A Practical Introduction to Data Structures and Algorithm Analysis

Edition 3.2 (C++ Version)

Clifford A. Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

March 28, 2013
Update 3.2.0.10
For a list of changes, see
http://people.cs.vt.edu/~shaffer/Book/errata.html

If you wish to have a printed version of this document, print copies are published by Dover Publications
(see http://store.doverpublications.com/048648582x.html).
Further information about this text is available at
http://people.cs.vt.edu/~shaffer/Book/.

Contents

Preface

We study data structures so that we can learn to write more efficient programs. But why must programs be efficient when new computers are faster every year? The reason is that our ambitions grow with our capabilities. Instead of rendering efficiency needs obsolete, the modern revolution in computing power and storage capability merely raises the efficiency stakes as we attempt more complex tasks.

The quest for program efficiency need not and should not conflict with sound design and clear coding. Creating efficient programs has little to do with "programming tricks" but rather is based on good organization of information and good algorithms. A programmer who has not mastered the basic principles of clear design is not likely to write efficient programs. Conversely, concerns related to development costs and maintainability should not be used as an excuse to justify inefficient performance. Generality in design can and should be achieved without sacrificing performance, but this can only be done if the designer understands how to measure performance and does so as an integral part of the design and implementation process. Most computer science curricula recognize that good programming skills begin with a strong emphasis on fundamental software engineering principles. Then, once a programmer has learned the principles of clear program design and implementation, the next step is to study the effects of data organization and algorithms on program efficiency.

Approach: This book describes many techniques for representing data. These techniques are presented within the context of the following principles:

1. Each data structure and each algorithm has costs and benefits. Practitioners need a thorough understanding of how to assess costs and benefits to be able to adapt to new design challenges. This requires an understanding of the principles of algorithm analysis, and also an appreciation for the significant effects of the physical medium employed (e.g., data stored on disk versus main memory).
2. Related to costs and benefits is the notion of tradeoffs. For example, it is quite common to reduce time requirements at the expense of an increase in space requirements, or vice versa. Programmers face tradeoff issues regularly in all

phases of software design and implementation, so the concept must become deeply ingrained.

3. Programmers should know enough about common practice to avoid reinventing the wheel. Thus, programmers need to learn the commonly used data structures, their related algorithms, and the most frequently encountered design patterns found in programming.
4. Data structures follow needs. Programmers must learn to assess application needs first, then find a data structure with matching capabilities. To do this requires competence in Principles 1, 2, and 3.

As I have taught data structures through the years, I have found that design issues have played an ever greater role in my courses. This can be traced through the various editions of this textbook by the increasing coverage for design patterns and generic interfaces. The first edition had no mention of design patterns. The second edition had limited coverage of a few example patterns, and introduced the dictionary ADT and comparator classes. With the third edition, there is explicit coverage of some design patterns that are encountered when programming the basic data structures and algorithms covered in the book.

Using the Book in Class: Data structures and algorithms textbooks tend to fall into one of two categories: teaching texts or encyclopedias. Books that attempt to do both usually fail at both. This book is intended as a teaching text. I believe it is more important for a practitioner to understand the principles required to select or design the data structure that will best solve some problem than it is to memorize a lot of textbook implementations. Hence, I have designed this as a teaching text that covers most standard data structures, but not all. A few data structures that are not widely adopted are included to illustrate important principles. Some relatively new data structures that should become widely used in the future are included.

Within an undergraduate program, this textbook is designed for use in either an advanced lower division (sophomore or junior level) data structures course, or for a senior level algorithms course. New material has been added in the third edition to support its use in an algorithms course. Normally, this text would be used in a course beyond the standard freshman level "CS2" course that often serves as the initial introduction to data structures. Readers of this book should typically have two semesters of the equivalent of programming experience, including at least some exposure to **C**++. Readers who are already familiar with recursion will have an advantage. Students of data structures will also benefit from having first completed a good course in Discrete Mathematics. Nonetheless, Chapter 2 attempts to give a reasonably complete survey of the prerequisite mathematical topics at the level necessary to understand their use in this book. Readers may wish to refer back to the appropriate sections as needed when encountering unfamiliar mathematical material.

A sophomore-level class where students have only a little background in basic data structures or analysis (that is, background equivalent to what would be had from a traditional CS2 course) might cover Chapters 1-11 in detail, as well as selected topics from Chapter 13. That is how I use the book for my own sophomore-level class. Students with greater background might cover Chapter 1, skip most of Chapter 2 except for reference, briefly cover Chapters 3 and 4, and then cover chapters 5-12 in detail. Again, only certain topics from Chapter 13 might be covered, depending on the programming assignments selected by the instructor. A senior-level algorithms course would focus on Chapters 11 and 14-17.

Chapter 13 is intended in part as a source for larger programming exercises. I recommend that all students taking a data structures course be required to implement some advanced tree structure, or another dynamic structure of comparable difficulty such as the skip list or sparse matrix representations of Chapter 12. None of these data structures are significantly more difficult to implement than the binary search tree, and any of them should be within a student's ability after completing Chapter 5.

