initially be the empty set $\emptyset$. At each iteration of the algorithm, we select a vertex u not in C with smallest $D[u]$ label, and we pull u into C. In the very first

iteration we will, of course, pull v into C. Once a new vertex u is pulled into C, we then update the label $D[z]$ of each vertex z that is adjacent to u and is outside of C, to reflect the fact that there may be a new and better way to get to z via u.

This update operation is known as a ***relaxation*** procedure, because it takes an old estimate and checks if it can be improved to get closer to its true value. (A metaphor for why we call this a relaxation comes from a spring that is stretched out and then "relaxed" back to its true resting shape.) In the case of Dijkstra's algorithm, the relaxation is performed for an edge (u, z) such that we have computed a new value of $D[u]$ and wish to see if there is a better value for $D[z]$ using the edge (u, z). The specific edge relaxation operation is as follows:

> **Edge Relaxation**:
>
> $$\textbf{if } D[u] + w((u, z)) < D[z] \textbf{ then}$$
> $$D[z] \leftarrow D[u] + w((u, z))$$

We give the pseudo-code for Dijkstra's algorithm in Code Fragment 13.24. Note that we use a priority queue Q to store the vertices outside of the cloud C.

Algorithm ShortestPath(G, v):

    Input: A simple undirected weighted graph G with nonnegative edge weights and a distinguished vertex v of G

    Output: A label $D[u]$, for each vertex u of G, such that $D[u]$ is the length of a shortest path from v to u in G

    Initialize $D[v] \leftarrow 0$ and $D[u] \leftarrow +\infty$ for each vertex $u \neq v$.

    Let a priority queue Q contain all the vertices of G using the D labels as keys.

    while Q is not empty **do**

        {pull a new vertex u into the cloud}

        $u \leftarrow Q.$removeMin()

        for each vertex z adjacent to u such that z is in Q **do**

            {perform the ***relaxation*** procedure on edge (u, z)}

            if $D[u] + w((u, z)) < D[z]$ **then**

                $D[z] \leftarrow D[u] + w((u, z))$

                Change to $D[z]$ the key of vertex z in Q.

    return the label $D[u]$ of each vertex u

Code Fragment 13.24: Dijkstra's algorithm for the single-source, shortest-path problem.

We illustrate several iterations of Dijkstra's algorithm in Figures 13.14 and 13.15.
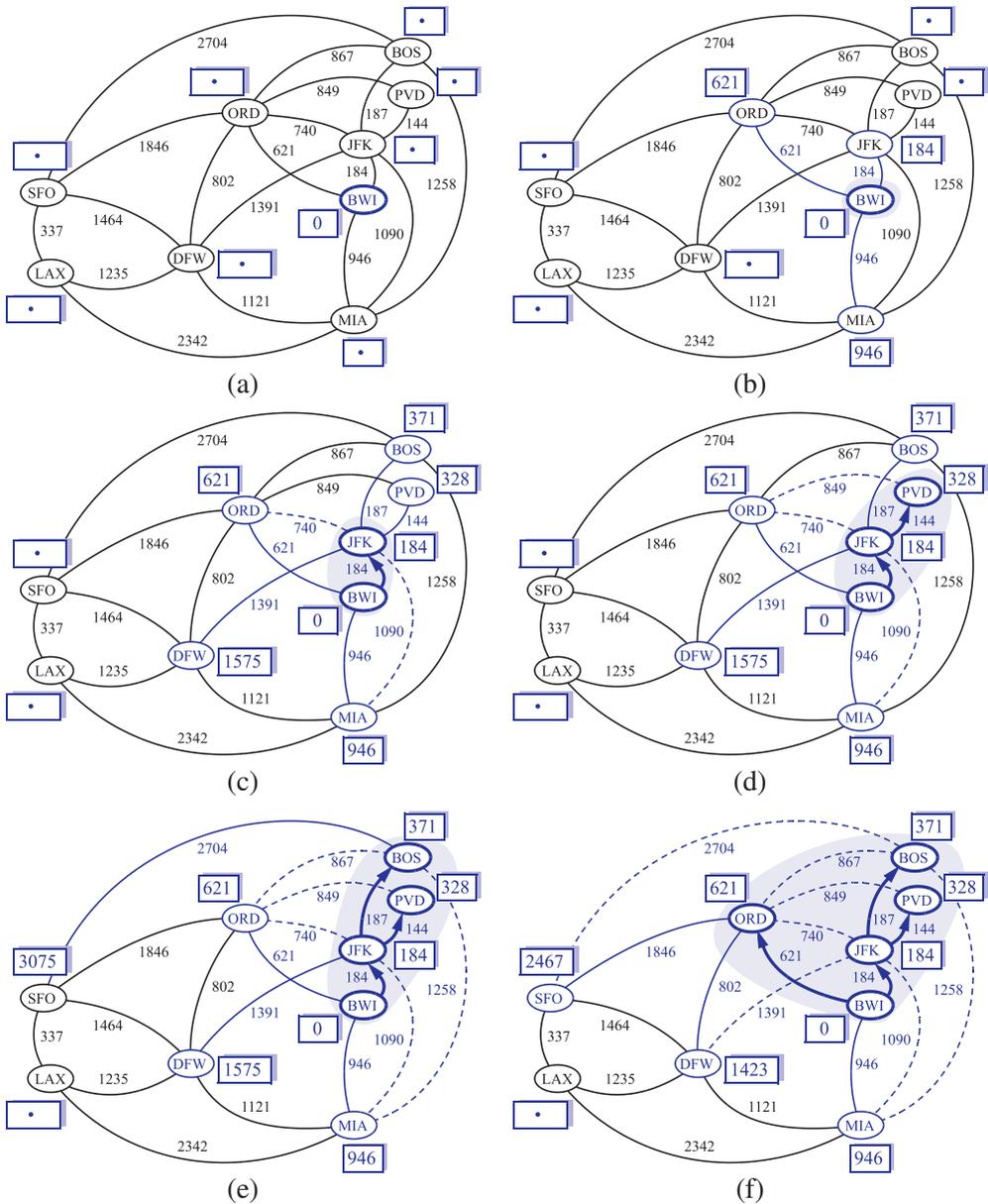
Figure 13.14: An execution of Dijkstra's algorithm on a weighted graph. The start vertex is BWI. A box next to each vertex v stores the label $D[v]$. The symbol $\bullet$ is used instead of $+\infty$. The edges of the shortest-path tree are drawn as thick blue arrows and, for each vertex u outside the "cloud," we show the current best edge for pulling in u with a solid blue line. (Continues in Figure 13.15.)

(g)

(h)

(i)

(j)
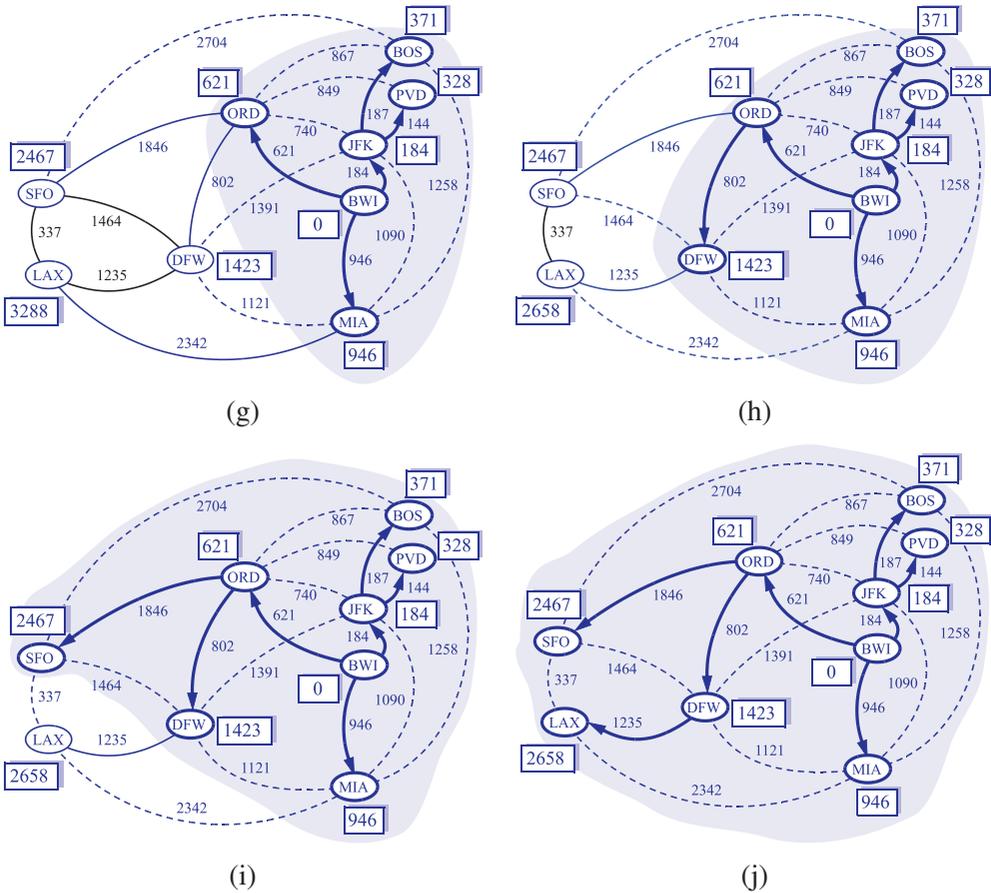
Figure 13.15: An example execution of Dijkstra's algorithm. (Continued from Figure 13.14.)

Why It Works

The interesting, and possibly even a little surprising, aspect of the Dijkstra algorithm is that, at the moment a vertex u is pulled into C, its label $D[u]$ stores the correct length of a shortest path from v to u. Thus, when the algorithm terminates, it will have computed the shortest-path distance from v to every vertex of G. That is, it will have solved the single-source, shortest-path problem.

It is probably not immediately clear why Dijkstra's algorithm correctly finds the shortest path from the start vertex v to each other vertex u in the graph. Why is it that the distance from v to u is equal to the value of the label $D[u]$ at the time vertex u is pulled into the cloud C (which is also the time u is removed from the priority queue Q)? The answer to this question depends on there being no negative-weight edges in the graph, since that allows the greedy method to work correctly, as we show in the proposition that follows.

Proposition 13.23: *In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v,u)$, the length of a shortest path from v to u.*

Justification: Suppose that $D[t] > d(v,t)$ for some vertex t in V, and let u be the **first** vertex the algorithm pulled into the cloud C (that is, removed from Q) such that $D[u] > d(v,u)$. There is a shortest path P from v to u (for otherwise $d(v,u) = +\infty = D[u]$). Let us therefore consider the moment when u is pulled into C, and let z be the first vertex of P (when going from v to u) that is not in C at this moment. Let y be the predecessor of z in path P (note that we could have $y = v$). (See Figure 13.16.) We know, by our choice of z, that y is already in C at this point. Moreover, $D[y] = d(v,y)$, since u is the **first** incorrect vertex. When y was pulled into C, we tested (and possibly updated) $D[z]$ so that we had at that point

$$D[z] \le D[y] + w((y,z)) = d(v,y) + w((y,z)).$$

But since z is the next vertex on the shortest path from v to u, this implies that

$$D[z] = d(v,z).$$

But we are now at the moment when we pick u, not z, to join C; hence

$$D[u] \le D[z].$$

It should be clear that a subpath of a shortest path is itself a shortest path. Hence, since z is on the shortest path from v to u

$$d(v,z) + d(z,u) = d(v,u).$$

Moreover, $d(z,u) \ge 0$ because there are no negative-weight edges. Therefore

$$D[u] \le D[z] = d(v,z) \le d(v,z) + d(z,u) = d(v,u).$$

But this contradicts the definition of u; hence, there can be no such vertex u. ■
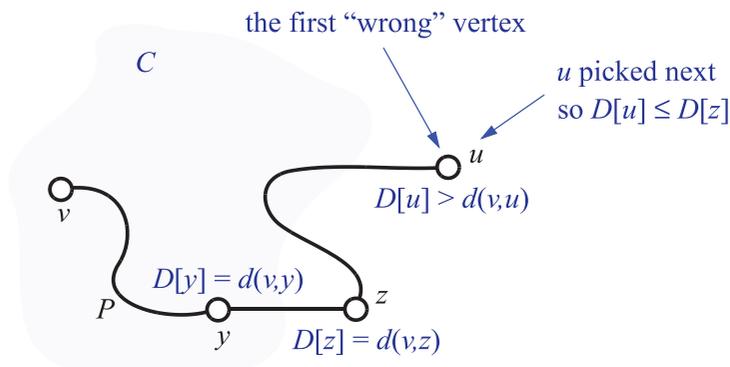
Figure 13.16: A schematic for the justification of Proposition 13.23.

The Running Time of Dijkstra's Algorithm

In this section, we analyze the time complexity of Dijkstra's algorithm. We denote the number of vertices and edges of the input graph G with n and m, respectively. We assume that the edge weights can be added and compared in constant time. Because of the high level of the description we gave for Dijkstra's algorithm in Code Fragment 13.24, analyzing its running time requires that we give more details on its implementation. Specifically, we should indicate the data structures used and how they are implemented.

Let us first assume that we are representing the graph G using an adjacency list structure. This data structure allows us to step through the vertices adjacent to u during the relaxation step in time proportional to their number. It still does not settle all the details for the algorithm, however, as we must say more about how to implement the other principle data structure in the algorithm—the priority queue Q.

An efficient implementation of the priority queue Q uses a heap (Section 8.3). This allows us to extract the vertex u with smallest D label (call to the removeMin function) in $O(\log n)$ time. As noted in the pseudo-code, each time we update a $D[z]$ label we need to update the key of z in the priority queue. Thus, we actually need a heap implementation of an adaptable priority queue (Section 8.4). If Q is an adaptable priority queue implemented as a heap, then this key update can, for example, be done using the replace(e,k), where e is the entry storing the key for the vertex z. If e is location aware, then we can easily implement such key updates in $O(\log n)$ time, since a location-aware entry for vertex z would allow Q to have immediate access to the entry e storing z in the heap (see Section 8.4.2). Assuming this implementation of Q, Dijkstra's algorithm runs in $O((n+m)\log n)$ time.

Referring back to Code Fragment 13.24, the details of the running-time analysis are as follows:

- Inserting all the vertices in Q with their initial key value can be done in $O(n\log n)$ time by repeated insertions, or in $O(n)$ time using bottom-up heap construction (see Section 8.3.6).
- At each iteration of the **while** loop, we spend $O(\log n)$ time to remove vertex u from Q, and $O(\text{degree}(v)\log n)$ time to perform the relaxation procedure on the edges incident on u.
- The overall running time of the **while** loop is

$$\sum_{v \text{ in } G} (1+\text{degree}(v))\log n,$$

which is $O((n+m)\log n)$ by Proposition 13.6.

Note that if we wish to express the running time as a function of n only, then it is $O(n^2 \log n)$ in the worst case.

13.6 Minimum Spanning Trees

Suppose we wish to connect all the computers in a new office building using the least amount of cable. We can model this problem using a weighted graph G whose vertices represent the computers, and whose edges represent all the possible pairs (u, v) of computers, where the weight $w((v, u))$ of edge (v, u) is equal to the amount of cable needed to connect computer v to computer u. Rather than computing a shortest-path tree from some particular vertex v, we are interested instead in finding a (free) tree T that contains all the vertices of G and has the minimum total weight over all such trees. Methods for finding such a tree are the focus of this section.

Problem Definition

Given a weighted undirected graph G, we are interested in finding a tree T that contains all the vertices in G and minimizes the sum

$$w(T) = \sum_{(v, u) \text{ in } T} w((v, u)).$$

A tree, such as this, that contains every vertex of a connected graph G is said to be a ***spanning tree***, and the problem of computing a spanning tree T with smallest total weight is known as the ***minimum spanning tree*** (or ***MST***) problem.

The development of efficient algorithms for the minimum spanning tree problem predates the modern notion of computer science itself. In this section, we discuss two classic algorithms for solving the MST problem. These algorithms are both applications of the ***greedy method***, which, as was discussed briefly in the previous section, is based on choosing objects to join a growing collection by iteratively picking an object that minimizes some cost function. The first algorithm we discuss is Kruskal's algorithm, which "grows" the MST in clusters by considering edges in order of their weights. The second algorithm we discuss is the Prim-Jarník algorithm, which grows the MST from a single root vertex, much in the same way as Dijkstra's shortest-path algorithm.

As in Section 13.5.2, in order to simplify the description of the algorithms, we assume, in the following, that the input graph G is undirected (that is, all its edges are undirected) and simple (that is, it has no self-loops and no parallel edges). Hence, we denote the edges of G as unordered vertex pairs (u, z).

Before we discuss the details of these algorithms, however, let us give a crucial fact about minimum spanning trees that forms the basis of the algorithms.