While I have attempted to arrange the presentation in an order that makes sense, instructors should feel free to rearrange the topics as they see fit. The book has been written so that once the reader has mastered Chapters 1-6, the remaining material has relatively few dependencies. Clearly, external sorting depends on understanding internal sorting and disk files. Section 6.2 on the UNION/FIND algorithm is used in Kruskal's Minimum-Cost Spanning Tree algorithm. Section 9.2 on self-organizing lists mentions the buffer replacement schemes covered in Section 8.3. Chapter 14 draws on examples from throughout the book. Section 17.2 relies on knowledge of graphs. Otherwise, most topics depend only on material presented earlier within the same chapter.

Most chapters end with a section entitled "Further Reading." These sections are not comprehensive lists of references on the topics presented. Rather, I include books and articles that, in my opinion, may prove exceptionally informative or entertaining to the reader. In some cases I include references to works that should become familiar to any well-rounded computer scientist.

Use of C++: The programming examples are written in **C++**, but I do not wish to discourage those unfamiliar with **C++** from reading this book. I have attempted to make the examples as clear as possible while maintaining the advantages of **C++**. **C++** is used here strictly as a tool to illustrate data structures concepts. In particular, I make use of **C++**'s support for hiding implementation details, including features such as classes, private class members, constructors, and destructors. These features of the language support the crucial concept of separating logical design, as embodied in the abstract data type, from physical implementation as embodied in the data structure.

To keep the presentation as clear as possible, some important features of **C**++ are avoided here. I deliberately minimize use of certain features commonly used by experienced **C**++ programmers such as class hierarchy, inheritance, and virtual functions. Operator and function overloading is used sparingly. **C**-like initialization syntax is preferred to some of the alternatives offered by **C**++.

While the **C**++ features mentioned above have valid design rationale in real programs, they tend to obscure rather than enlighten the principles espoused in this book. For example, inheritance is an important tool that helps programmers avoid duplication, and thus minimize bugs. From a pedagogical standpoint, however, inheritance often makes code examples harder to understand since it tends to spread the description for one logical unit among several classes. Thus, my class definitions only use inheritance where inheritance is explicitly relevant to the point illustrated (e.g., Section 5.3.1). This does not mean that a programmer should do likewise. Avoiding code duplication and minimizing errors are important goals. Treat the programming examples as illustrations of data structure principles, but do not copy them directly into your own programs.

One painful decision I had to make was whether to use templates in the code examples. In the first edition of this book, the decision was to leave templates out as it was felt that their syntax obscures the meaning of the code for those not familiar with **C**++. In the years following, the use of **C**++ in computer science curricula has greatly expanded. I now assume that readers of the text will be familiar with template syntax. Thus, templates are now used extensively in the code examples.

My implementations are meant to provide concrete illustrations of data structure principles, as an aid to the textual exposition. Code examples should not be read or used in isolation from the associated text because the bulk of each example's documentation is contained in the text, not the code. The code complements the text, not the other way around. They are not meant to be a series of commercial-quality class implementations. If you are looking for a complete implementation of a standard data structure for use in your own code, you would do well to do an Internet search.

For instance, the code examples provide less parameter checking than is sound programming practice, since including such checking would obscure rather than illuminate the text. Some parameter checking and testing for other constraints (e.g., whether a value is being removed from an empty container) is included in the form of a call to **`Assert`**. The inputs to **`Assert`** are a Boolean expression and a character string. If this expression evaluates to **`false`**, then a message is printed and the program terminates immediately. Terminating a program when a function receives a bad parameter is generally considered undesirable in real programs, but is quite adequate for understanding how a data structure is meant to operate. In real programming applications, **C**++'s exception handling features should be used to deal with input data errors. However, assertions provide a simpler mechanism for indi-

cating required conditions in a way that is both adequate for clarifying how a data structure is meant to operate, and is easily modified into true exception handling. See the Appendix for the implementation of **`Assert`**.

I make a distinction in the text between "**C**++ implementations" and "pseudocode." Code labeled as a **C**++ implementation has actually been compiled and tested on one or more **C**++ compilers. Pseudocode examples often conform closely to **C**++ syntax, but typically contain one or more lines of higher-level description. Pseudocode is used where I perceived a greater pedagogical advantage to a simpler, but less precise, description.

Exercises and Projects: Proper implementation and analysis of data structures cannot be learned simply by reading a book. You must practice by implementing real programs, constantly comparing different techniques to see what really works best in a given situation.

One of the most important aspects of a course in data structures is that it is where students really learn to program using pointers and dynamic memory allocation, by implementing data structures such as linked lists and trees. It is often where students truly learn recursion. In our curriculum, this is the first course where students do significant design, because it often requires real data structures to motivate significant design exercises. Finally, the fundamental differences between memory-based and disk-based data access cannot be appreciated without practical programming experience. For all of these reasons, a data structures course cannot succeed without a significant programming component. In our department, the data structures course is one of the most difficult programming course in the curriculum.

Students should also work problems to develop their analytical abilities. I provide over 450 exercises and suggestions for programming projects. I urge readers to take advantage of them.

Contacting the Author and Supplementary Materials: A book such as this is sure to contain errors and have room for improvement. I welcome bug reports and constructive criticism. I can be reached by electronic mail via the Internet at **`shaffer@vt.edu`**. Alternatively, comments can be mailed to

Cliff Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

The electronic posting of this book, along with a set of lecture notes for use in class can be obtained at

`http://www.cs.vt.edu/~shaffer/book.html.`

The code examples used in the book are available at the same site. Online Web pages for Virginia Tech's sophomore-level data structures class can be found at

http://courses.cs.vt.edu/~cs3114.

Readers of this textbook will be interested in our open-source, online eTextbook project, OpenDSA (**http://algoviz.org/OpenDSA**). The OpenDSA project's goal is to ceate a complete collection of tutorials that combine textbook-quality content with algorithm visualizations for every algorithm and data structure, and a rich collection of interactive exercises. When complete, OpenDSA will replace this book.

This book was typeset by the author using LaTeX. The bibliography was prepared using BibTeX. The index was prepared using **makeindex**. The figures were mostly drawn with **Xfig**. Figures 3.1 and 9.10 were partially created using Mathematica.

Acknowledgments: It takes a lot of help from a lot of people to make a book. I wish to acknowledge a few of those who helped to make this book possible. I apologize for the inevitable omissions.

Virginia Tech helped make this whole thing possible through sabbatical research leave during Fall 1994, enabling me to get the project off the ground. My department heads during the time I have written the various editions of this book, Dennis Kafura and Jack Carroll, provided unwavering moral support for this project. Mike Keenan, Lenny Heath, and Jeff Shaffer provided valuable input on early versions of the chapters. I also wish to thank Lenny Heath for many years of stimulating discussions about algorithms and analysis (and how to teach both to students). Steve Edwards deserves special thanks for spending so much time helping me on various redesigns of the **C**++ and Java code versions for the second and third editions, and many hours of discussion on the principles of program design. Thanks to Layne Watson for his help with Mathematica, and to Bo Begole, Philip Isenhour, Jeff Nielsen, and Craig Struble for much technical assistance. Thanks to Bill McQuain, Mark Abrams and Dennis Kafura for answering lots of silly questions about **C**++ and Java.

I am truly indebted to the many reviewers of the various editions of this manuscript. For the first edition these reviewers included J. David Bezek (University of Evansville), Douglas Campbell (Brigham Young University), Karen Davis (University of Cincinnati), Vijay Kumar Garg (University of Texas – Austin), Jim Miller (University of Kansas), Bruce Maxim (University of Michigan – Dearborn), Jeff Parker (Agile Networks/Harvard), Dana Richards (George Mason University), Jack Tan (University of Houston), and Lixin Tao (Concordia University). Without their help, this book would contain many more technical errors and many fewer insights.

For the second edition, I wish to thank these reviewers: Gurdip Singh (Kansas State University), Peter Allen (Columbia University), Robin Hill (University of Wyoming), Norman Jacobson (University of California – Irvine), Ben Keller (Eastern Michigan University), and Ken Bosworth (Idaho State University). In addition,

I wish to thank Neil Stewart and Frank J. Thesen for their comments and ideas for improvement.

Third edition reviewers included Randall Lechlitner (University of Houstin, Clear Lake) and Brian C. Hipp (York Technical College). I thank them for their comments.

Prentice Hall was the original print publisher for the first and second editions. Without the hard work of many people there, none of this would be possible. Authors simply do not create printer-ready books on their own. Foremost thanks go to Kate Hargett, Petra Rector, Laura Steele, and Alan Apt, my editors over the years. My production editors, Irwin Zucker for the second edition, Kathleen Caren for the original **C**++ version, and Ed DeFelippis for the Java version, kept everything moving smoothly during that horrible rush at the end. Thanks to Bill Zobrist and Bruce Gregory (I think) for getting me into this in the first place. Others at Prentice Hall who helped me along the way include Truly Donovan, Linda Behrens, and Phyllis Bregman. Thanks to Tracy Dunkelberger for her help in returning the copyright to me, thus enabling the electronic future of this work. I am sure I owe thanks to many others at Prentice Hall for their help in ways that I am not even aware of.

I am thankful to Shelley Kronzek at Dover publications for her faith in taking on the print publication of this third edition. Much expanded, with both Java and **C**++ versions, and many inconsistencies corrected, I am confident that this is the best edition yet. But none of us really knows whether students will prefer a free online textbook or a low-cost, printed bound version. In the end, we believe that the two formats will be mutually supporting by offering more choices. Production editor James Miller and design manager Marie Zaczkiewicz have worked hard to ensure that the production is of the highest quality.

I wish to express my appreciation to Hanan Samet for teaching me about data structures. I learned much of the philosophy presented here from him as well, though he is not responsible for any problems with the result. Thanks to my wife Terry, for her love and support, and to my daughters Irena and Kate for pleasant diversions from working too hard. Finally, and most importantly, to all of the data structures students over the years who have taught me what is important and what should be skipped in a data structures course, and the many new insights they have provided. This book is dedicated to them.

Cliff Shaffer
Blacksburg, Virginia

PART I

Preliminaries

1

Data Structures and Algorithms

How many cities with more than 250,000 people lie within 500 miles of Dallas, Texas? How many people in my company make over $100,000 per year? Can we connect all of our telephone customers with less than 1,000 miles of cable? To answer questions like these, it is not enough to have the necessary information. We must organize that information in a way that allows us to find the answers in time to satisfy our needs.

Representing information is fundamental to computer science. The primary purpose of most computer programs is not to perform calculations, but to store and retrieve information — usually as fast as possible. For this reason, the study of data structures and the algorithms that manipulate them is at the heart of computer science. And that is what this book is about — helping you to understand how to structure information to support efficient processing.