A Crucial Fact About Minimum Spanning Trees

The two MST algorithms we discuss are based on the greedy method, which in this case depends crucially on the following fact. (See Figure 13.17.)

e Belongs to a Minimum Spanning Tree
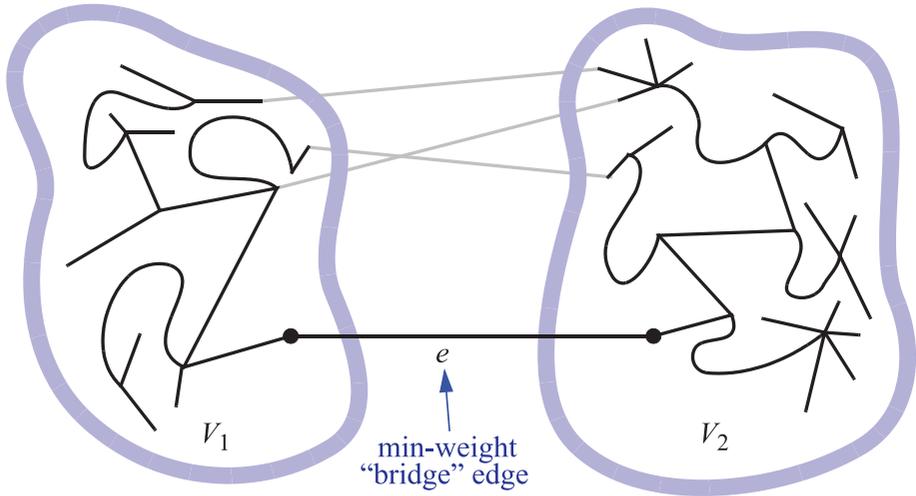
Figure 13.17: The crucial fact about minimum spanning trees.

Proposition 13.24: *Let G be a weighted connected graph, and let V_1 and V_2 be a partition of the vertices of G into two disjoint nonempty sets. Furthermore, let e be an edge in G with minimum weight from among those with one endpoint in V_1 and the other in V_2. There is a minimum spanning tree T that has e as one of its edges.*

Justification: Let T be a minimum spanning tree of G. If T does not contain edge e, the addition of e to T must create a cycle. Therefore, there is some edge f of this cycle that has one endpoint in V_1 and the other in V_2. Moreover, by the choice of e, $w(e) \leq w(f)$. If we remove f from $T \cup \{e\}$, we obtain a spanning tree whose total weight is no more than before. Since T is a minimum spanning tree, this new tree must also be a minimum spanning tree.   ■

In fact, if the weights in G are distinct, then the minimum spanning tree is unique. We leave the justification of this less crucial fact as an exercise (Exercise C-13.17). In addition, note that Proposition 13.24 remains valid even if the graph G contains negative-weight edges or negative-weight cycles, unlike the algorithms we presented for shortest paths.

13.6.1 Kruskal's Algorithm

The reason Proposition 13.24 is so important is that it can be used as the basis for building a minimum spanning tree. In Kruskal's algorithm, it is used to build the minimum spanning tree in clusters. Initially, each vertex is in its own cluster all by itself. The algorithm then considers each edge in turn, ordered by increasing weight. If an edge e connects two different clusters, then e is added to the set of edges of the minimum spanning tree, and the two clusters connected by e are merged into a single cluster. If, on the other hand, e connects two vertices that are already in the same cluster, then e is discarded. Once the algorithm has added enough edges to form a spanning tree, it terminates and outputs this tree as the minimum spanning tree.

We give pseudo-code for Kruskal's MST algorithm in Code Fragment 13.25 and we show the working of this algorithm in Figures 13.18, 13.19, and 13.20.

Algorithm Kruskal(G):
    Input: A simple connected weighted graph G with n vertices and m edges
    Output: A minimum spanning tree T for G
  for each vertex v in G **do**
    Define an elementary cluster $C(v) \leftarrow \{v\}$.
  Initialize a priority queue Q to contain all edges in G, using the weights as keys.
  $T \leftarrow \emptyset$ \{T will ultimately contain the edges of the MST\}
  while T has fewer than $n - 1$ edges **do**
    $(u, v) \leftarrow Q$.removeMin()
    Let $C(v)$ be the cluster containing v, and let $C(u)$ be the cluster containing u.
    if $C(v) \neq C(u)$ **then**
      Add edge (v, u) to T.
      Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$.
  return tree T

Code Fragment 13.25: Kruskal's algorithm for the MST problem.

As mentioned before, the correctness of Kruskal's algorithm follows from the crucial fact about minimum spanning trees, Proposition 13.24. Each time Kruskal's algorithm adds an edge (v, u) to the minimum spanning tree T, we can define a partitioning of the set of vertices V (as in the proposition) by letting V_1 be the cluster containing v and letting V_2 contain the rest of the vertices in V. This clearly defines a disjoint partitioning of the vertices of V and, more importantly, since we are extracting edges from Q in order by their weights, e must be a minimum-weight edge with one vertex in V_1 and the other in V_2. Thus, Kruskal's algorithm always adds a valid minimum spanning tree edge.
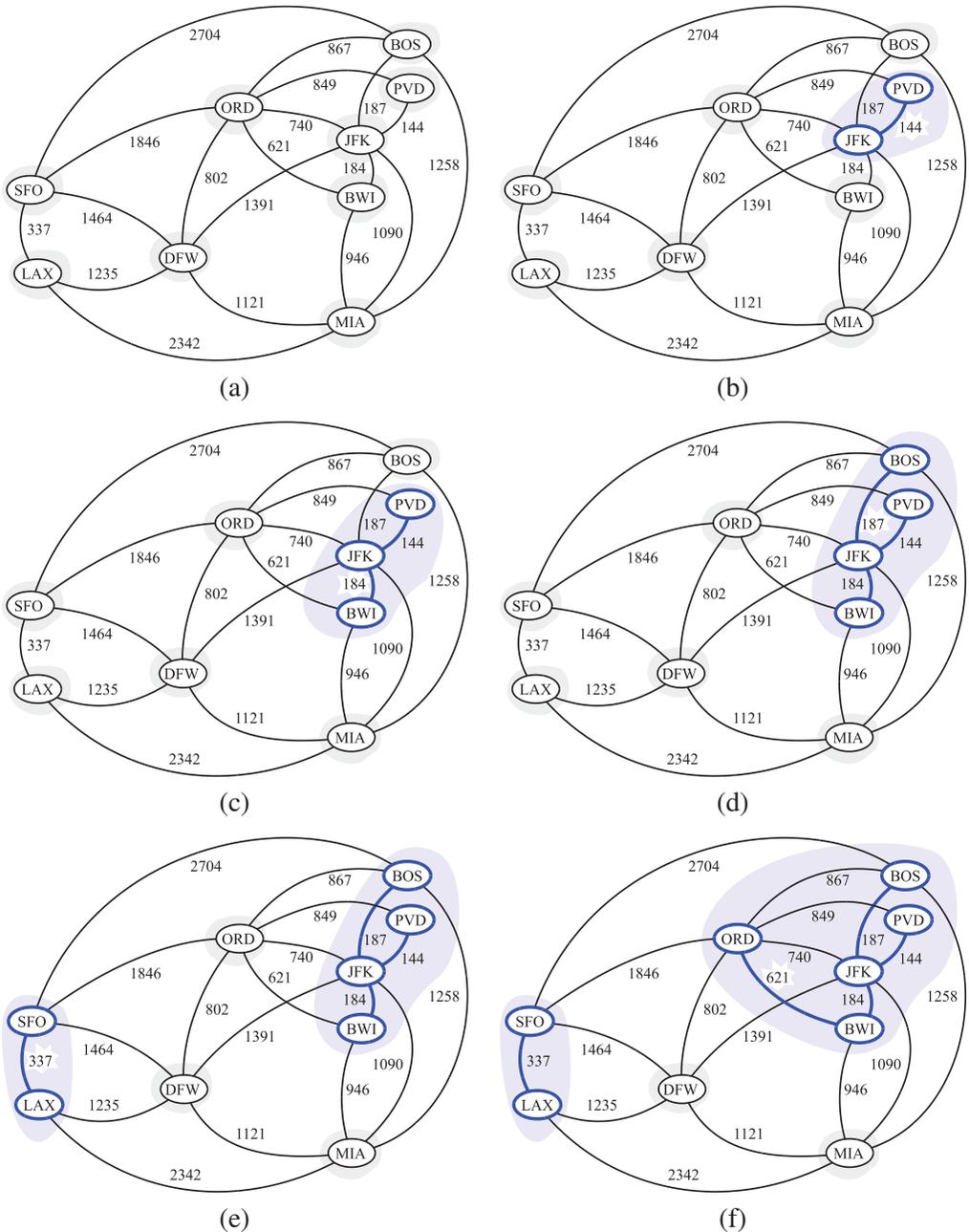
Figure 13.18: Example of an execution of Kruskal's MST algorithm on a graph with integer weights. We show the clusters as shaded regions and we highlight the edge being considered in each iteration. (Continues in Figure 13.19.)
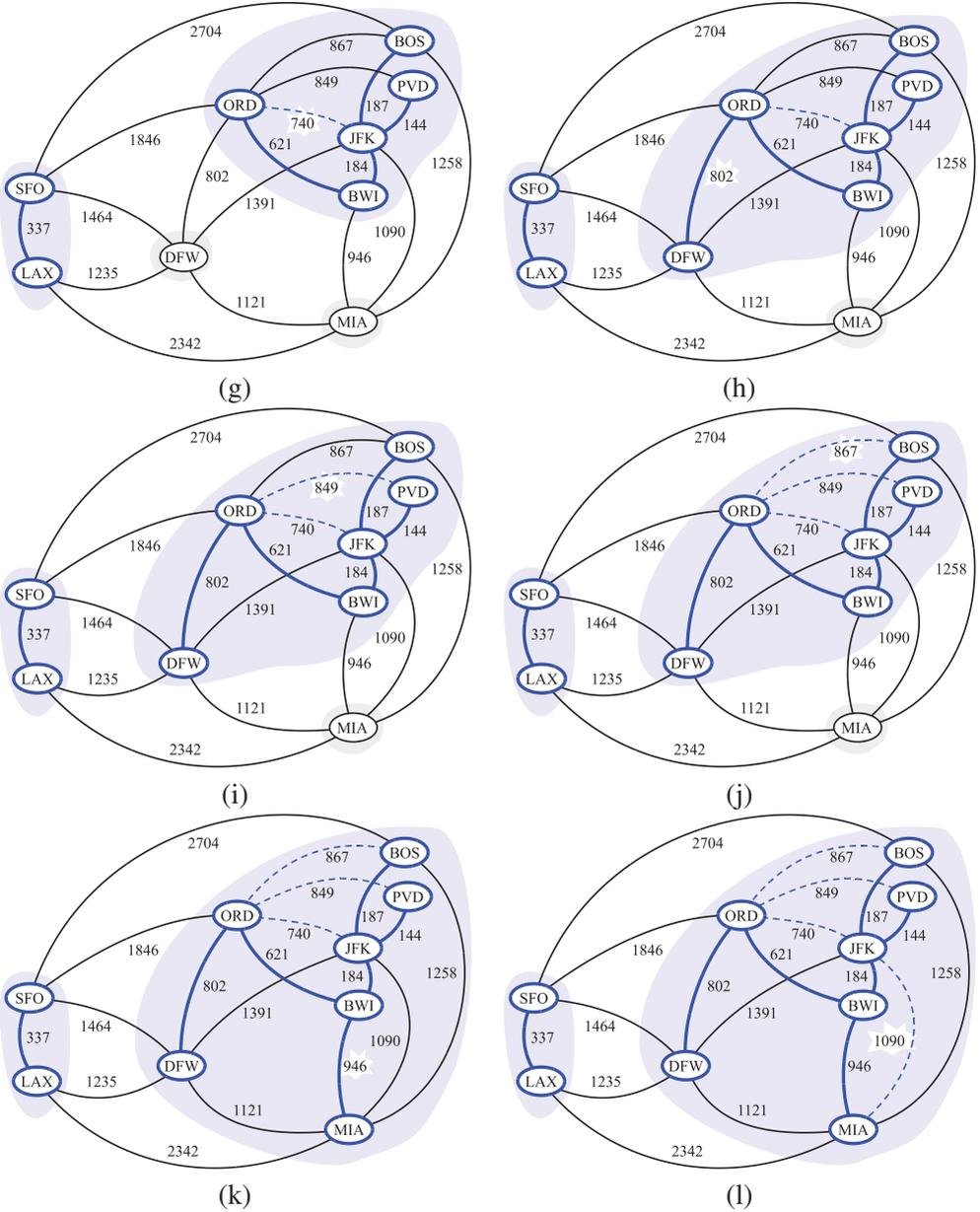
Figure 13.19: Example of an execution of Kruskal's MST algorithm. Rejected edges are shown dashed. (Continues in Figure 13.20.)

(m)   (n)
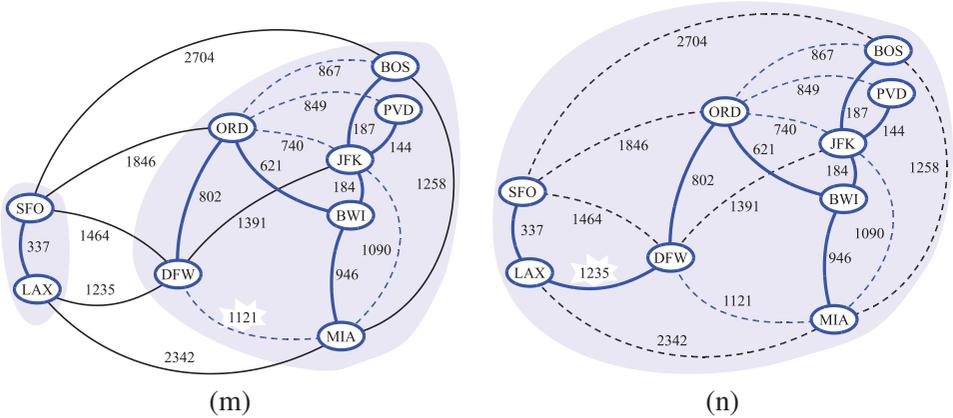
Figure 13.20: Example of an execution of Kruskal's MST algorithm. The edge considered in (n) merges the last two clusters, which concludes this execution of Kruskal's algorithm. (Continued from Figure 13.19.)

The Running Time of Kruskal's Algorithm

We denote the number of vertices and edges of the input graph G with n and m, respectively. Because of the high level of the description we gave for Kruskal's algorithm in Code Fragment 13.25, analyzing its running time requires that we give more details on its implementation. Specifically, we should indicate the data structures used and how they are implemented.

We can implement the priority queue Q using a heap. Thus, we can initialize Q in $O(m\log m)$ time by repeated insertions, or in $O(m)$ time using bottom-up heap construction (see Section 8.3.6). In addition, at each iteration of the **while** loop, we can remove a minimum-weight edge in $O(\log m)$ time, which actually is $O(\log n)$, since G is simple. Thus, the total time spent performing priority queue operations is no more than $O(m\log n)$.

We can represent each cluster C using one of the union-find partition data structures discussed in Section 11.4.3. Recall that the sequence-based union-find structure allows us to perform a series of N union and find operations in $O(N\log N)$ time, and the tree-based version can implement such a series of operations in $O(N\log^* N)$ time. Thus, since we perform $n-1$ calls to function union and at most m calls to find, the total time spent on merging clusters and determining the clusters that vertices belong to is no more than $O(m\log n)$ using the sequence-based approach or $O(m\log^* n)$ using the tree-based approach.

Therefore, using arguments similar to those used for Dijkstra's algorithm, we conclude that the running time of Kruskal's algorithm is $O((n+m)\log n)$, which can be simplified as $O(m\log n)$, since G is simple and connected.

13.6.2 The Prim-Jarník Algorithm

In the Prim-Jarník algorithm, we grow a minimum spanning tree from a single cluster starting from some "root" vertex v. The main idea is similar to that of Dijkstra's algorithm. We begin with some vertex v, defining the initial "cloud" of vertices C. Then, in each iteration, we choose a minimum-weight edge $e = (v, u)$, connecting a vertex v in the cloud C to a vertex u outside of C. The vertex u is then brought into the cloud C and the process is repeated until a spanning tree is formed. Again, the crucial fact about minimum spanning trees comes to play, because by always choosing the smallest-weight edge joining a vertex inside C to one outside C, we are assured of always adding a valid edge to the MST.

To efficiently implement this approach, we can take another cue from Dijkstra's algorithm. We maintain a label $D[u]$ for each vertex u outside the cloud C, so that $D[u]$ stores the weight of the best current edge for joining u to the cloud C. These labels allow us to reduce the number of edges that we must consider in deciding which vertex is next to join the cloud. We give the pseudo-code in Code Fragment 13.26.