This book has three primary goals. The first is to present the commonly used data structures. These form a programmer's basic data structure "toolkit." For many problems, some data structure in the toolkit provides a good solution.

The second goal is to introduce the idea of tradeoffs and reinforce the concept that there are costs and benefits associated with every data structure. This is done by describing, for each data structure, the amount of space and time required for typical operations.

The third goal is to teach how to measure the effectiveness of a data structure or algorithm. Only through such measurement can you determine which data structure in your toolkit is most appropriate for a new problem. The techniques presented also allow you to judge the merits of new data structures that you or others might invent.

There are often many approaches to solving a problem. How do we choose between them? At the heart of computer program design are two (sometimes conflicting) goals:

1. To design an algorithm that is easy to understand, code, and debug.
2. To design an algorithm that makes efficient use of the computer's resources.

Ideally, the resulting program is true to both of these goals. We might say that such a program is "elegant." While the algorithms and program code examples presented here attempt to be elegant in this sense, it is not the purpose of this book to explicitly treat issues related to goal (1). These are primarily concerns of the discipline of Software Engineering. Rather, this book is mostly about issues relating to goal (2).

How do we measure efficiency? Chapter 3 describes a method for evaluating the efficiency of an algorithm or computer program, called **asymptotic analysis**. Asymptotic analysis also allows you to measure the inherent difficulty of a problem. The remaining chapters use asymptotic analysis techniques to estimate the time cost for every algorithm presented. This allows you to see how each algorithm compares to other algorithms for solving the same problem in terms of its efficiency.

This first chapter sets the stage for what is to follow, by presenting some higher-order issues related to the selection and use of data structures. We first examine the process by which a designer selects a data structure appropriate to the task at hand. We then consider the role of abstraction in program design. We briefly consider the concept of a design pattern and see some examples. The chapter ends with an exploration of the relationship between problems, algorithms, and programs.

1.1 A Philosophy of Data Structures

1.1.1 The Need for Data Structures

You might think that with ever more powerful computers, program efficiency is becoming less important. After all, processor speed and memory size still continue to improve. Won't any efficiency problem we might have today be solved by tomorrow's hardware?

As we develop more powerful computers, our history so far has always been to use that additional computing power to tackle more complex problems, be it in the form of more sophisticated user interfaces, bigger problem sizes, or new problems previously deemed computationally infeasible. More complex problems demand more computation, making the need for efficient programs even greater. Worse yet, as tasks become more complex, they become less like our everyday experience. Today's computer scientists must be trained to have a thorough understanding of the principles behind efficient program design, because their ordinary life experiences often do not apply when designing computer programs.

In the most general sense, a data structure is any data representation and its associated operations. Even an integer or floating point number stored on the computer can be viewed as a simple data structure. More commonly, people use the term "data structure" to mean an organization or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring.

Given sufficient space to store a collection of data items, it is always possible to search for specified items within the collection, print or otherwise process the data items in any desired order, or modify the value of any particular data item. Thus, it is possible to perform all necessary operations on any data structure. However, using the proper data structure can make the difference between a program running in a few seconds and one requiring many days.

A solution is said to be **efficient** if it solves the problem within the required **resource constraints**. Examples of resource constraints include the total space available to store the data — possibly divided into separate main memory and disk space constraints — and the time allowed to perform each subtask. A solution is sometimes said to be efficient if it requires fewer resources than known alternatives, regardless of whether it meets any particular requirements. The **cost** of a solution is the amount of resources that the solution consumes. Most often, cost is measured in terms of one key resource such as time, with the implied assumption that the solution meets the other resource constraints.

It should go without saying that people write programs to solve problems. However, it is crucial to keep this truism in mind when selecting a data structure to solve a particular problem. Only by first analyzing the problem to determine the performance goals that must be achieved can there be any hope of selecting the right data structure for the job. Poor program designers ignore this analysis step and apply a data structure that they are familiar with but which is inappropriate to the problem. The result is typically a slow program. Conversely, there is no sense in adopting a complex representation to "improve" a program that can meet its performance goals when implemented using a simpler design.

When selecting a data structure to solve a problem, you should follow these steps.

1. Analyze your problem to determine the basic operations that must be supported. Examples of basic operations include inserting a data item into the data structure, deleting a data item from the data structure, and finding a specified data item.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

This three-step approach to selecting a data structure operationalizes a data-centered view of the design process. The first concern is for the data and the operations to be performed on them, the next concern is the representation for those data, and the final concern is the implementation of that representation.

Resource constraints on certain key operations, such as search, inserting data records, and deleting data records, normally drive the data structure selection process. Many issues relating to the relative importance of these operations are addressed by the following three questions, which you should ask yourself whenever you must choose a data structure:

- Are all data items inserted into the data structure at the beginning, or are insertions interspersed with other operations? Static applications (where the data are loaded at the beginning and never change) typically require only simpler data structures to get an efficient implementation than do dynamic applications.
- Can data items be deleted? If so, this will probably make the implementation more complicated.
- Are all data items processed in some well-defined order, or is search for specific data items allowed? "Random access" search generally requires more complex data structures.

1.1.2 Costs and Benefits

Each data structure has associated costs and benefits. In practice, it is hardly ever true that one data structure is better than another for use in all situations. If one data structure or algorithm is superior to another in all respects, the inferior one will usually have long been forgotten. For nearly every data structure and algorithm presented in this book, you will see examples of where it is the best choice. Some of the examples might surprise you.

A data structure requires a certain amount of space for each data item it stores, a certain amount of time to perform a single basic operation, and a certain amount of programming effort. Each problem has constraints on available space and time. Each solution to a problem makes use of the basic operations in some relative proportion, and the data structure selection process must account for this. Only after a careful analysis of your problem's characteristics can you determine the best data structure for the task.

Example 1.1 A bank must support many types of transactions with its customers, but we will examine a simple model where customers wish to open accounts, close accounts, and add money or withdraw money from accounts. We can consider this problem at two distinct levels: (1) the requirements for the physical infrastructure and workflow process that the bank uses in its interactions with its customers, and (2) the requirements for the database system that manages the accounts.

The typical customer opens and closes accounts far less often than he or she accesses the account. Customers are willing to wait many minutes while accounts are created or deleted but are typically not willing to wait more than a brief time for individual account transactions such as a deposit or withdrawal. These observations can be considered as informal specifications for the time constraints on the problem.

It is common practice for banks to provide two tiers of service. Human tellers or automated teller machines (ATMs) support customer access

to account balances and updates such as deposits and withdrawals. Special service representatives are typically provided (during restricted hours) to handle opening and closing accounts. Teller and ATM transactions are expected to take little time. Opening or closing an account can take much longer (perhaps up to an hour from the customer's perspective).

From a database perspective, we see that ATM transactions do not modify the database significantly. For simplicity, assume that if money is added or removed, this transaction simply changes the value stored in an account record. Adding a new account to the database is allowed to take several minutes. Deleting an account need have no time constraint, because from the customer's point of view all that matters is that all the money be returned (equivalent to a withdrawal). From the bank's point of view, the account record might be removed from the database system after business hours, or at the end of the monthly account cycle.

When considering the choice of data structure to use in the database system that manages customer accounts, we see that a data structure that has little concern for the cost of deletion, but is highly efficient for search and moderately efficient for insertion, should meet the resource constraints imposed by this problem. Records are accessible by unique account number (sometimes called an **exact-match query**). One data structure that meets these requirements is the hash table described in Chapter 9.4. Hash tables allow for extremely fast exact-match search. A record can be modified quickly when the modification does not affect its space requirements. Hash tables also support efficient insertion of new records. While deletions can also be supported efficiently, too many deletions lead to some degradation in performance for the remaining operations. However, the hash table can be reorganized periodically to restore the system to peak efficiency. Such reorganization can occur offline so as not to affect ATM transactions.

Example 1.2 A company is developing a database system containing information about cities and towns in the United States. There are many thousands of cities and towns, and the database program should allow users to find information about a particular place by name (another example of an exact-match query). Users should also be able to find all places that match a particular value or range of values for attributes such as location or population size. This is known as a **range query**.

A reasonable database system must answer queries quickly enough to satisfy the patience of a typical user. For an exact-match query, a few seconds is satisfactory. If the database is meant to support range queries that can return many cities that match the query specification, the entire opera-

tion may be allowed to take longer, perhaps on the order of a minute. To meet this requirement, it will be necessary to support operations that process range queries efficiently by processing all cities in the range as a batch, rather than as a series of operations on individual cities.

The hash table suggested in the previous example is inappropriate for implementing our city database, because it cannot perform efficient range queries. The B^+-tree of Section 10.5.1 supports large databases, insertion and deletion of data records, and range queries. However, a simple linear index as described in Section 10.1 would be more appropriate if the database is created once, and then never changed, such as an atlas distributed on a CD or accessed from a website.

1.2 Abstract Data Types and Data Structures

The previous section used the terms "data item" and "data structure" without properly defining them. This section presents terminology and motivates the design process embodied in the three-step approach to selecting a data structure. This motivation stems from the need to manage the tremendous complexity of computer programs.

A **type** is a collection of values. For example, the Boolean type consists of the values `true` and `false`. The integers also form a type. An integer is a **simple type** because its values contain no subparts. A bank account record will typically contain several pieces of information such as name, address, account number, and account balance. Such a record is an example of an **aggregate type** or **composite type**. A **data item** is a piece of information or a record whose value is drawn from a type. A data item is said to be a **member** of a type.

A **data type** is a type together with a collection of operations to manipulate the type. For example, an integer variable is a member of the integer data type. Addition is an example of an operation on the integer data type.

A distinction should be made between the logical concept of a data type and its physical implementation in a computer program. For example, there are two traditional implementations for the list data type: the linked list and the array-based list. The list data type can therefore be implemented using a linked list or an array. Even the term "array" is ambiguous in that it can refer either to a data type or an implementation. "Array" is commonly used in computer programming to mean a contiguous block of memory locations, where each memory location stores one fixed-length data item. By this meaning, an array is a physical data structure. However, array can also mean a logical data type composed of a (typically homogeneous) collection of data items, with each data item identified by an index number. It is possible to implement arrays in many different ways. For exam-

ple, Section 12.2 describes the data structure used to implement a sparse matrix, a large two-dimensional array that stores only a relatively few non-zero values. This implementation is quite different from the physical representation of an array as contiguous memory locations.