Algorithm PrimJarnik(G):

    Input: A weighted connected graph G with n vertices and m edges

    Output: A minimum spanning tree T for G

  Pick any vertex v of G

  $D[v] \leftarrow 0$

  for each vertex $u \neq v$ **do**

    $D[u] \leftarrow +\infty$

  Initialize $T \leftarrow \emptyset$.

  Initialize a priority queue Q with an entry $((u, null), D[u])$ for each vertex u, where $(u, null)$ is the element and $D[u])$ is the key.

  while Q is not empty **do**

    $(u, e) \leftarrow Q.\mathsf{removeMin}()$

    Add vertex u and edge e to T.

    for each vertex z adjacent to u such that z is in Q **do**

      {perform the relaxation procedure on edge (u, z)}

      if $w((u, z)) < D[z]$ **then**

        $D[z] \leftarrow w((u, z))$

        Change to $(z, (u, z))$ the element of vertex z in Q.

        Change to $D[z]$ the key of vertex z in Q.

  return the tree T

Code Fragment 13.26: The Prim-Jarník algorithm for the MST problem.

Analyzing the Prim-Jarník Algorithm

Let n and m denote the number of vertices and edges of the input graph G, respectively. The implementation issues for the Prim-Jarník algorithm are similar to those for Dijkstra's algorithm. If we implement the adaptable priority queue Q as a heap that supports location-aware entries (Section 8.4.2), then we can extract the vertex u in each iteration in $O(\log n)$ time. In addition, we can update each $D[z]$ value in $O(\log n)$ time, as well, which is a computation considered at most once for each edge (u, z). The other steps in each iteration can be implemented in constant time. Thus, the total running time is $O((n + m) \log n)$, which is $O(m \log n)$.

Illustrating the Prim-Jarník Algorithm

We illustrate the Prim-Jarník algorithm in Figures 13.21 through 13.22.
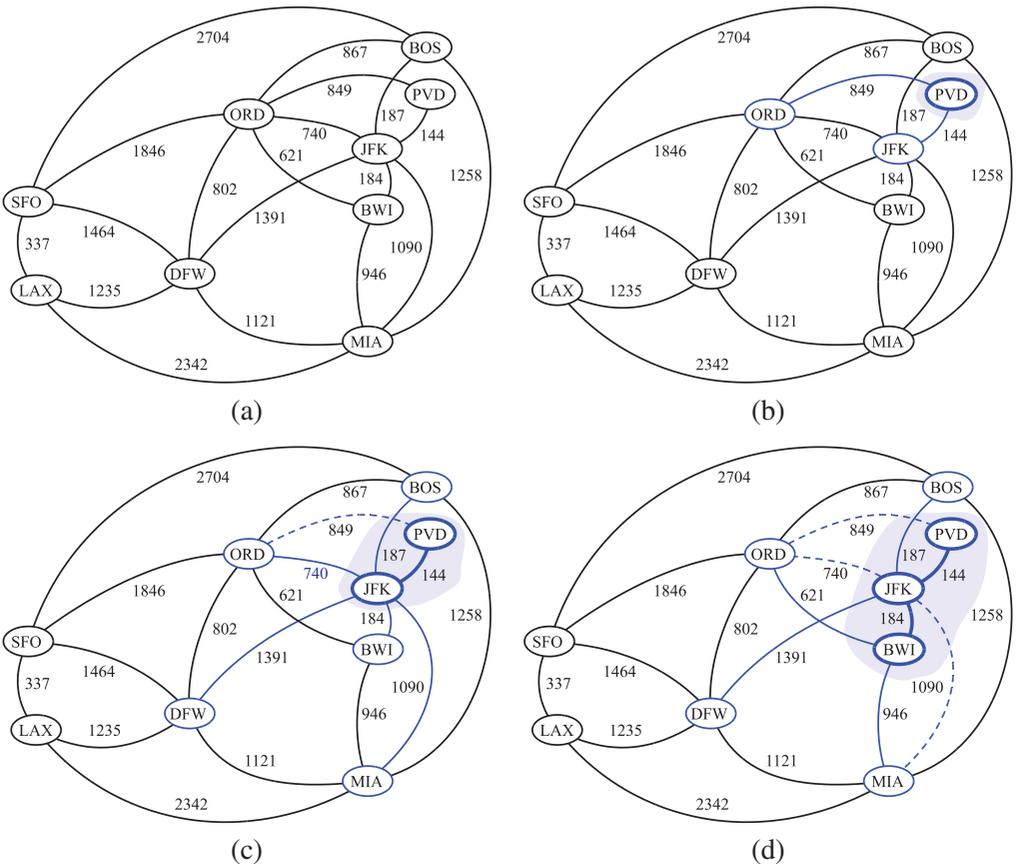
Figure 13.21: The Prim-Jarník MST algorithm. (Continues in Figure 13.22.)
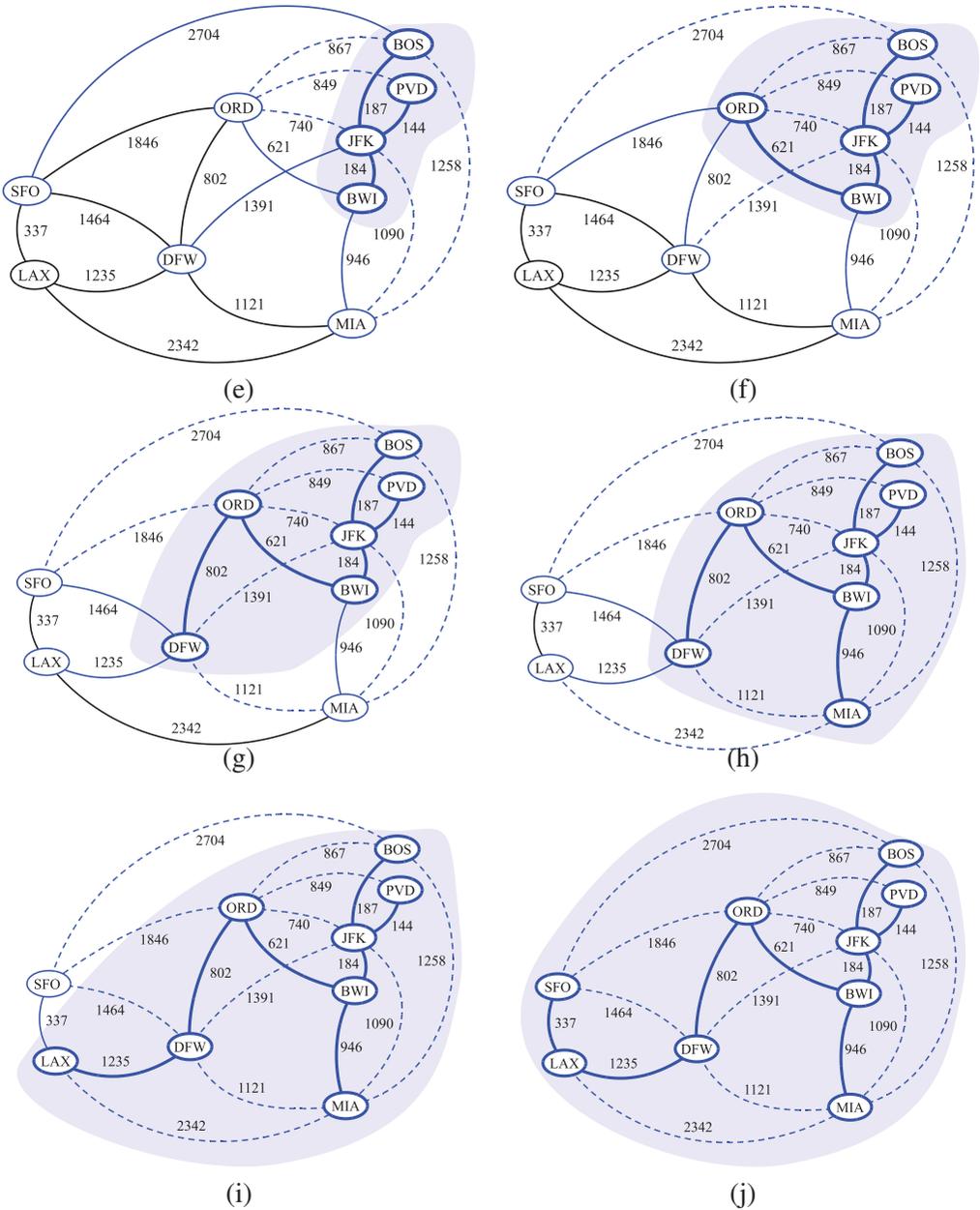
Figure 13.22: The Prim-Jarník MST algorithm. (Continued from Figure 13.21.)

13.7 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-13.1 Draw a simple undirected graph G that has 12 vertices, 18 edges, and 3 connected components. Why would it be impossible to draw G with 3 connected components if G had 66 edges?

R-13.2 Draw an adjacency list and adjacency matrix representation of the undirected graph shown in Figure 13.1.

R-13.3 Draw a simple connected directed graph with 8 vertices and 16 edges such that the in-degree and out-degree of each vertex is 2. Show that there is a single (nonsimple) cycle that includes all the edges of your graph, that is, you can trace all the edges in their respective directions without ever lifting your pencil. (Such a cycle is called an ***Euler tour***.)

R-13.4 Repeat the previous problem and then remove one edge from the graph. Show that now there is a single (nonsimple) path that includes all the edges of your graph. (Such a path is called an ***Euler path***.)

R-13.5 Bob loves foreign languages and wants to plan his course schedule for the following years. He is interested in the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169. The course prerequisites are:

- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32

Find the sequence of courses that allows Bob to satisfy all the prerequisites.

R-13.6 Suppose we represent a graph G having n vertices and m edges with the edge list structure. Why, in this case, does the insertVertex function run in $O(1)$ time while the eraseVertex function runs in $O(m)$ time?

R-13.7 Let G be a graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given by the table below:

Vertex	Adjacent Vertices
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

Assume that, in a traversal of G, the adjacent vertices of a given vertex are returned in the same order as they are listed in the table above.

   a. Draw G.
   b. Give the sequence of vertices of G visited using a DFS traversal starting at vertex 1.
   c. Give the sequence of vertices visited using a BFS traversal starting at vertex 1.

R-13.8 Would you use the adjacency list structure or the adjacency matrix structure in each of the following cases? Justify your choice.

   a. The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible.
   b. The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.
   c. You need to answer the query isAdjacentTo as fast as possible, no matter how much space you use.

R-13.9 Explain why the DFS traversal runs in $O(n^2)$ time on an n-vertex simple graph that is represented with the adjacency matrix structure.

R-13.10 Draw the transitive closure of the directed graph shown in Figure 13.2.

R-13.11 Compute a topological ordering for the directed graph drawn with solid edges in Figure 13.8(d).

R-13.12 Can we use a queue instead of a stack as an auxiliary data structure in the topological sorting algorithm shown in Code Fragment 13.23? Why or why not?

R-13.13 Draw a simple, connected, weighted graph with 8 vertices and 16 edges, each with unique edge weights. Identify one vertex as a "start" vertex and illustrate a running of Dijkstra's algorithm on this graph.

R-13.14 Show how to modify the pseudo-code for Dijkstra's algorithm for the case when the graph may contain parallel edges and self-loops.

R-13.15 Show how to modify the pseudo-code for Dijkstra's algorithm for the case
when the graph is directed and we want to compute shortest directed paths
from the source vertex to all the other vertices.

R-13.16 Show how to modify Dijkstra's algorithm to not only output the distance
from v to each vertex in G, but also to output a tree T rooted at v such that
the path in T from v to a vertex u is a shortest path in G from v to u.

R-13.17 There are eight small islands in a lake, and the state wants to build seven
bridges to connect them so that each island can be reached from any other
one via one or more bridges. The cost of constructing a bridge is propor-
tional to its length. The distances between pairs of islands are given in the
following table.

	1	2	3	4	5	6	7	8
1	-	240	210	340	280	200	345	120
2	-	-	265	175	215	180	185	155
3	-	-	-	260	115	350	435	195
4	-	-	-	-	160	330	295	230
5	-	-	-	-	-	360	400	170
6	-	-	-	-	-	-	175	205
7	-	-	-	-	-	-	-	305
8	-	-	-	-	-	-	-	-

Find which bridges to build to minimize the total construction cost.

R-13.18 Draw a simple, connected, undirected, weighted graph with 8 vertices
and 16 edges, each with unique edge weights. Illustrate the execution of
Kruskal's algorithm on this graph. (Note that there is only one minimum
spanning tree for this graph.)

R-13.19 Repeat the previous problem for the Prim-Jarník algorithm.

R-13.20 Consider the unsorted sequence implementation of the priority queue Q
used in Dijkstra's algorithm. In this case, why is this the best-case running
time of Dijkstra's algorithm $O(n^2)$ on an n-vertex graph?

R-13.21 Describe the meaning of the graphical conventions used in Figure 13.6
illustrating a DFS traversal. What do the colors blue and black refer to?
What do the arrows signify? How about thick lines and dashed lines?

R-13.22 Repeat Exercise R-13.21 for Figure 13.7 illustrating a BFS traversal.

R-13.23 Repeat Exercise R-13.21 for Figure 13.9 illustrating a directed DFS traver-
sal.

R-13.24 Repeat Exercise R-13.21 for Figure 13.10 illustrating the Floyd-Warshall
algorithm.

R-13.25 Repeat Exercise R-13.21 for Figure 13.12 illustrating the topological sort-
ing algorithm.

R-13.26 Repeat Exercise R-13.21 for Figures 13.14 and 13.15 illustrating Dijkstra's
algorithm.

R-13.27 Repeat Exercise R-13.21 for Figures 13.18 and 13.20 illustrating Kruskal's algorithm.

R-13.28 Repeat Exercise R-13.21 for Figures 13.21 and 13.22 illustrating the Prim-Jarník algorithm.

R-13.29 How many edges are in the transitive closure of a graph that consists of a simple directed path of n vertices?

R-13.30 Given a complete binary tree T with n nodes, consider a directed graph G having the nodes of T as its vertices. For each parent-child pair in T, create a directed edge in G from the parent to the child. Show that the transitive closure of G has $O(n \log n)$ edges.

R-13.31 A simple undirected graph is **complete** if it contains an edge between every pair of distinct vertices. What does a depth-first search tree of a complete graph look like?

R-13.32 Recalling the definition of a complete graph from Exercise R-13.31, what does a breadth-first search tree of a complete graph look like?

R-13.33 Say that a maze is **constructed correctly** if there is one path from the start to the finish, the entire maze is reachable from the start, and there are no loops around any portions of the maze. Given a maze drawn in an $n \times n$ grid, how can we determine if it is constructed correctly? What is the running time of this algorithm?

Creativity

C-13.1 Say that an n-vertex directed acyclic graph G is **compact** if there is some way of numbering the vertices of G with the integers from 0 to $n-1$ such that G contains the edge (i, j) if and only if $i < j$, for all i, j in $[0, n-1]$. Give an $O(n^2)$-time algorithm for detecting if G is compact.

C-13.2 Describe, in pseudo-code, an $O(n+m)$-time algorithm for computing **all** the connected components of an undirected graph G with n vertices and m edges.

C-13.3 Let T be the spanning tree rooted at the start vertex produced by the depth-first search of a connected, undirected graph G. Argue why every edge of G not in T goes from a vertex in T to one of its ancestors, that is, it is a **back edge**.

C-13.4 Suppose we wish to represent an n-vertex graph G using the edge list structure, assuming that we identify the vertices with the integers in the set $\{0, 1, \ldots, n-1\}$. Describe how to implement the collection E to support $O(\log n)$-time performance for the areAdjacent function. How are you implementing the function in this case?

C-13.5 Tamarindo University and many other schools worldwide are doing a joint project on multimedia. A computer network is built to connect these schools using communication links that form a free tree. The schools decide to install a file server at one of the schools to share data among all the schools. Since the transmission time on a link is dominated by the link setup and synchronization, the cost of a data transfer is proportional to the number of links used. Hence, it is desirable to choose a "central" location for the file server. Given a free tree T and a node v of T, the *eccentricity* of v is the length of a longest path from v to any other node of T. A node of T with minimum eccentricity is called a *center* of T.

  a. Design an efficient algorithm that, given an n-node free tree T, computes a center of T.
  b. Is the center unique? If not, how many distinct centers can a free tree have?

C-13.6 Show that, if T is a BFS tree produced for a connected graph G, then, for each vertex v at level i, the path of T between s and v has i edges, and any other path of G between s and v has at least i edges.

C-13.7 The time delay of a long-distance call can be determined by multiplying a small fixed constant by the number of communication links on the telephone network between the caller and callee. Suppose the telephone network of a company named RT&T is a free tree. The engineers of RT&T want to compute the maximum possible time delay that may be experienced in a long-distance call. Given a free tree T, the *diameter* of T is the length of a longest path between two nodes of T. Give an efficient algorithm for computing the diameter of T.

C-13.8 A company named RT&T has a network of n switching stations connected by m high-speed communication links. Each customer's phone is directly connected to one station in his or her area. The engineers of RT&T have developed a prototype video-phone system that allows two customers to see each other during a phone call. In order to have acceptable image quality, however, the number of links used to transmit video signals between the two parties cannot exceed four. Suppose that RT&T's network is represented by a graph. Design an efficient algorithm that computes, for each station, the set of stations reachable using no more than four links.

C-13.9 Explain why there are no forward nontree edges with respect to a BFS tree constructed for a directed graph.

C-13.10 An *Euler tour* of a directed graph G with n vertices and m edges is a cycle that traverses each edge of G exactly once according to its direction. Such a tour always exists if G is connected and the in-degree equals the out-degree of each vertex in G. Describe an $O(n+m)$-time algorithm for finding an Euler tour of such a digraph G.

C-13.11 An independent set of an undirected graph $G = (V, E)$ is a subset I of V such that no two vertices in I are adjacent. That is, if u and v are in I, then (u, v) is not in E. A ***maximal independent set M*** is an independent set such that, if we were to add any additional vertex to M, then it would not be independent any more. Every graph has a maximal independent set. (Can you see this? This question is not part of the exercise, but it is worth thinking about.) Give an efficient algorithm that computes a maximal independent set for a graph G. What is this algorithm's running time?

C-13.12 Let G be an undirected graph G with n vertices and m edges. Describe an $O(n + m)$-time algorithm for traversing each edge of G exactly once in each direction.

C-13.13 Justify Proposition 13.14.

C-13.14 Give an example of an n-vertex simple graph G that causes Dijkstra's algorithm to run in $\Omega(n^2 \log n)$ time when its implemented with a heap.

C-13.15 Give an example of a weighted directed graph G with negative-weight edges but no negative-weight cycle, such that Dijkstra's algorithm incorrectly computes the shortest-path distances from some start vertex v.

C-13.16 Consider the following greedy strategy for finding a shortest path from vertex *start* to vertex *goal* in a given connected graph.

1. Initialize *path* to *start*.
2. Initialize *VisitedVertices* to {*start*}.
3. If *start=goal*, return *path* and exit. Otherwise, continue.
4. Find the edge (*start,v*) of minimum weight such that v is adjacent to *start* and v is not in *VisitedVertices*.
5. Add v to *path*.
6. Add v to *VisitedVertices*.
7. Set *start* equal to v and go to step 3.

Does this greedy strategy always find a shortest path from *start* to *goal*? Either explain intuitively why it works, or give a counter example.

C-13.17 Show that if all the weights in a connected weighted graph G are distinct, then there is exactly one minimum spanning tree for G.

C-13.18 Design an efficient algorithm for finding a ***longest*** directed path from a vertex s to a vertex t of an acyclic weighted digraph G. Specify the graph representation used and any auxiliary data structures used. Also, analyze the time complexity of your algorithm.

C-13.19 Consider a diagram of a telephone network, which is a graph G whose vertices represent switching centers and whose edges represent communication lines joining pairs of centers. Edges are marked by their bandwidth, and the bandwidth of a path is the bandwidth of its lowest bandwidth edge. Give an algorithm that, given a diagram and two switching centers a and b, outputs the maximum bandwidth of a path between a and b.

C-13.20 Computer networks should avoid single points of failure, that is, network nodes that can disconnect the network if they fail. We say a connected graph G is **biconnected** if it contains no vertex whose removal would divide G into two or more connected components. Give an $O(n+m)$-time algorithm for adding at most n edges to a connected graph G, with $n \geq 3$ vertices and $m \geq n - 1$ edges, to guarantee that G is biconnected.

C-13.21 NASA wants to link n stations spread over the country using communication channels. Each pair of stations has a different bandwidth available, which is known a priori. NASA wants to select $n - 1$ channels (the minimum possible) in such a way that all the stations are linked by the channels and the total bandwidth (defined as the sum of the individual bandwidths of the channels) is maximum. Give an efficient algorithm for this problem and determine its worst-case time complexity. Consider the weighted graph $G = (V, E)$, where V is the set of stations and E is the set of channels between the stations. Define the weight $w(e)$ of an edge e in E as the bandwidth of the corresponding channel.

C-13.22 Suppose you are given a *timetable*, which consists of:

- A set $\mathcal{A}$ of n airports, and for each airport a in $\mathcal{A}$, a minimum connecting time $c(a)$.
- A set $\mathcal{F}$ of m flights, and the following, for each flight f in $\mathcal{F}$:
    - Origin airport $a_1(f)$ in $\mathcal{A}$
    - Destination airport $a_2(f)$ in $\mathcal{A}$
    - Departure time $t_1(f)$
    - Arrival time $t_2(f)$

Describe an efficient algorithm for the flight scheduling problem. In this problem, we are given airports a and b, and a time t, and we wish to compute a sequence of flights that allows one to arrive at the earliest possible time in b when departing from a at or after time t. Minimum connecting times at intermediate airports should be observed. What is the running time of your algorithm as a function of n and m?

C-13.23 Inside the Castle of Asymptopia there is a maze, and along each corridor of the maze there is a bag of gold coins. The amount of gold in each bag varies. A noble knight, named Sir Paul, will be given the opportunity to walk through the maze, picking up bags of gold. He may enter the maze only through a door marked "ENTER" and exit through another door marked "EXIT." While in the maze, he may not retrace his steps. Each corridor of the maze has an arrow painted on the wall. Sir Paul may only go down the corridor in the direction of the arrow. There is no way to traverse a "loop" in the maze. Given a map of the maze, including the amount of gold in and the direction of each corridor, describe an algorithm to help Sir Paul pick up the most gold.

C-13.24 Let G be a weighted digraph with n vertices. Design a variation of Floyd-Warshall's algorithm for computing the lengths of the shortest paths from each vertex to every other vertex in $O(n^3)$ time.

C-13.25 Suppose we are given a directed graph G with n vertices, and let M be the $n \times n$ adjacency matrix corresponding to G.

    a. Let the product of M with itself (M^2) be defined, for $1 \le i, j \le n$, as follows

$$M^2(i, j) = M(i, 1) \odot M(1, j) \oplus \cdots \oplus M(i, n) \odot M(n, j),$$

    where "$\oplus$" is the Boolean **or** operator and "$\odot$" is Boolean **and**. Given this definition, what does $M^2(i, j) = 1$ imply about the vertices i and j? What if $M^2(i, j) = 0$?

    b. Suppose M^4 is the product of M^2 with itself. What do the entries of M^4 signify? How about the entries of $M^5 = (M^4)(M)$? In general, what information is contained in the matrix M^p?

    c. Now suppose that G is weighted and assume the following:

      (a) [1.] For $1 \le i \le n, M(i, i) = 0$.
      (b) [2.] For $1 \le i, j \le n, M(i, j) = weight(i, j)$ if (i, j) is in E.
      (c) [3.] For for $1 \le i, j \le n, M(i, j) = \infty$ if (i, j) is not in E.
    Also, let M^2 be defined, for $1 \le i, j \le n$, as follows

$$M^2(i, j) = \min\{M(i, 1) + M(1, j), \dots, M(i, n) + M(n, j)\}.$$

    If $M^2(i, j) = k$, what may we conclude about the relationship between vertices i and j?

C-13.26 A graph G is **bipartite** if its vertices can be partitioned into two sets X and Y such that every edge in G has one end vertex in X and the other in Y. Design and analyze an efficient algorithm for determining if an undirected graph G is bipartite (without knowing the sets X and Y in advance).

C-13.27 An old MST method, called **Barůvka's algorithm**, works as follows on a graph G having n vertices and m edges with distinct weights.

    Let T be a subgraph of G initially containing just the vertices in V.
    while T has fewer than $n - 1$ edges **do**
      for each connected component C_i of T **do**
        Find the lowest-weight edge (v, u) in E with v in C_i and u not in C_i.
        Add (v, u) to T (unless it is already in T).
    return T

Argue why this algorithm is correct and why it runs in $O(m \log n)$ time.

C-13.28 Let G be a graph with n vertices and m edges such that all the edge weights in G are integers in the range $[1, n]$. Give an algorithm for finding a minimum spanning tree for G in $O(m \log^* n)$ time.

Projects

P-13.1 Write a class implementing a simplified graph ADT that has only functions relevant to undirected graphs and does not include update functions using the adjacency matrix structure. Your class should include a constructor that takes two collections (for example, sequences)—a collection V of vertex elements and a collection E of pairs of vertex elements—and produces the graph G that these two collections represent.

P-13.2 Implement the simplified graph ADT described in Project P-13.1 using the adjacency list structure.

P-13.3 Implement the simplified graph ADT described in Project P-13.1 using the edge list structure.

P-13.4 Extend the class of Project P-13.2 to support all the functions of the graph ADT (including functions for directed edges).

P-13.5 Implement a generic BFS traversal using the template method pattern.

P-13.6 Implement the topological sorting algorithm.

P-13.7 Implement the Floyd-Warshall transitive closure algorithm.

P-13.8 Design an experimental comparison of repeated DFS traversals versus the Floyd-Warshall algorithm for computing the transitive closure of a digraph.

P-13.9 Implement Dijkstra's algorithm assuming that the edge weights are integers.

P-13.10 Implement Kruskal's algorithm assuming that the edge weights are integers.

P-13.11 Implement the Prim-Jarník algorithm assuming that the edge weights are integers.

P-13.12 Perform an experimental comparison of two of the minimum spanning tree algorithms discussed in this chapter (Kruskal and Prim-Jarník). Develop an extensive set of experiments to test the running times of these algorithms using randomly generated graphs.

P-13.13 One way to construct a *maze* starts with an $n \times n$ grid such that each grid cell is bounded by four unit-length walls. We then remove two boundary unit-length walls, to represent the start and finish. For each remaining unit-length wall not on the boundary, we assign a random value and create a graph G, called the *dual*, such that each grid cell is a vertex in G and there is an edge joining the vertices for two cells if and only if the cells share a common wall. The weight of each edge is the weight of the corresponding wall. We construct the maze by finding a minimum spanning tree T for G and removing all the walls corresponding to edges in T. Write a program that uses this algorithm to generate mazes and then

solves them. Minimally, your program should draw the maze and, ideally, it should visualize the solution as well.

P-13.14 Write a program that builds the routing tables for the nodes in a computer network, based on shortest-path routing, where path distance is measured by hop count, that is, the number of edges in a path. The input for this problem is the connectivity information for all the nodes in the network, as in the following example:

241.12.31.14: 241.12.31.15 241.12.31.18 241.12.31.19

which indicates three network nodes that are connected to 241.12.31.14, that is, three nodes that are one hop away. The routing table for the node at address A is a set of pairs (B, C), which indicates that, to route a message from A to B, the next node to send to (on the shortest path from A to B) is C. Your program should output the routing table for each node in the network, given an input list of node connectivity lists, each of which is input in the syntax as shown above, one per line.

Chapter Notes

The depth-first search method is a part of the "folklore" of computer science, but Hopcroft and Tarjan [46, 94] are the ones who showed how useful this algorithm is for solving several different graph problems. Knuth [59] discusses the topological sorting problem. The simple linear-time algorithm that we describe for determining if a directed graph is strongly connected is due to Kosaraju. The Floyd-Warshall algorithm appears in a paper by Floyd [32] and is based upon a theorem of Warshall [102]. To learn about different algorithms for drawing graphs, please see the book chapter by Tamassia and Liotta [92] and the book by Di Battista, Eades, Tamassia and Tollis [28]. The first known minimum spanning tree algorithm is due to Barůvka [8], and was published in 1926. The Prim-Jarník algorithm was first published in Czech by Jarník [50] in 1930 and in English in 1957 by Prim [85]. Kruskal published his minimum spanning tree algorithm in 1956 [62]. The reader interested in further study of the history of the minimum spanning tree problem is referred to the paper by Graham and Hell [41]. The current asymptotically fastest minimum spanning tree algorithm is a randomized algorithm of Karger, Klein, and Tarjan [52] that runs in $O(m)$ expected time.

Dijkstra [29] published his single-source, shortest-path algorithm in 1959. The reader interested in further study of graph algorithms is referred to the books by Ahuja, Magnanti, and Orlin [6], Cormen, Leiserson, and Rivest [24], Even [31], Gibbons [36], Mehlhorn [74], and Tarjan [95], and the book chapter by van Leeuwen [98]. Incidentally, the running time for the Prim-Jarník algorithm, and also that of Dijkstra's algorithm, can actually be improved to be $O(n \log n + m)$ by implementing the queue Q with either of two more sophisticated data structures, the "Fibonacci Heap" [34] or the "Relaxed Heap" [30].

This page intentionally left blank

Chapter

14 Memory Management and B-Trees

Contents

14.1   Memory Management

In order to implement any data structure on an actual computer, we need to use computer memory. Computer memory is simply a sequence of memory *words*, each of which usually consists of 4, 8, or 16 bytes (depending on the computer). These memory words are numbered from 0 to $N - 1$, where N is the number of memory words available to the computer. The number associated with each memory word is known as its *address*. Thus, the memory in a computer can be viewed as basically one giant array of memory words. Using this memory to construct data structures (and run programs) requires that we *manage* the computer's memory to provide the space needed for data—including variables, nodes, pointers, arrays, and character strings—and the programs the computer runs. We discuss the basics of memory management in this section.

The C++ Run-Time Stack

A C++ program is compiled into a binary executable file, which is then executed within the context of the C++ run-time environment. The *run-time environment* provides important functions for executing your program, such as managing memory and performing input and output.

Stacks have an important application to the run-time environment of C++ programs. A running program has a private stack, called the *function call stack* or just *call stack* for short, which is used to keep track of local variables and other important information on functions as they are invoked during execution. (See Figure 14.1.)

More specifically, during the execution of a program, the run-time environment maintains a stack whose elements are descriptors of the currently active (that is, nonterminated) invocations of functions. These descriptors are called *frames*. A frame for some invocation of function "fool" stores the current values of the local variables and parameters of function fool, as well as information on function "cool" that called fool and on what needs to be returned to function "cool."

Keeping Track of the Program Counter

Your computer's run-time system maintains a special variable, called the *program counter*, which keeps track of which machine instruction is currently being executed. When the function cool() invokes another function fool(), the current value of the program counter is recorded in the frame of the current invocation of cool() (so the system knows where to return to when function fool() is done). At the top of the stack is the frame of the *running function*, that is, the function that is currently
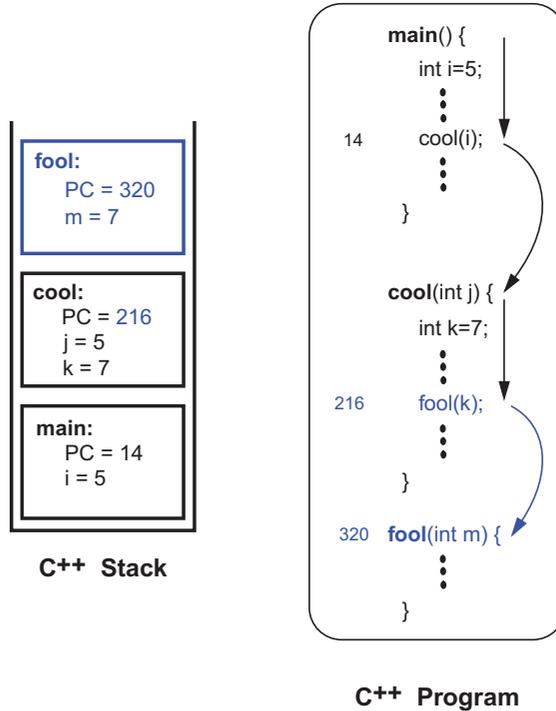
C++ Stack

C++ Program

Figure 14.1: An example of the C++ call stack: function fool has just been called by function cool, which itself was previously called by function main. Note the values of the program counter, parameters, and local variables stored in the stack frames. When the invocation of function fool terminates, the invocation of function cool resumes its execution at instruction 217, which is obtained by incrementing the value of the program counter stored in the stack frame.

executing. The remaining elements of the stack are frames of the *suspended functions*, that is, functions that have invoked another function and are currently waiting for it to return control to them upon its termination. The order of the elements in the stack corresponds to the chain of invocations of the currently active functions. When a new function is invoked, a frame for this function is pushed onto the stack. When it terminates, its frame is popped from the stack and the system resumes the processing of the previously suspended function.

Understanding Call-by-Value Parameter Passing

The system uses the call stack to perform parameter passing to functions. Unless reference parameters are involved, C++ uses the *call-by-value* parameter passing protocol. This means that the current *value* of a variable (or expression) is what is passed as an argument to a called function.

If the variable x being passed is not specified as a reference parameter, its value is copied to a local variable in the called function's frame. This applies to primitive types (such as int and float), pointers (such as "int*"), and even to classes (such as "std::vector<int>"). Note that if the called function changes the value of this local variable, it will ***not*** change the value of the variable in the calling function.