An **abstract data type** (ADT) is the realization of a data type as a software component. The interface of the ADT is defined in terms of a type and a set of operations on that type. The behavior of each operation is determined by its inputs and outputs. An ADT does not specify *how* the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as **encapsulation**.

A **data structure** is the implementation for an ADT. In an object-oriented language such as **C**++, an ADT and its implementation together make up a **class**. Each operation associated with the ADT is implemented by a **member function** or **method**. The variables that define the space required by a data item are referred to as **data members**. An **object** is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.

The term "data structure" often refers to data stored in a computer's main memory. The related term **file structure** often refers to the organization of data on peripheral storage, such as a disk drive or CD.

Example 1.3 The mathematical concept of an integer, along with operations that manipulate integers, form a data type. The **C**++ `int` variable type is a physical representation of the abstract integer. The `int` variable type, along with the operations that act on an `int` variable, form an ADT. Unfortunately, the `int` implementation is not completely true to the abstract integer, as there are limitations on the range of values an `int` variable can store. If these limitations prove unacceptable, then some other representation for the ADT "integer" must be devised, and a new implementation must be used for the associated operations.

Example 1.4 An ADT for a list of integers might specify the following operations:

- Insert a new integer at a particular position in the list.
- Return `true` if the list is empty.
- Reinitialize the list.
- Return the number of integers currently in the list.
- Delete the integer at a particular position in the list.

From this description, the input and output of each operation should be clear, but the implementation for lists has not been specified.

One application that makes use of some ADT might use particular member functions of that ADT more than a second application, or the two applications might have different time requirements for the various operations. These differences in the requirements of applications are the reason why a given ADT might be supported by more than one implementation.

Example 1.5 Two popular implementations for large disk-based database applications are hashing (Section 9.4) and the B^+-tree (Section 10.5). Both support efficient insertion and deletion of records, and both support exact-match queries. However, hashing is more efficient than the B^+-tree for exact-match queries. On the other hand, the B^+-tree can perform range queries efficiently, while hashing is hopelessly inefficient for range queries. Thus, if the database application limits searches to exact-match queries, hashing is preferred. On the other hand, if the application requires support for range queries, the B^+-tree is preferred. Despite these performance issues, both implementations solve versions of the same problem: updating and searching a large collection of records.

The concept of an ADT can help us to focus on key issues even in non-computing applications.

Example 1.6 When operating a car, the primary activities are steering, accelerating, and braking. On nearly all passenger cars, you steer by turning the steering wheel, accelerate by pushing the gas pedal, and brake by pushing the brake pedal. This design for cars can be viewed as an ADT with operations "steer," "accelerate," and "brake." Two cars might implement these operations in radically different ways, say with different types of engine, or front- versus rear-wheel drive. Yet, most drivers can operate many different cars because the ADT presents a uniform method of operation that does not require the driver to understand the specifics of any particular engine or drive design. These differences are deliberately hidden.

The concept of an ADT is one instance of an important principle that must be understood by any successful computer scientist: managing complexity through abstraction. A central theme of computer science is complexity and techniques for handling it. Humans deal with complexity by assigning a label to an assembly of objects or concepts and then manipulating the label in place of the assembly. Cognitive psychologists call such a label a **metaphor**. A particular label might be related to other pieces of information or other labels. This collection can in turn be given a label, forming a hierarchy of concepts and labels. This hierarchy of labels allows us to focus on important issues while ignoring unnecessary details.

Example 1.7 We apply the label "hard drive" to a collection of hardware that manipulates data on a particular type of storage device, and we apply the label "CPU" to the hardware that controls execution of computer instructions. These and other labels are gathered together under the label "computer." Because even the smallest home computers today have millions of components, some form of abstraction is necessary to comprehend how a computer operates.

Consider how you might go about the process of designing a complex computer program that implements and manipulates an ADT. The ADT is implemented in one part of the program by a particular data structure. While designing those parts of the program that use the ADT, you can think in terms of operations on the data type without concern for the data structure's implementation. Without this ability to simplify your thinking about a complex program, you would have no hope of understanding or implementing it.

Example 1.8 Consider the design for a relatively simple database system stored on disk. Typically, records on disk in such a program are accessed through a buffer pool (see Section 8.3) rather than directly. Variable length records might use a memory manager (see Section 12.3) to find an appropriate location within the disk file to place the record. Multiple index structures (see Chapter 10) will typically be used to access records in various ways. Thus, we have a chain of classes, each with its own responsibilities and access privileges. A database query from a user is implemented by searching an index structure. This index requests access to the record by means of a request to the buffer pool. If a record is being inserted or deleted, such a request goes through the memory manager, which in turn interacts with the buffer pool to gain access to the disk file. A program such as this is far too complex for nearly any human programmer to keep all of the details in his or her head at once. The only way to design and implement such a program is through proper use of abstraction and metaphors. In object-oriented programming, such abstraction is handled using classes.