On the other hand, if the variable x is passed as a reference parameter, such as "int&," the address of x is passed instead, and this address is assigned to some local variable y in the called function. Thus, y and x refer to the same object. If the called function changes the internal state of the object that y refers to, it will simultaneously be changing the internal state of the object that x refers to (since they refer to the same object).

C++ arrays behave somewhat differently, however. Recall from Section 1.1.3, that a C++ array is represented internally as a pointer to its first element. Thus, passing an array parameter passes a copy of this pointer, not a copy of the array contents. Since the variable x in the calling function and the associated local variable y in the called function share the same copy of this pointer, $x[i]$ and $y[i]$ refer to the same object in memory.

Implementing Recursion

One of the benefits of using a stack to implement function invocation is that it allows programs to use ***recursion***. That is, it allows a function to call itself, as discussed in Section 3.5. Interestingly, early programming languages, such as Cobol and Fortran, did not originally use run-time stacks to implement function and procedure calls. But because of the elegance and efficiency that recursion allows, all modern programming languages, including the modern versions of classic languages like Cobol and Fortran, utilize a run-time stack for function and procedure calls.

In the execution of a recursive function, each box of the recursion trace corresponds to a frame of the call stack. Also, the content of the call stack corresponds to the chain of boxes from the initial function invocation to the current one.

To better illustrate how a run-time stack allows for recursive functions, let us consider a C++ implementation of the classic recursive definition of the factorial tion

$$n! = n(n-1)(n-2)\cdots 1$$

as shown in Code Fragment 14.1.

The first time we call function factorial, its stack frame includes a local variable storing the value n. Function factorial recursively calls itself to compute $(n-1)!$, which pushes a new frame on the call stack. In turn, this recursive invocation calls itself to compute $(n-2)!$, etc. The chain of recursive invocations, and thus the run-time stack, only grows up to size n, because calling factorial(1) returns

```
int recursiveFactorial(int n) {            // recursive factorial function
  if (n == 0) return 1;                    // basis case
  else return n * recursiveFactorial(n−1); // recursive case
}
```

Code Fragment 14.1: A recursive implementation of the factorial function.

1 immediately without invoking itself recursively. The run-time stack allows for function factorial to exist simultaneously in several active frames (as many as *n* at some point). Each frame stores the value of its parameter *n* as well as the value to be returned. Eventually, when the first recursive call terminates, it returns $(n-1)!$, which is then multiplied by *n* to compute *n*! for the original call of the factorial function.

14.1.1 Memory Allocation in C++

We have already discussed (in Section 14.1) how the C++ run-time system allocates a function's local variables in that function's frame on the run-time stack. The stack is not the only kind of memory available for program data in C++, however. Memory can also be allocated dynamically by using the **new** operator, which is built into C++. For example, in Chapter 1, we learned that we can allocate an array of 100 integers as follows:

```
int* items = new int[100];
```

Memory allocated in this manner can be deallocated with "delete [ ] items."

The Memory Heap

Instead of using the run-time stack for this object's memory, C++ uses memory from another area of storage—the ***memory heap*** (which should not be confused with the "heap" data structure discussed in Chapter 8). We illustrate this memory area, together with the other memory areas, in Figure 14.2. The storage available in the memory heap is divided into ***blocks***, which are contiguous array-like "chunks" of memory that may be of variable or fixed sizes.

To simplify the discussion, let us assume that blocks in the memory heap are of a fixed size, say, $1,024$ bytes, and that one block is big enough for any object we might want to create. (Efficiently handling the more general case is actually an interesting research problem.)

The memory heap must be able to allocate memory blocks quickly for new objects. Different run-time systems use different approaches. We therefore exercise this freedom and choose to use a queue to manage the unused blocks in the memory heap. When a function uses the **new** operator to request a block of memory for

| Program Code | C++ Stack | ➤ Free Memory ◀ | Memory Heap |

fixed size – doesn't grow   grows into higher memory   grows into lower memory
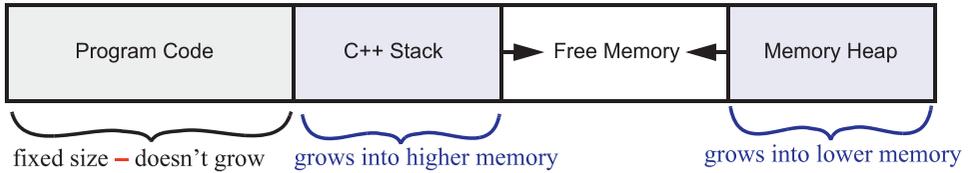
Figure 14.2: A schematic view of the layout of memory in a C++ program.

some new object, the run-time system can perform a dequeue operation on the queue of unused blocks to provide a free block of memory in the memory heap. Likewise, when the user deallocates a block of memory using **delete**, then the run-time system can perform an enqueue operation to return this block to the queue of available blocks.

Memory Allocation Algorithms

It is important that the run-time systems of modern programming languages, such as C++ and Java, are able to quickly allocate memory for new objects. Different systems adopt difference approaches. One popular method is to keep contiguous "holes" of available free memory in a doubly linked list, called the *free list*. The links joining these holes are stored inside the holes themselves, since their memory is not being used. As memory is allocated and deallocated, the collection of holes in the free lists changes, with the unused memory being separated into disjoint holes divided by blocks of used memory. This separation of unused memory into separate holes is known as *fragmentation*. Of course, we would like to minimize fragmentation as much as possible.

There are two kinds of fragmentation that can occur. *Internal fragmentation* occurs when a portion of an allocated memory block is not actually used. For example, a program may request an array of size $1,000$ but only use the first 100 cells of this array. There isn't much that a run-time environment can do to reduce internal fragmentation. *External fragmentation*, on the other hand, occurs when there is a significant amount of unused memory between several contiguous blocks of allocated memory. Since the run-time environment has control over where to allocate memory when it is requested (for example, when the **new** keyword is used in C++), the run-time environment should allocate memory in a way that tries to reduce external fragmentation as much as reasonably possible.

Several heuristics have been suggested for allocating memory from the heap in order to minimize external fragmentation. The *best-fit algorithm* searches the entire free list to find the hole whose size is closest to the amount of memory being requested. The *first-fit algorithm* searches from the beginning of the free list for the first hole that is large enough. The *next-fit algorithm* is similar, in that it also searches the free list for the first hole that is large enough, but it begins its search

from where it left off previously, viewing the free list as a circularly linked list (Section 3.4.1). The ***worst-fit algorithm*** searches the free list to find the largest hole of available memory, which might be done faster than a search of the entire free list if this list were maintained as a priority queue (Chapter 8). In each algorithm, the requested amount of memory is subtracted from the chosen memory hole and the leftover part of that hole is returned to the free list.

Although it might sound good at first, the best-fit algorithm tends to produce the worst external fragmentation, since the leftover parts of the chosen holes tend to be small. The first-fit algorithm is fast, but it tends to produce a lot of external fragmentation at the front of the free list, which slows down future searches. The next-fit algorithm spreads fragmentation more evenly throughout the memory heap, thus keeping search times low. This spreading also makes it more difficult to allocate large blocks, however. The worst-fit algorithm attempts to avoid this problem by keeping contiguous sections of free memory as large as possible.

14.1.2 Garbage Collection

In C++, the memory space for objects must be explicitly allocated and deallocated by the programmer through the use of the operators **new** and **delete**, respectively. Other programming languages, such as Java, place the burden of memory management entirely on the run-time environment. In this section, we discuss how the run-time systems of languages like Java manage the memory used by objects allocated by the **new** operation.

As mentioned above, memory for objects is allocated from the memory heap and the space for the member variables of a running program are placed in its call stacks, one for each running program. Since member variables in a call stack can refer to objects in the memory heap, all the variables and objects in the call stacks of running threads are called ***root objects***. All those objects that can be reached by following object references that start from a root object are called ***live objects***. The live objects are the active objects currently being used by the running program; these objects should ***not*** be deallocated. For example, a running program may store, in a variable, a reference to a sequence S that is implemented using a doubly linked list. The reference variable to S is a root object, while the object for S is a live object, as are all the node objects that are referenced from this object and all the elements that are referenced from these node objects.

From time to time, the run-time environment may notice that available space in the memory heap is becoming scarce. At such times, the system can elect to reclaim the space that is being used for objects that are no longer live, and return the reclaimed memory to the free list. This reclamation process is known as ***garbage collection***. There are several different algorithms for garbage collection, but one of the most used is the ***mark-sweep algorithm***.

In the mark-sweep garbage collection algorithm, we associate a "mark" bit with each object that identifies if that object is live or not. When we determine, at some point, that garbage collection is needed, we suspend all other running threads and clear the mark bits of all the objects currently allocated in the memory heap. We then trace through the call stack of the currently running program and we mark all the (root) objects in this stack as "live." We must then determine all the other live objects—the ones that are reachable from the root objects. To do this efficiently, we can use the directed-graph version of the depth-first search traversal (Section 13.3.1). In this case, each object in the memory heap is viewed as a vertex in a directed graph, and the reference from one object to another is viewed as a directed edge. By performing a directed DFS from each root object, we can correctly identify and mark each live object. This process is known as the "mark" phase. Once this process has completed, we then scan through the memory heap and reclaim any space that is being used for an object that has not been marked. At this time, we can also optionally coalesce all the allocated space in the memory heap into a single block, thereby eliminating external fragmentation for the time being. This scanning and reclamation process is known as the "sweep" phase, and when it completes, we resume running the suspended threads. Thus, the mark-sweep garbage collection algorithm will reclaim unused space in time proportional to the number of live objects and their references plus the size of the memory heap.

Performing DFS In-place

The mark-sweep algorithm correctly reclaims unused space in the memory heap, but there is an important issue we must face during the mark phase. Since we are reclaiming memory space at a time when available memory is scarce, we must take care not to use extra space during the garbage collection itself. The trouble is that the DFS algorithm, in the recursive way we described it in Section 13.3.1, can use space proportional to the number of vertices in the graph. In the case of garbage collection, the vertices in our graph are the objects in the memory heap; hence, we probably don't have this much memory to use. We want a way to perform DFS in-place, using only a constant amount of additional storage.

The main idea for performing DFS in-place is to simulate the recursion stack using the edges of the graph (which, in the case of garbage collection, corresponds to object references). When we traverse an edge from a visited vertex v to a new vertex w, we change the edge (v, w) stored in v's adjacency list to point back to v's parent in the DFS tree. When we return back to v (simulating the return from the "recursive" call at w), we can now switch the edge we modified to point back to w. Of course, we need to have some way of identifying which edge we need to change back. One possibility is to number the references going out of v as 1, 2, and so on, and store, in addition to the mark bit (which we are using for the "visited" tag in our DFS), a count identifier that tells us which edges we have modified.

14.2 External Memory and Caching

There are several computer applications that must deal with a large amount of data. Examples include the analysis of scientific data sets, the processing of financial transactions, and the organization and maintenance of databases (such as telephone directories). In fact, the amount of data that must be dealt with is often too large to fit entirely in the internal memory of a computer.

14.2.1 The Memory Hierarchy

In order to accommodate large data sets, computers have a *hierarchy* of different kinds of memories that vary in terms of their size and distance from the CPU. Closest to the CPU are the internal registers that the CPU itself uses. Access to such locations is very fast, but there are relatively few such locations. At the second level in the hierarchy is the *cache* memory. This memory is considerably larger than the register set of a CPU, but accessing it takes longer (and there may even be multiple caches with progressively slower access times). At the third level in the hierarchy is the *internal memory*, which is also known as *main memory* or *core memory*. The internal memory is considerably larger than the cache memory, but also requires more time to access. Finally, at the highest level in the hierarchy is the *external memory*, which usually consists of disks, CD drives, DVD drives, and/or tapes. This memory is very large, but it is also very slow. Thus, the memory hierarchy for computers can be viewed as consisting of four levels, each of which is larger and slower than the previous level. (See Figure 14.3.)

In most applications, however, only two levels really matter—the one that can hold all data items and the level just below that one. Bringing data items in and out of the higher memory that can hold all items will typically be the computational bottleneck in this case.
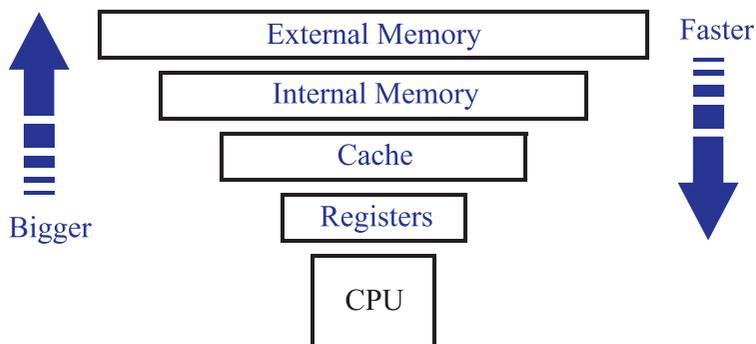
Figure 14.3: The memory hierarchy.

Caches and Disks

Specifically, the two levels that matter most depend on the size of the problem we are trying to solve. For a problem that can fit entirely in main memory, the two most important levels are the cache memory and the internal memory. Access times for internal memory can be as much as 10 to 100 times longer than those for cache memory. It is desirable, therefore, to be able to perform most memory accesses in cache memory. For a problem that does not fit entirely in main memory, on the other hand, the two most important levels are the internal memory and the external memory. Here the differences are even more dramatic. For access times for disks, the usual general-purpose, external-memory devices, are typically as much as $100,000$ to $1,000,000$ times longer than those for internal memory.

To put this latter figure into perspective, imagine there is a student in Baltimore who wants to send a request-for-money message to his parents in Chicago. If the student sends his parents an e-mail message, it can arrive at their home computer in about five seconds. Think of this mode of communication as corresponding to an internal-memory access by a CPU. A mode of communication corresponding to an external-memory access that is $500,000$ times slower would be for the student to walk to Chicago and deliver his message in person, which would take about a month if he can average 20 miles per day. Thus, we should make as few accesses to external memory as possible.

14.2.2 Caching Strategies

Most algorithms are not designed with the memory hierarchy in mind, in spite of the great variance between access times for the different levels. Indeed, all of the algorithm analyses described in this book so far have assumed that all memory accesses are equal. This assumption might seem, at first, to be a great oversight—and one we are only addressing now in the final chapter—but there are good reasons why it is actually a reasonable assumption to make.

One justification for this assumption is that it is often necessary to assume that all memory accesses take the same amount of time, since specific device-dependent information about memory sizes is often hard to come by. In fact, information about memory size may be impossible to get. For example, a C++ program that is designed to run on many different computer platforms cannot be defined in terms of a specific computer architecture configuration. We can certainly use architecture-specific information, if we have it (and we show how to exploit such information later in this chapter). But once we have optimized our software for a certain architecture configuration, our software is no longer device-independent. Fortunately, such optimizations are not always necessary, primarily because of the second justification for the equal-time, memory-access assumption.

Caching and Blocking

Another justification for the memory-access equality assumption is that operating system designers have developed general mechanisms that allow for most memory accesses to be fast. These mechanisms are based on two important *locality-of-reference* properties that most software possesses.

- **Temporal locality**: If a program accesses a certain memory location, then it is likely to access this location again in the near future. For example, it is quite common to use the value of a counter variable in several different expressions, including one to increment the counter's value. In fact, a common adage among computer architects is that "a program spends 90 percent of its time in 10 percent of its code."
- **Spatial locality**: If a program accesses a certain memory location, then it is likely to access other locations that are near this one. For example, a program using an array is likely to access the locations of this array in a sequential or near-sequential manner.

Computer scientists and engineers have performed extensive software profiling experiments to justify the claim that most software possesses both of these kinds of locality-of-reference. For example, a for-loop used to scan through an array exhibits both kinds of locality.

Temporal and spatial localities have, in turn, given rise to two fundamental design choices for two-level computer memory systems (which are present in the interface between cache memory and internal memory, and also in the interface between internal memory and external memory).

The first design choice is called *virtual memory*. This concept consists of providing an address space as large as the capacity of the secondary-level memory, and of transferring data located in the secondary level, into the primary level, when they are addressed. Virtual memory does not limit the programmer to the constraint of the internal memory size. The concept of bringing data into primary memory is called *caching*, and it is motivated by temporal locality. Because, by bringing data into primary memory, we are hoping that it will be accessed again soon, and we will be able to respond quickly to all the requests for this data that come in the near future.

The second design choice is motivated by spatial locality. Specifically, if data stored at a secondary-level memory location l is accessed, then we bring into primary-level memory, a large block of contiguous locations that include the location l. (See Figure 14.4.) This concept is known as *blocking*, and it is motivated by the expectation that other secondary-level memory locations close to l will soon be accessed. In the interface between cache memory and internal memory, such blocks are often called *cache lines*, and in the interface between internal memory and external memory, such blocks are often called *pages*.