Data types have both a **logical** and a **physical** form. The definition of the data type in terms of an ADT is its logical form. The implementation of the data type as a data structure is its physical form. Figure 1.1 illustrates this relationship between logical and physical forms for data types. When you implement an ADT, you are dealing with the physical form of the associated data type. When you use an ADT elsewhere in your program, you are concerned with the associated data type's logical form. Some sections of this book focus on physical implementations for a

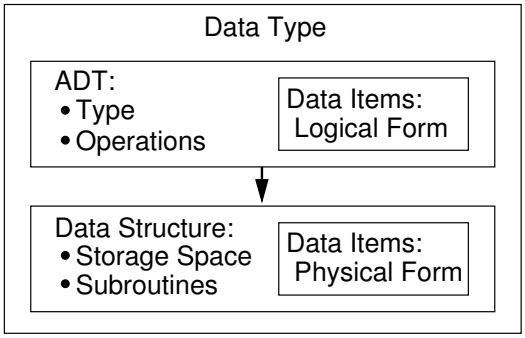


Figure 1.1 The relationship between data items, abstract data types, and data structures. The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

given data structure. Other sections use the logical ADT for the data structure in the context of a higher-level task.

Example 1.9 A particular C++ environment might provide a library that includes a list class. The logical form of the list is defined by the public functions, their inputs, and their outputs that define the class. This might be all that you know about the list class implementation, and this should be all you need to know. Within the class, a variety of physical implementations for lists is possible. Several are described in Section 4.1.

1.3 Design Patterns

At a higher level of abstraction than ADTs are abstractions for describing the design of programs — that is, the interactions of objects and classes. Experienced software designers learn and reuse patterns for combining software components. These have come to be referred to as **design patterns**.

A design pattern embodies and generalizes important design concepts for a recurring problem. A primary goal of design patterns is to quickly transfer the knowledge gained by expert designers to newer programmers. Another goal is to allow for efficient communication between programmers. It is much easier to discuss a design issue when you share a technical vocabulary relevant to the topic.

Specific design patterns emerge from the realization that a particular design problem appears repeatedly in many contexts. They are meant to solve real problems. Design patterns are a bit like templates. They describe the structure for a design solution, with the details filled in for any given problem. Design patterns are a bit like data structures: Each one provides costs and benefits, which implies

that tradeoffs are possible. Therefore, a given design pattern might have variations on its application to match the various tradeoffs inherent in a given situation.

The rest of this section introduces a few simple design patterns that are used later in the book.

1.3.1 Flyweight

The Flyweight design pattern is meant to solve the following problem. You have an application with many objects. Some of these objects are identical in the information that they contain, and the role that they play. But they must be reached from various places, and conceptually they really are distinct objects. Because there is so much duplication of the same information, we would like to take advantage of the opportunity to reduce memory cost by sharing that space. An example comes from representing the layout for a document. The letter "C" might reasonably be represented by an object that describes that character's strokes and bounding box. However, we do not want to create a separate "C" object everywhere in the document that a "C" appears. The solution is to allocate a single copy of the shared representation for "C" objects. Then, every place in the document that needs a "C" in a given font, size, and typeface will reference this single copy. The various instances of references to a specific form of "C" are called flyweights.

We could describe the layout of text on a page by using a tree structure. The root of the tree represents the entire page. The page has multiple child nodes, one for each column. The column nodes have child nodes for each row. And the rows have child nodes for each character. These representations for characters are the flyweights. The flyweight includes the reference to the shared shape information, and might contain additional information specific to that instance. For example, each instance for "C" will contain a reference to the shared information about strokes and shapes, and it might also contain the exact location for that instance of the character on the page.

Flyweights are used in the implementation for the PR quadtree data structure for storing collections of point objects, described in Section 13.3. In a PR quadtree, we again have a tree with leaf nodes. Many of these leaf nodes represent empty areas, and so the only information that they store is the fact that they are empty. These identical nodes can be implemented using a reference to a single instance of the flyweight for better memory efficiency.

1.3.2 Visitor

Given a tree of objects to describe a page layout, we might wish to perform some activity on every node in the tree. Section 5.2 discusses tree traversal, which is the process of visiting every node in the tree in a defined order. A simple example for our text composition application might be to count the number of nodes in the tree

that represents the page. At another time, we might wish to print a listing of all the nodes for debugging purposes.

We could write a separate traversal function for each such activity that we intend to perform on the tree. A better approach would be to write a generic traversal function, and pass in the activity to be performed at each node. This organization constitutes the visitor design pattern. The visitor design pattern is used in Sections 5.2 (tree traversal) and 11.3 (graph traversal).

1.3.3 Composite

There are two fundamental approaches to dealing with the relationship between a collection of actions and a hierarchy of object types. First consider the typical procedural approach. Say we have a base class for page layout entities, with a subclass hierarchy to define specific subtypes (page, columns, rows, figures, characters, etc.). And say there are actions to be performed on a collection of such objects (such as rendering the objects to the screen). The procedural design approach is for each action to be implemented as a method that takes as a parameter a pointer to the base class type. Each action such method will traverse through the collection of objects, visiting each object in turn. Each action method contains something like a switch statement that defines the details of the action for each subclass in the collection (e.g., page, column, row, character). We can cut the code down some by using the visitor design pattern so that we only need to write the traversal once, and then write a visitor subroutine for each action that might be applied to the collection of objects. But each such visitor subroutine must still contain logic for dealing with each of the possible subclasses.