A block on disk

A block in the external memory address space

0 1 2 3 ...   1024 ...   2048 ...
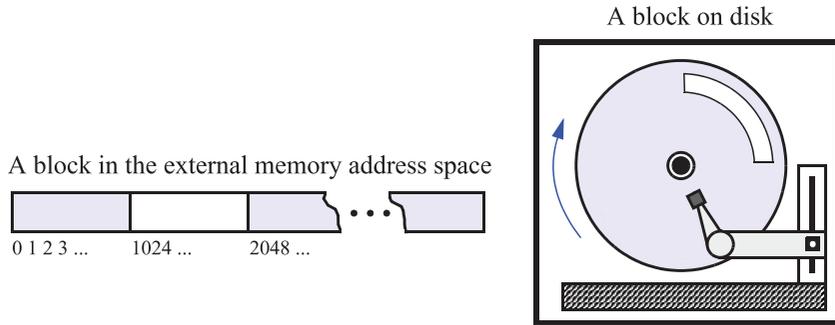
Figure 14.4: Blocks in external memory.

When implemented with caching and blocking, virtual memory often allows us to perceive secondary-level memory as being faster than it really is. There is still a problem, however. Primary-level memory is much smaller than secondary-level memory. Moreover, because memory systems use blocking, any program of substance will likely reach a point where it requests data from secondary-level memory, but the primary memory is already full of blocks. In order to fulfill the request and maintain our use of caching and blocking, we must remove some block from primary memory to make room for a new block from secondary memory in this case. Deciding how to do this eviction brings up a number of interesting data structure and algorithm design issues.

Caching Algorithms

There are several Web applications that must deal with revisiting information presented in Web pages. These revisits have been shown to exhibit localities of reference, both in time and in space. To exploit these localities of reference, it is often advantageous to store copies of Web pages in a *cache* memory, so these pages can be quickly retrieved when requested again. In particular, suppose we have a cache memory that has m "slots" that can contain Web pages. We assume that a Web page can be placed in any slot of the cache. This is known as a *fully associative* cache.

As a browser executes, it requests different Web pages. Each time the browser requests such a Web page l, the browser determines (using a quick test) if l is unchanged and currently contained in the cache. If l is contained in the cache, then the browser satisfies the request using the cached copy. If l is not in the cache, however, the page for l is requested over the Internet and transferred into the cache. If one of the m slots in the cache is available, then the browser assigns l to one of the empty slots. But if all the m cells of the cache are occupied, then the computer must determine which previously viewed Web page to evict before bringing in l to take its place. There are, of course, many different policies that can be used to determine the page to evict.

Page Replacement Algorithms

Some of the better-known page replacement policies include the following (see Figure 14.5):

- **First-in, fist-out (FIFO)** : Evict the page that has been in the cache the longest, that is, the page that was transferred to the cache furthest in the past.
- **Least recently used (LRU)**: Evict the page whose last request occurred furthest in the past.

In addition, we can consider a simple and purely random strategy:

- **Random**: Choose a page at random to evict from the cache.
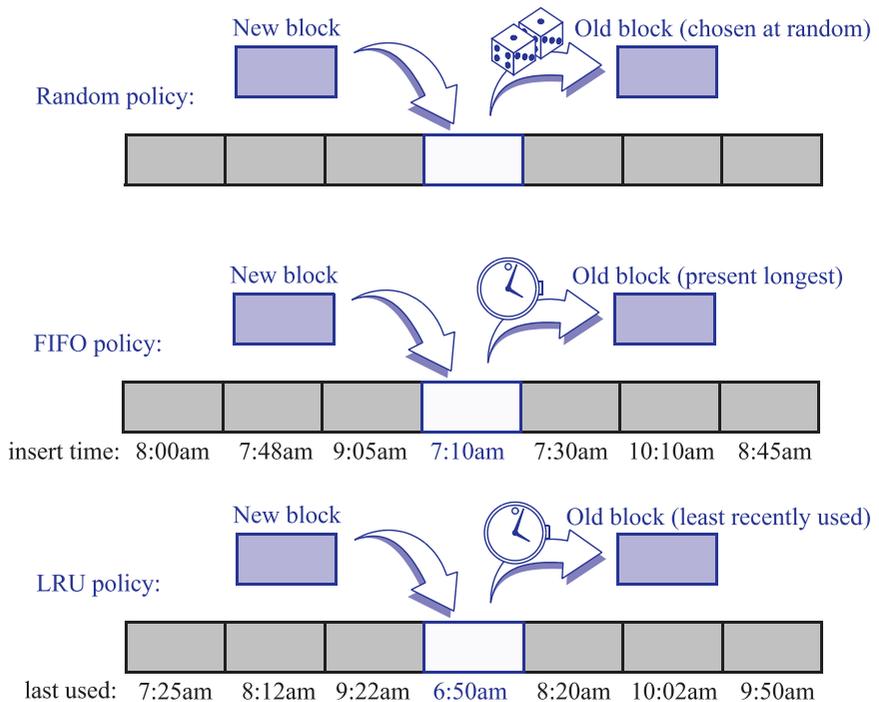
Figure 14.5: The Random, FIFO, and LRU page replacement policies.

The Random strategy is one of the easiest policies to implement, because it only requires a random or pseudo-random number generator. The overhead involved in implementing this policy is an $O(1)$ additional amount of work per page replacement. Moreover, there is no additional overhead for each page request, other than to determine whether a page request is in the cache or not. Still, this policy makes no attempt to take advantage of any temporal or spatial localities that a user's browsing exhibits.

The FIFO strategy is quite simple to implement, because it only requires a queue Q to store references to the pages in the cache. Pages are enqueued in Q when they are referenced by a browser, and then are brought into the cache. When a page needs to be evicted, the computer simply performs a dequeue operation on Q to determine which page to evict. Thus, this policy also requires $O(1)$ additional work per page replacement. Also, the FIFO policy incurs no additional overhead for page requests. Moreover, it tries to take some advantage of temporal locality.

The LRU strategy goes a step further than the FIFO strategy, since the LRU strategy explicitly takes advantage of temporal locality as much as possible, by always evicting the page that was least recently used. From a policy point of view, this is an excellent approach, but it is costly from an implementation point of view. That is, its way of optimizing temporal and spatial locality is fairly costly. Implementing the LRU strategy requires the use of a priority queue Q that supports searching for existing pages, for example, using special pointers or "locators." If Q is implemented with a sorted sequence based on a linked list, then the overhead for each page request and page replacement is $O(1)$. When we insert a page in Q or update its key, the page is assigned the highest key in Q and is placed at the end of the list, which can also be done in $O(1)$ time. Even though the LRU strategy has constant-time overhead, using the implementation above, the constant factors involved, in terms of the additional time overhead and the extra space for the priority queue Q, make this policy less attractive from a practical point of view.

Since these different page replacement policies have different trade-offs between implementation difficulty and the degree to which they seem to take advantage of localities, it is natural for us to ask for some kind of comparative analysis of these methods to see which one, if any, is the best.

From a worst-case point of view, the FIFO and LRU strategies have fairly unattractive competitive behavior. For example, suppose we have a cache containing m pages, and consider the FIFO and LRU methods for performing page replacement for a program that has a loop that repeatedly requests $m+1$ pages in a cyclic order. Both the FIFO and LRU policies perform badly on such a sequence of page requests, because they perform a page replacement on every page request. Thus, from a worst-case point of view, these policies are almost the worst we can imagine—they require a page replacement on every page request.

This worst-case analysis is a little too pessimistic, however, for it focuses on each protocol's behavior for one bad sequence of page requests. An ideal analysis would be to compare these methods over all possible page-request sequences. Of course, this is impossible to do exhaustively, but there have been a great number of experimental simulations done on page-request sequences derived from real programs. Based on these experimental comparisons, the LRU strategy has been shown to be usually superior to the FIFO strategy, which is usually better than the Random strategy.

14.3 External Searching and B-Trees

Consider the problem of implementing the map ADT for a large collection of items that do not fit in main memory. Since one of the main uses of a large map is in a database, we refer to the secondary-memory blocks as **disk blocks**. Likewise, we refer to the transfer of a block between secondary memory and primary memory as a **disk transfer**. Recalling the great time difference that exists between main memory accesses and disk accesses, the main goal of maintaining a map in external memory is to minimize the number of disk transfers needed to perform a query or update. In fact, the difference in speed between disk and internal memory is so great that we should be willing to perform a considerable number of internal-memory accesses if they allow us to avoid a few disk transfers. Let us, therefore, analyze the performance of map implementations by counting the number of disk transfers each would require to perform the standard map search and update operations. We refer to this count as the **I/O complexity** of the algorithms involved.

Some Inefficient External-Memory Dictionaries

Let us first consider the simple map implementations that use a list to store n entries. If the list is implemented as an unsorted, doubly linked list, then insert and remove operations can be performed with $O(1)$ transfers each, but removals and searches require n transfers in the worst case, since each link hop we perform could access a different block. This search time can be improved to $O(n/B)$ transfers (see Exercise C-14.2), where B denotes the number of nodes of the list that can fit into a block, but this is still poor performance. We could alternately implement the sequence using a sorted array. In this case, a search performs $O(\log_2 n)$ transfers, via binary search, which is a nice improvement. But this solution requires $\Theta(n/B)$ transfers to implement an insert or remove operation in the worst case, because we may have to access all blocks to move elements up or down. Thus, list-based map implementations are not efficient in external memory.

Since these simple implementations are I/O inefficient, we should consider the logarithmic-time, internal-memory strategies that use balanced binary trees (for example, AVL trees or red-black trees) or other search structures with logarithmic average-case query and update times (for example, skip lists or splay trees). These methods store the map items at the nodes of a binary tree or of a graph. Typically, each node accessed for a query or update in one of these structures will be in a different block. Thus, these methods all require $O(\log_2 n)$ transfers in the worst case to perform a query or update operation. This performance is good, but we can do better. In particular, we can perform map queries and updates using only $O(\log_B n) = O(\log n / \log B)$ transfers.

14.3.1 (a, b) Trees

To reduce the importance of the performance difference between internal-memory accesses and external-memory accesses for searching, we can represent our map using a multi-way search tree (Section 10.4.1). This approach gives rise to a generalization of the $(2, 4)$ tree data structure known as the (a, b) tree.

An (a, b) tree is a multi-way search tree such that each node has between a and b children and stores between $a - 1$ and $b - 1$ entries. The algorithms for searching, inserting, and removing entries in an (a, b) tree are straightforward generalizations of the corresponding algorithms for $(2, 4)$ trees. The advantage of generalizing $(2, 4)$ trees to (a, b) trees is that a generalized class of trees provides a flexible search structure, where the size of the nodes and the running time of the various map operations depends on the parameters a and b. By setting the parameters a and b appropriately with respect to the size of disk blocks, we can derive a data structure that achieves good external-memory performance.

Definition of an (a, b) Tree

An (a, b) **tree**, where a and b are integers, such that $2 \leq a \leq (b + 1)/2$, is a multi-way search tree T with the following additional restrictions:

Size Property: Each internal node has at least a children, unless it is the root, and has at most b children.

Depth Property: All the external nodes have the same depth.

Proposition 14.1: *The height of an (a, b) tree storing n entries is $\Omega(\log n / \log b)$ and $O(\log n / \log a)$.*

Justification: Let T be an (a, b) tree storing n entries, and let h be the height of T. We justify the proposition by establishing the following bounds on h

$$\frac{1}{\log b} \log(n + 1) \leq h \leq \frac{1}{\log a} \log \frac{n + 1}{2} + 1.$$

By the size and depth properties, the number n'' of external nodes of T is at least $2a^{h-1}$ and at most b^h. By Proposition 10.7, $n'' = n + 1$. Thus

$$2a^{h-1} \leq n + 1 \leq b^h.$$

Taking the logarithm in base 2 of each term, we get

$$(h - 1) \log a + 1 \leq \log(n + 1) \leq h \log b.$$

Search and Update Operations

We recall that in a multi-way search tree T, each node v of T holds a secondary structure $M(v)$, which is itself a map (Section 10.4.1). If T is an (a,b) tree, then $M(v)$ stores at most b entries. Let $f(b)$ denote the time for performing a search in a map, $M(v)$. The search algorithm in an (a,b) tree is exactly like the one for multi-way search trees given in Section 10.4.1. Hence, searching in an (a,b) tree T with n entries takes $O((f(b)/\log a)\log n)$ time. Note that if b is a constant (and thus a is also), then the search time is $O(\log n)$.

The main application of (a,b) trees is for maps stored in external memory. Namely, to minimize disk accesses, we select the parameters a and b so that each tree node occupies a single disk block (so that $f(b) = 1$ if we wish to simply count block transfers). Providing the right a and b values in this context gives rise to a data structure known as the B-tree, which we describe shortly. Before we describe this structure, however, let us discuss how insertions and removals are handled in (a,b) trees.

The insertion algorithm for an (a,b) tree is similar to that for a $(2,4)$ tree. An overflow occurs when an entry is inserted into a b-node v, which becomes an illegal $(b+1)$-node. (Recall that a node in a multi-way tree is a d-node if it has d children.) To remedy an overflow, we split node v by moving the median entry of v into the parent of v and replacing v with a $\lceil (b+1)/2 \rceil$-node v' and a $\lfloor (b+1)/2 \rfloor$-node v''. We can now see the reason for requiring $a \le (b+1)/2$ in the definition of an (a,b) tree. Note that, as a consequence of the split, we need to build the secondary structures $M(v')$ and $M(v'')$.

Removing an entry from an (a,b) tree is similar to what was done for $(2,4)$ trees. An underflow occurs when a key is removed from an a-node v, distinct from the root, which causes v to become an illegal $(a-1)$-node. To remedy an underflow, we perform a transfer with a sibling of v that is not an a-node or we perform a fusion of v with a sibling that is an a-node. The new node w resulting from the fusion is a $(2a-1)$-node, which is another reason for requiring $a \le (b+1)/2$.

Table 14.1 shows the performance of a map realized with an (a,b) tree.

Operation	Time
find	$O\left(\dfrac{f(b)}{\log a} \log n \right)$
insert	$O\left(\dfrac{g(b)}{\log a} \log n \right)$
erase	$O\left(\dfrac{g(b)}{\log a} \log n \right)$

Table 14.1: Time bounds for an n-entry map realized by an (a,b) tree T. We assume the secondary structure of the nodes of T support search in $f(b)$ time, and split and fusion operations in $g(b)$ time, for some functions $f(b)$ and $g(b)$, which can be made to be $O(1)$ when we are only counting disk transfers.

14.3.2   B-Trees

A version of the (a,b) tree data structure, which is the best known method for maintaining a map in external memory, is called the "B-tree." (See Figure 14.6.) A **B-tree of order** d is an (a,b) tree with $a = \lceil d/2 \rceil$ and $b = d$. Since we discussed the standard map query and update methods for (a,b) trees above, we restrict our discussion here to the I/O complexity of B-trees.
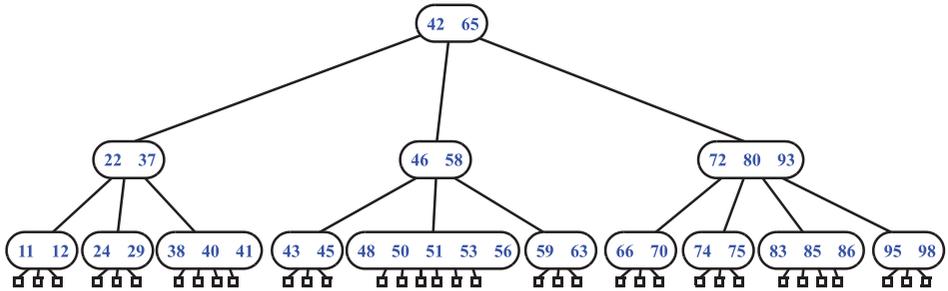
Figure 14.6: A B-tree of order 6.

An important property of B-trees is that we can choose d so that the d children references and the $d - 1$ keys stored at a node can all fit into a single disk block, implying that d is proportional to B. This choice allows us to assume that a and b are also proportional to B in the analysis of the search and update operations on (a,b) trees. Thus, $f(b)$ and $g(b)$ are both $O(1)$, because each time we access a node to perform a search or an update operation, we need only perform a single disk transfer.

As we have already observed above, each search or update requires that we examine at most $O(1)$ nodes for each level of the tree. Therefore, any map search or update operation on a B-tree requires only $O(\log_{\lceil d/2 \rceil} n)$, that is, $O(\log n / \log B)$ disk transfers. For example, an insert operation proceeds down the B-tree to locate the node in which to insert the new entry. If the node **overflows** (to have $d + 1$ children) because of this addition, then this node is **split** into two nodes that have $\lfloor (d+1)/2 \rfloor$ and $\lceil (d+1)/2 \rceil$ children, respectively. This process is then repeated at the next level up, and continues for at most $O(\log_B n)$ levels.

Likewise, if a remove operation results in a node **underflow** (to have $\lceil d/2 \rceil - 1$ children), then we move references from a sibling node with at least $\lceil d/2 \rceil + 1$ children or we need to perform a **fusion** operation of this node with its sibling (and repeat this computation at the parent). As with the insert operation, this continues up the B-tree for at most $O(\log_B n)$ levels. The requirement that each internal node has at least $\lceil d/2 \rceil$ children implies that each disk block used to support a B-tree is at least half full. Thus, we have the following.

Proposition 14.2: *A B-tree with n entries has I/O complexity $O(\log_B n)$ for search or update operation, and uses $O(n/B)$ blocks, where B is the size of a block.*

14.4 External-Memory Sorting

In addition to data structures, such as maps, that need to be implemented in external memory, there are many algorithms that must also operate on input sets that are too large to fit entirely into internal memory. In this case, the objective is to solve the algorithmic problem using as few block transfers as possible. The most classic domain for such external-memory algorithms is the sorting problem.

Multi-Way Merge-Sort

An efficient way to sort a set S of n objects in external memory amounts to a simple external-memory variation on the familiar merge-sort algorithm. The main idea behind this variation is to merge many recursively sorted lists at a time, thereby reducing the number of levels of recursion. Specifically, a high-level description of this ***multi-way merge-sort*** method is to divide S into d subsets S_1, S_2, ..., S_d of roughly equal size, recursively sort each subset S_i, and then simultaneously merge all d sorted lists into a sorted representation of S. If we can perform the merge process using only $O(n/B)$ disk transfers, then, for large enough values of n, the total number of transfers performed by this algorithm satisfies the following recurrence

$$t(n) = d \cdot t(n/d) + cn/B,$$

for some constant $c \geq 1$. We can stop the recursion when $n \leq B$, since we can perform a single block transfer at this point, getting all of the objects into internal memory, and then sort the set with an efficient internal-memory algorithm. Thus, the stopping criterion for $t(n)$ is

$$t(n) = 1 \quad \text{if } n/B \leq 1.$$

This implies a closed-form solution that $t(n)$ is $O((n/B)\log_d(n/B))$, which is

$$O((n/B)\log(n/B)/\log d).$$

Thus, if we can choose d to be $\Theta(M/B)$, then the worst-case number of block transfers performed by this multi-way merge-sort algorithm is quite low. We choose

$$d = (1/2)M/B.$$

The only aspect of this algorithm left to specify is how to perform the d-way merge using only $O(n/B)$ block transfers.

14.4.1 Multi-Way Merging

We perform the d-way merge by running a "tournament." We let T be a complete binary tree with d external nodes, and we keep T entirely in internal memory. We associate each external node i of T with a different sorted list S_i. We initialize T by reading into each external node i, the first object in S_i. This has the effect of reading into internal memory the first block of each sorted list S_i. For each internal-node parent v of two external nodes, we then compare the objects stored at v's children and we associate the smaller of the two with v. We repeat this comparison test at the next level up in T, and the next, and so on. When we reach the root r of T, we associate the smallest object from among all the lists with r. This completes the initialization for the d-way merge. (See Figure 14.7.)
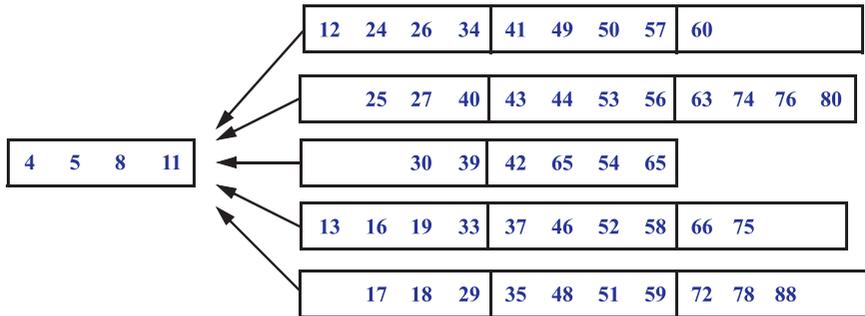
Figure 14.7: A d-way merge. We show a five-way merge with $B = 4$.

In a general step of the d-way merge, we move the object o associated with the root r of T into an array we are building for the merged list S'. We then trace down T, following the path to the external node i that o came from. We then read into i the next object in the list S_i. If o was not the last element in its block, then this next object is already in internal memory. Otherwise, we read in the next block of S_i to access this new object (if S_i is now empty, associate the node i with a pseudo-object with key $+\infty$). We then repeat the minimum computations for each of the internal nodes from i to the root of T. This again gives us the complete tree T. We then repeat this process of moving the object from the root of T to the merged list S', and rebuilding T, until T is empty of objects. Each step in the merge takes $O(\log d)$ time; hence, the internal time for the d-way merge is $O(n \log d)$. The number of transfers performed in a merge is $O(n/B)$, since we scan each list S_i in order once, and we write out the merged list S' once. Thus, we have:

Proposition 14.3: *Given an array-based sequence S of n elements stored in external memory, we can sort S using $O((n/B) \log(n/B) / \log(M/B))$ transfers and $O(n \log n)$ internal CPU time, where M is the size of the internal memory and B is the size of a block.*

14.5 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-14.1 Julia just bought a new computer that uses 64-bit integers to address memory cells. Argue why Julia will never in her life be able to upgrade the main memory of her computer so that it is the maximum size possible, assuming that you have to have distinct atoms to represent different bits.

R-14.2 Describe, in detail, add and remove algorithms for an (a, b) tree.

R-14.3 Suppose T is a multi-way tree in which each internal node has at least five and at most eight children. For what values of a and b is T a valid (a, b) tree?

R-14.4 For what values of d is the tree T of the previous exercise an order-d B-tree?

R-14.5 Show each level of recursion in performing a four-way, external-memory merge-sort of the sequence given in the previous exercise.

R-14.6 Consider an initially empty memory cache consisting of four pages. How many page misses does the LRU algorithm incur on the following page request sequence: $(2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3)$?

R-14.7 Consider an initially empty memory cache consisting of four pages. How many page misses does the FIFO algorithm incur on the following page request sequence: $(2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3)$?

R-14.8 Consider an initially empty memory cache consisting of four pages. How many page misses can the random algorithm incur on the following page request sequence: $(2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3)$? Show all of the random choices your algorithm made in this case.

R-14.9 Draw the result of inserting, into an initially empty order-7 B-tree, the keys $(4, 40, 23, 50, 11, 34, 62, 78, 66, 22, 90, 59, 25, 72, 64, 77, 39, 12)$.

R-14.10 Show each level of recursion in performing a four-way merge-sort of the sequence given in the previous exercise.

Creativity

C-14.1 Describe an efficient external-memory algorithm for removing all the duplicate entries in a vector of size n.

C-14.2 Show how to implement a map in external memory using an unordered sequence so that insertions require only $O(1)$ transfers and searches require $O(n/B)$ transfers in the worst case, where n is the number of elements and B is the number of list nodes that can fit into a disk block.

C-14.3 Change the rules that define red-black trees so that each red-black tree T has a corresponding $(4, 8)$ tree and vice versa.

C-14.4 Describe a modified version of the B-tree insertion algorithm so that each time we create an overflow because of a split of a node v, we redistribute keys among all of v's siblings, so that each sibling holds roughly the same number of keys (possibly cascading the split up to the parent of v). What is the minimum fraction of each block that will always be filled using this scheme?

C-14.5 Another possible external-memory map implementation is to use a skip list, but to collect consecutive groups of $O(B)$ nodes, in individual blocks, on any level in the skip list. In particular, we define an *order-d B-skip list* to be such a representation of a skip-list structure, where each block contains at least $\lceil d/2 \rceil$ list nodes and at most d list nodes. Let us also choose d in this case to be the maximum number of list nodes from a level of a skip list that can fit into one block. Describe how we should modify the skip-list insertion and removal algorithms for a B-skip list so that the expected height of the structure is $O(\log n/\log B)$.

C-14.6 Describe an external-memory data structure to implement the queue ADT so that the total number of disk transfers needed to process a sequence of n enqueue and dequeue operations is $O(n/B)$.

C-14.7 Solve the previous problem for the deque ADT.

C-14.8 Describe how to use a B-tree to implement the partition (union-find) ADT (from Section 11.4.3) so that the union and find operations each use at most $O(\log n/\log B)$ disk transfers.

C-14.9 Suppose we are given a sequence S of n elements with integer keys such that some elements in S are colored "blue" and some elements in S are colored "red." In addition, say that a red element e *pairs* with a blue element f if they have the same key value. Describe an efficient external-memory algorithm for finding all the red-blue pairs in S. How many disk transfers does your algorithm perform?

C-14.10 Consider the page caching problem where the memory cache can hold m pages, and we are given a sequence P of n requests taken from a pool of $m + 1$ possible pages. Describe the optimal strategy for the offline algorithm and show that it causes at most $m + n/m$ page misses in total, starting from an empty cache.

C-14.11 Consider the page caching strategy based on the ***least frequently used*** (LFU) rule, where the page in the cache that has been accessed the least often is the one that is evicted when a new page is requested. If there are ties, LFU evicts the least frequently used page that has been in the cache the longest. Show that there is a sequence P of n requests that causes LFU to miss $\Omega(n)$ times for a cache of m pages, whereas the optimal algorithm will miss only $O(m)$ times.

C-14.12 Suppose that instead of having the node-search function $f(d) = 1$ in an order-d B-tree T, we have $f(d) = \log d$. What does the asymptotic running time of performing a search in T now become?

C-14.13 Describe an efficient external-memory algorithm that determines whether an array of n integers contains a value occurring more than $n/2$ times.

Projects

P-14.1 Write a C++ class that simulates the best-fit, worst-fit, first-fit, and next-fit algorithms for memory management. Determine experimentally which method is the best under various sequences of memory requests.

P-14.2 Write a C++ class that implements all the functions of the ordered map ADT by means of an (a, b) tree, where a and b are integer constants passed as parameters to a constructor.

P-14.3 Implement the B-tree data structure, assuming a block size of $1,024$ and integer keys. Test the number of "disk transfers" needed to process a sequence of map operations.

P-14.4 Implement an external-memory sorting algorithm and compare it experimentally to any internal-memory sorting algorithm.

Chapter Notes

The mark-sweep garbage collection method we describe is one of many different algorithms for performing garbage collection. We encourage the reader interested in further study of garbage collection to examine the book by Jones [51].

Knuth [57] has very nice discussions about external-memory sorting and searching, and Ullman [97] discusses external memory structures for database systems. The reader interested in the study of the architecture of hierarchical memory systems is referred to the book chapter by Burger *et al.* [18] or the book by Hennessy and Patterson [44]. The handbook by Gonnet and Baeza-Yates [37] compares the performance of a number of different sorting algorithms, many of which are external-memory algorithms.

B-trees were invented by Bayer and McCreight [10] and Comer [23] provides a very nice overview of this data structure. The books by Mehlhorn [73] and Samet [87] also have nice discussions about B-trees and their variants. Aggarwal and Vitter [2] study the I/O

complexity of sorting and related problems, establishing upper and lower bounds, including the lower bound for sorting given in this chapter. Goodrich *et al.* [40] study the I/O complexity of several computational geometry problems. The reader interested in further study of I/O-efficient algorithms is encouraged to examine the survey paper of Vitter [99].

Appendix

A

Useful Mathematical Facts

In this appendix, we give several useful mathematical facts. We begin with some combinatorial definitions and facts.

Logarithms and Exponents

The logarithm function is defined as

$$\log_b a = c \qquad \text{if} \qquad a = b^c.$$

The following identities hold for logarithms and exponents:

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c \log_b a$
4. $\log_b a = (\log_c a)/\log_c b$
5. $b^{\log_c a} = a^{\log_c b}$
6. $(b^a)^c = b^{ac}$
7. $b^a b^c = b^{a+c}$
8. $b^a / b^c = b^{a-c}$

In addition, we have the following.

Proposition A.1: *If $a > 0$, $b > 0$, and $c > a+b$, then*

$$\log a + \log b \leq 2\log c - 2.$$

Justification: It is enough to show that $ab < c^2/4$. We can write

$$
\begin{aligned}
ab &= \frac{a^2 + 2ab + b^2 - a^2 + 2ab - b^2}{4} \\
&= \frac{(a+b)^2 - (a-b)^2}{4} \leq \frac{(a+b)^2}{4} < \frac{c^2}{4}.
\end{aligned}
$$
∎

The ***natural logarithm*** function $\ln x = \log_e x$, where $e = 2.71828\ldots$, is the value of the following progression:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots.$$

In addition,

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

$$\ln(1+x) = x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^4}{4!} + \cdots.$$

There are a number of useful inequalities relating to these functions (which derive from these definitions).

Proposition A.2: *If $x > -1$*

$$\frac{x}{1+x} \le \ln(1+x) \le x.$$

Proposition A.3: *For $0 \le x < 1$*

$$1 + x \le e^x \le \frac{1}{1-x}.$$

Proposition A.4: *For any two positive real numbers x and n*

$$\left(1 + \frac{x}{n}\right)^n \le e^x \le \left(1 + \frac{x}{n}\right)^{n+x/2}.$$

Integer Functions and Relations

The "floor" and "ceiling" functions are defined respectively as follows:

1. $\lfloor x \rfloor$ = the largest integer less than or equal to x
2. $\lceil x \rceil$ = the smallest integer greater than or equal to x.

The ***modulo*** operator is defined for integers $a \ge 0$ and $b > 0$ as

$$a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b.$$

The ***factorial*** function is defined as

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1)n.$$

The binomial coefficient is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

which is equal to the number of different ***combinations*** one can define by choosing k different items from a collection of n items (where the order does not matter). The name "binomial coefficient" derives from the ***binomial expansion***

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}.$$

We also have the following relationships.

Proposition A.5: If $0 \le k \le n$, then

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \frac{n^k}{k!}.$$

Proposition A.6: *Stirlings Approximation*

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \varepsilon(n)\right),$$

where $\varepsilon(n)$ is $O(1/n^2)$.

The **Fibonacci progression** is a numeric progression such that $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \ge 2$.

Proposition A.7: *If F_n is defined by the Fibonacci progression, then F_n is $\Theta(g^n)$, where $g = (1 + \sqrt{5})/2$ is the so-called **golden ratio**.*

Summations

There are a number of useful facts about summations.

Proposition A.8: *Factoring summations*

$$\sum_{i=1}^{n} af(i) = a \sum_{i=1}^{n} f(i),$$

provided a does not depend upon i.

Proposition A.9: *Reversing the order*

$$\sum_{i=1}^{n} \sum_{j=1}^{m} f(i, j) = \sum_{j=1}^{m} \sum_{i=1}^{n} f(i, j).$$

One special form of summation is a **telescoping sum**

$$\sum_{i=1}^{n} (f(i) - f(i-1)) = f(n) - f(0),$$

which arises often in the amortized analysis of a data structure or algorithm.

The following are some other facts about summations that arise often in the analysis of data structures and algorithms.

Proposition A.10: $\sum_{i=1}^{n} i = n(n+1)/2$.

Proposition A.11: $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$.

Proposition A.12: *If $k \geq 1$ is an integer constant, then*

$$\sum_{i=1}^{n} i^k \text{ is } \Theta(n^{k+1}).$$

Another common summation is the ***geometric sum***, $\sum_{i=0}^{n} a^i$, for any fixed real number $0 < a \neq 1$.
Proposition A.13:

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1},$$

for any real number $0 < a \neq 1$.
Proposition A.14:

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1 - a}$$

for any real number $0 < a < 1$.

There is also a combination of the two common forms, called the ***linear exponential*** summation, which has the following expansions
Proposition A.15: *For $0 < a \neq 1$, and $n \geq 2$*

$$\sum_{i=1}^{n} ia^i = \frac{a - (n+1)a^{(n+1)} + na^{(n+2)}}{(1 - a)^2}.$$

The nth ***Harmonic number*** H_n is defined as

$$H_n = \sum_{i=1}^{n} \frac{1}{i}.$$

Proposition A.16: *If H_n is the nth harmonic number, then H_n is $\ln n + \Theta(1)$.*

Basic Probability

We review some basic facts from probability theory. The most basic such fact is that any statement about a probability is defined upon a ***sample space*** S, which is defined as the set of all possible outcomes from some experiment. We leave the terms "outcomes" and "experiment" undefined in any formal sense.

Example A.17: *Consider an experiment that consists of the outcome from flipping a coin 5 times. This sample space has 2^5 different outcomes, one for each different ordering of possible flips that can occur.*

Sample spaces can also be infinite, as the following example illustrates.

Example A.18: *Consider an experiment that consists of flipping a coin until it comes up heads. This sample space is infinite, with each outcome being a sequence of i tails followed by a single flip that comes up heads, for $i = 1, 2, 3, \ldots$.*

A **probability space** is a sample space S together with a probability function Pr that maps subsets of S to real numbers in the interval $[0, 1]$. It mathematically captures the notion of the probability of certain "events" occurring. Formally, each subset A of S is called an **event**, and the probability function Pr is assumed to possess the following basic properties with respect to events defined from S:

1. $\Pr(\emptyset) = 0$
2. $\Pr(S) = 1$
3. $0 \le \Pr(A) \le 1$, for any $A \subseteq S$
4. If $A, B \subseteq S$ and $A \cap B = \emptyset$, then $\Pr(A \cup B) = \Pr(A) + \Pr(B)$

Two events A and B are **independent** if

$$\Pr(A \cap B) = \Pr(A) \cdot \Pr(B).$$

A collection of events $\{A_1, A_2, \ldots, A_n\}$ is **mutually independent** if

$$\Pr(A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}) = \Pr(A_{i_1}) \Pr(A_{i_2}) \cdots \Pr(A_{i_k}).$$

for any subset $\{A_{i_1}, A_{i_2}, \ldots, A_{i_k}\}$.

The **conditional probability** that an event A occurs, given an event B is denoted as $\Pr(A|B)$, and is defined as the ratio

$$\frac{\Pr(A \cap B)}{\Pr(B)},$$

assuming that $\Pr(B) > 0$.

An elegant way of dealing with events is in terms of **random variables**. Intuitively, random variables are variables whose values depend upon the outcome of some experiment. Formally, a **random variable** is a function X that maps outcomes from some sample space S to real numbers. An **indicator random variable** is a random variable that maps outcomes to the set $\{0, 1\}$. Often in data structure and algorithm analysis we use a random variable X to characterize the running time of a randomized algorithm. In this case the sample space S is defined by all possible outcomes of the random sources used in the algorithm.

In such cases we are most interested in the typical, average, or "expected" value of such a random variable. The **expected value** of a random variable X is defined as

$$\mathbf{E}(X) = \sum_x x \Pr(X = x),$$

where the summation is defined over the range of X (which in this case is assumed to be discrete).

Proposition A.19 (The Linearity of Expectation): *Let X and Y be two random variables and let c be a number. Then*

$$E(X+Y) = E(X) + E(Y) \qquad and \qquad E(cX) = cE(X).$$

Example A.20: *Let X be a random variable that assigns the outcome of the roll of two fair dice to the sum of the number of dots showing. Then $\mathbf{E}(X) = 7$.*

Justification: *To justify this claim let X_1 and X_2 be random variables corresponding to the number of dots on each die. Thus, $X_1 = X_2$ (that is, they are two instances of the same function) and $E(X) = \mathbf{E}(X_1 + X_2) = \mathbf{E}(X_1) + \mathbf{E}(X_2)$. Each outcome of the roll of a fair die occurs with probability $1/6$. Thus,*

$$\mathbf{E}(X_i) = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = \frac{7}{2},$$

for $i = 1, 2$. Therefore, $E(X) = 7$.   ■

Two random variables X and Y are ***independent*** if

$$\Pr(X = x | Y = y) = \Pr(X = x),$$

for all real numbers x and y.

Proposition A.21: *If two random variables X and Y are independent, then*

$$\mathbf{E}(XY) = \mathbf{E}(X)\mathbf{E}(Y).$$

Example A.22: *Let X be a random variable that assigns the outcome of a roll of two fair dice to the product of the number of dots showing. Then $\mathbf{E}(X) = 49/4$.*

Justification: *Let X_1 and X_2 be random variables denoting the number of dots on each die. The variables X_1 and X_2 are clearly independent; hence,*

$$\mathbf{E}(X) = \mathbf{E}(X_1 X_2) = \mathbf{E}(X_1)\mathbf{E}(X_2) = (7/2)^2 = 49/4.$$ ■

The following bound and corollaries that follow from it are known as ***Chernoff bounds***.

Proposition A.23: *Let X be the sum of a finite number of independent 0/1 random variables and let $\mu > 0$ be the expected value of X. Then, for $\delta > 0$*

$$\Pr(X > (1+\delta)\mu) < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \right]^{\mu}.$$

Useful Mathematical Techniques

To compare the growth rates of different functions, it is sometimes helpful to apply the following rule.

Proposition A.24 (L'Hôpital's Rule): *If we have* $\lim_{n\to\infty} f(n) = +\infty$ *and we have* $\lim_{n\to\infty} g(n) = +\infty$, *then* $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} f'(n)/g'(n)$, *where* $f'(n)$ *and* $g'(n)$ *denote the derivatives of* $f(n)$ *and* $g(n)$, *respectively.*

In deriving an upper or lower bound for a summation, it is often useful to **split a summation** as follows

$$\sum_{i=1}^{n} f(i) = \sum_{i=1}^{j} f(i) + \sum_{i=j+1}^{n} f(i).$$

Another useful technique is to **bound a sum by an integral**. If f is a non-decreasing function, then, assuming the following terms are defined

$$\int_{a-1}^{b} f(x)\,dx \le \sum_{i=a}^{b} f(i) \le \int_{a}^{b+1} f(x)\,dx.$$

There is a general form of recurrence relation that arises in the analysis of divide-and-conquer algorithms

$$T(n) = aT(n/b) + f(n),$$

for constants $a \ge 1$ and $b > 1$.

Proposition A.25: *Let $T(n)$ be defined as above. Then:*

1. *If $f(n)$ is $O(n^{\log_b a - \varepsilon})$, for some constant $\varepsilon > 0$, then $T(n)$ is $\Theta(n^{\log_b a})$*
2. *If $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, for a fixed nonnegative integer $k \ge 0$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$*
3. *If $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, for some constant $\varepsilon > 0$, and if $af(n/b) \le cf(n)$, then $T(n)$ is $\Theta(f(n))$*

This proposition is known as the **master method** for characterizing divide-and-conquer recurrence relations asymptotically.

This page intentionally left blank

Bibliography

[1] G. M. Adel'son-Vel'skii and Y. M. Landis, "An algorithm for the organization of information," *Doklady Akademii Nauk SSSR*, vol. 146, pp. 263–266, 1962. English translation in *Soviet Math. Dokl.*, **3**, 1259–1262.

[2] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, pp. 1116–1127, 1988.

[3] A. V. Aho, "Algorithms for finding patterns in strings," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 255–300, Amsterdam: Elsevier, 1990.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.

[6] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[7] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Reading, Mass.: Addison-Wesley, 1999.

[8] O. Baruvka, "O jistem problemu minimalnim," *Praca Moravske Prirodovedecke Spolecnosti*, vol. 3, pp. 37–58, 1926. (in Czech).

[9] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance," *Acta Informatica*, vol. 1, no. 4, pp. 290–306, 1972.

[10] R. Bayer and McCreight, "Organization of large ordered indexes," *Acta Inform.*, vol. 1, pp. 173–189, 1972.

[11] J. L. Bentley, "Programming pearls: Writing correct programs," *Communications of the ACM*, vol. 26, pp. 1040–1045, 1983.

[12] J. L. Bentley, "Programming pearls: Thanks, heaps," *Communications of the ACM*, vol. 28, pp. 245–250, 1985.

[13] G. Booch, *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummings, 1994.

[14] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[15] G. Brassard, "Crusade for a better notation," *SIGACT News*, vol. 17, no. 1, pp. 60–64, 1985.

[16] T. Budd, *An Introduction to Object-Oriented Programming*. Reading, Mass.: Addison-Wesley, 1991.

[17] T. Budd, *C++ for Java Programmers*. Reading, Mass.: Addison-Wesley, 1999.

[18] D. Burger, J. R. Goodman, and G. S. Sohi, "Memory systems," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 18, pp. 447–461, CRC Press, 1997.

[19] L. Cardelli and P. Wegner, "On understanding types, data abstraction and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–522, 1985.

[20] S. Carlsson, "Average case results on heapsort," *BIT*, vol. 27, pp. 2–17, 1987.

[21] K. L. Clarkson, "Linear programming in $O(n3^{d^2})$ time," *Inform. Process. Lett.*, vol. 22, pp. 21–24, 1986.

[22] R. Cole, "Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm," *SIAM Journal on Computing*, vol. 23, no. 5, pp. 1075–1091, 1994.

[23] D. Comer, "The ubiquitous B-tree," *ACM Comput. Surv.*, vol. 11, pp. 121–137, 1979.

[24] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2nd ed., 2001.

[26] M. Crochemore and T. Lecroq, "Pattern matching and text compression algorithms," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 8, pp. 162–202, CRC Press, 1997.

[27] S. A. Demurjian, Sr., "Software design," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 108, pp. 2323–2351, CRC Press, 1997.

[28] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing*. Upper Saddle River, NJ: Prentice Hall, 1999.

[29] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[30] J. R. Driscoll, H. N. Gabow, R. Shrairaman, and R. E. Tarjan, "Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation.," *Commun. ACM*, vol. 31, pp. 1343–1354, 1988.

[31] S. Even, *Graph Algorithms*. Potomac, Maryland: Computer Science Press, 1979.

[32] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[33] R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, vol. 7, no. 12, p. 701, 1964.

[34] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, pp. 596–615, 1987.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.

[36] A. M. Gibbons, *Algorithmic Graph Theory*. Cambridge, UK: Cambridge University Press, 1985.

[37] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures in Pascal and C*. Reading, Mass.: Addison-Wesley, 1991.

[38] G. H. Gonnet and J. I. Munro, "Heaps on heaps," *SIAM Journal on Computing*, vol. 15, no. 4, pp. 964–971, 1986.

[39] M. T. Goodrich, M. Handy, B. Hudson, and R. Tamassia, "Accessing the internal organization of data structures in the JDSL library," in *Proc. Workshop on Algorithm Engineering and Experimentation* (M. T. Goodrich and C. C. McGeoch, eds.), vol. 1619 of *Lecture Notes Comput. Sci.*, pp. 124–139, Springer-Verlag, 1999.

[40] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-memory computational geometry," in *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci.*, pp. 714–723, 1993.

[41] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *Annals of the History of Computing*, vol. 7, no. 1, pp. 43–57, 1985.

[42] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci.*, Lecture Notes Comput. Sci., pp. 8–21, Springer-Verlag, 1978.

[43] Y. Gurevich, "What does $O(n)$ mean?," *SIGACT News*, vol. 17, no. 4, pp. 61–63, 1986.

[44] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann, 2nd ed., 1996.

[45] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, pp. 10–15, 1962.

[46] J. E. Hopcroft and R. E. Tarjan, "Efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, 1973.

[47] C. S. Horstmann, *Computing Concepts with C++ Essentials*. Ney York: John Wiley and Sons, 2nd ed., 1998.

[48] B. Huang and M. Langston, "Practical in-place merging," *Communications of the ACM*, vol. 31, no. 3, pp. 348–352, 1988.

[49] J. JáJá, *An Introduction to Parallel Algorithms*. Reading, Mass.: Addison-Wesley, 1992.

[50] V. Jarnik, "O jistem problemu minimalnim," *Praca Moravske Prirodovedecke Spolecnosti*, vol. 6, pp. 57–63, 1930. (in Czech).

[51] R. E. Jones, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[52] D. R. Karger, P. Klein, and R. E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *Journal of the ACM*, vol. 42, pp. 321–328, 1995.

[53] R. M. Karp and V. Ramachandran, "Parallel algorithms for shared memory machines," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), pp. 869–941, Amsterdam: Elsevier/The MIT Press, 1990.

[54] P. Kirschenhofer and H. Prodinger, "The path length of random skip lists," *Acta Informatica*, vol. 31, pp. 775–792, 1994.

[55] J. Kleinberg and E. Tardos, *Algorithm Design*. Reading, MA: Addison-Wesley, 2006.

[56] D. E. Knuth, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 2nd ed., 1973.

[57] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 1973.

[58] D. E. Knuth, "Big omicron and big omega and big theta," in *SIGACT News*, vol. 8, pp. 18–24, 1976.

[59] D. E. Knuth, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 3rd ed., 1997.

[60] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 2nd ed., 1998.

[61] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 323–350, 1977.

[62] J. B. Kruskal, Jr., "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. Amer. Math. Soc.*, vol. 7, pp. 48–50, 1956.

[63] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *IEEE Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[64] R. Levisse, "Some lessons drawn from the history of the binary search algorithm," *The Computer Journal*, vol. 26, pp. 154–163, 1983.

[65] A. Levitin, "Do we teach the right algorithm design techniques?," in *30th ACM SIGCSE Symp. on Computer Science Education*, pp. 179–183, 1999.

[66] S. Lippmann, *Essential C++*. Reading, Mass.: Addison-Wesley, 2000.

[67] S. Lippmann and J. Lajoie, *C++ Primer*. Reading, Mass.: Addison-Wesley, 3rd ed., 1998.

[68] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*. Cambridge, Mass./New York: The MIT Press/McGraw-Hill, 1986.

[69] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of Algorithms*, vol. 23, no. 2, pp. 262–272, 1976.

[70] C. J. H. McDiarmid and B. A. Reed, "Building heaps fast," *Journal of Algorithms*, vol. 10, no. 3, pp. 352–365, 1989.

[71] N. Megiddo, "Linear-time algorithms for linear programming in R^3 and related problems," *SIAM J. Comput.*, vol. 12, pp. 759–776, 1983.

[72] N. Megiddo, "Linear programming in linear time when the dimension is fixed," *J. ACM*, vol. 31, pp. 114–127, 1984.

[73] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, vol. 1 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.

[74] K. Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, vol. 2 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.

[75] K. Mehlhorn and A. Tsakalidis, "Data structures," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 301–341, Amsterdam: Elsevier, 1990.

[76] S. Meyers, *More Effective C++*. Reading, Mass.: Addison-Wesley, 1996.

[77] S. Meyers, *Effective C++*. Reading, Mass.: Addison-Wesley, 2nd ed., 1998.

[78] M. H. Morgan, *Vitruvius: The Ten Books on Architecture*. New York: Dover Publications, Inc., 1960.

[79] D. R. Morrison, "PATRICIA—practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514–534, 1968.

[80] R. Motwani and P. Raghavan, *Randomized Algorithms*. New York, NY: Cambridge University Press, 1995.

[81] D. R. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Reading, Mass.: Addison-Wesley, 1996.

[82] T. Papadakis, J. I. Munro, and P. V. Poblete, "Average search and update costs in skip lists," *BIT*, vol. 32, pp. 316–332, 1992.

[83] P. V. Poblete, J. I. Munro, and T. Papadakis, "The binomial transform and its application to the analysis of skip lists," in *Proceedings of the European Symposium on Algorithms (ESA)*, pp. 554–569, 1995.

[84] I. Pohl, *C++ For C Programmers*. Reading, Mass.: Addison-Wesley, 3rd ed., 1999.

[85] R. C. Prim, "Shortest connection networks and some generalizations," *Bell Syst. Tech. J.*, vol. 36, pp. 1389–1401, 1957.

[86] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[87] H. Samet, *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.

[88] R. Schaffer and R. Sedgewick, "The analysis of heapsort," *Journal of Algorithms*, vol. 15, no. 1, pp. 76–100, 1993.

[89] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652–686, 1985.

[90] G. A. Stephen, *String Searching Algorithms*. World Scientific Press, 1994.

[91] B. Stroustrup, *The C++ Programming Language*. Reading, Mass.: Addison-Wesley, 3rd ed., 1997.

[92] R. Tamassia and G. Liotta, "Graph drawing," in *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), CRC Press, second ed., 2004.

[93] R. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. Comput.*, vol. 14, pp. 862–874, 1985.

[94] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[95] R. E. Tarjan, *Data Structures and Network Algorithms*, vol. 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1983.

[96] A. B. Tucker, Jr., *The Computer Science and Engineering Handbook*. CRC Press, 1997.

[97] J. D. Ullman, *Principles of Database Systems*. Potomac, MD: Computer Science Press, 1983.

[98] J. van Leeuwen, "Graph algorithms," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 525–632, Amsterdam: Elsevier, 1990.

[99] J. S. Vitter, "Efficient memory access in large-scale computation," in *Proc. 8th Sympos. Theoret. Aspects Comput. Sci.*, Lecture Notes Comput. Sci., Springer-Verlag, 1991.

[100] J. S. Vitter and W. C. Chen, *Design and Analysis of Coalesced Hashing*. New York: Oxford University Press, 1987.

[101] J. S. Vitter and P. Flajolet, "Average-case analysis of algorithms and data structures," in *Algorithms and Complexity* (J. van Leeuwen, ed.), vol. A of *Handbook of Theoretical Computer Science*, pp. 431–524, Amsterdam: Elsevier, 1990.

[102] S. Warshall, "A theorem on boolean matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11–12, 1962.

[103] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.

[104] D. Wood, *Data Structures, Algorithms, and Performance*. Reading, Mass.: Addison-Wesley, 1993.

Index