In our page composition application, there are only a few activities that we would like to perform on the page representation. We might render the objects in full detail. Or we might want a "rough draft" rendering that prints only the bounding boxes of the objects. If we come up with a new activity to apply to the collection of objects, we do not need to change any of the code that implements the existing activities. But adding new activities won't happen often for this application. In contrast, there could be many object types, and we might frequently add new object types to our implementation. Unfortunately, adding a new object type requires that we modify each activity, and the subroutines implementing the activities get rather long switch statements to distinguish the behavior of the many subclasses.

An alternative design is to have each object subclass in the hierarchy embody the action for each of the various activities that might be performed. Each subclass will have code to perform each activity (such as full rendering or bounding box rendering). Then, if we wish to apply the activity to the collection, we simply call the first object in the collection and specify the action (as a method call on that object). In the case of our page layout and its hierarchical collection of objects, those objects that contain other objects (such as a row objects that contains letters)

will call the appropriate method for each child. If we want to add a new activity with this organization, we have to change the code for every subclass. But this is relatively rare for our text compositing application. In contrast, adding a new object into the subclass hierarchy (which for this application is far more likely than adding a new rendering function) is easy. Adding a new subclass does not require changing any of the existing subclasses. It merely requires that we define the behavior of each activity that can be performed on the new subclass.

This second design approach of burying the functional activity in the subclasses is called the Composite design pattern. A detailed example for using the Composite design pattern is presented in Section 5.3.1.

1.3.4 Strategy

Our final example of a design pattern lets us encapsulate and make interchangeable a set of alternative actions that might be performed as part of some larger activity. Again continuing our text compositing example, each output device that we wish to render to will require its own function for doing the actual rendering. That is, the objects will be broken down into constituent pixels or strokes, but the actual mechanics of rendering a pixel or stroke will depend on the output device. We don't want to build this rendering functionality into the object subclasses. Instead, we want to pass to the subroutine performing the rendering action a method or class that does the appropriate rendering details for that output device. That is, we wish to hand to the object the appropriate "strategy" for accomplishing the details of the rendering task. Thus, this approach is called the Strategy design pattern.

The Strategy design pattern will be discussed further in Chapter 7. There, a sorting function is given a class (called a comparator) that understands how to extract and compare the key values for records to be sorted. In this way, the sorting function does not need to know any details of how its record type is implemented.

One of the biggest challenges to understanding design patterns is that sometimes one is only subtly different from another. For example, you might be confused about the difference between the composite pattern and the visitor pattern. The distinction is that the composite design pattern is about whether to give control of the traversal process to the nodes of the tree or to the tree itself. Both approaches can make use of the visitor design pattern to avoid rewriting the traversal function many times, by encapsulating the activity performed at each node.

But isn't the strategy design pattern doing the same thing? The difference between the visitor pattern and the strategy pattern is more subtle. Here the difference is primarily one of intent and focus. In both the strategy design pattern and the visitor design pattern, an activity is being passed in as a parameter. The strategy design pattern is focused on encapsulating an activity that is part of a larger process, so that different ways of performing that activity can be substituted. The visitor design pattern is focused on encapsulating an activity that will be performed on all

members of a collection so that completely different activities can be substituted within a generic method that accesses all of the collection members.

1.4 Problems, Algorithms, and Programs

Programmers commonly deal with problems, algorithms, and computer programs. These are three distinct concepts.

Problems: As your intuition would suggest, a **problem** is a task to be performed. It is best thought of in terms of inputs and matching outputs. A problem definition should not include any constraints on *how* the problem is to be solved. The solution method should be developed only after the problem is precisely defined and thoroughly understood. However, a problem definition should include constraints on the resources that may be consumed by any acceptable solution. For any problem to be solved by a computer, there are always such constraints, whether stated or implied. For example, any computer program may use only the main memory and disk space available, and it must run in a "reasonable" amount of time.

Problems can be viewed as functions in the mathematical sense. A **function** is a matching between inputs (the **domain**) and outputs (the **range**). An input to a function might be a single value or a collection of information. The values making up an input are called the **parameters** of the function. A specific selection of values for the parameters is called an **instance** of the problem. For example, the input parameter to a sorting function might be an array of integers. A particular array of integers, with a given size and specific values for each position in the array, would be an instance of the sorting problem. Different instances might generate the same output. However, any problem instance must always result in the same output every time the function is computed using that particular input.

This concept of all problems behaving like mathematical functions might not match your intuition for the behavior of computer programs. You might know of programs to which you can give the same input value on two separate occasions, and two different outputs will result. For example, if you type "`date`" to a typical UNIX command line prompt, you will get the current date. Naturally the date will be different on different days, even though the same command is given. However, there is obviously more to the input for the date program than the command that you type to run the program. The date program computes a function. In other words, on any particular day there can only be a single answer returned by a properly running date program on a completely specified input. For all computer programs, the output is completely determined by the program's full set of inputs. Even a "random number generator" is completely determined by its inputs (although some random number generating systems appear to get around this by accepting a random input from a physical process beyond the user's control). The relationship between programs and functions is explored further in Section 17.3.

Algorithms: An **algorithm** is a method or a process followed to solve a problem. If the problem is viewed as a function, then an algorithm is an implementation for the function that transforms an input to the corresponding output. A problem can be solved by many different algorithms. A given algorithm solves only one problem (i.e., computes a particular function). This book covers many problems, and for several of these problems I present more than one algorithm. For the important problem of sorting I present nearly a dozen algorithms!

The advantage of knowing several solutions to a problem is that solution A might be more efficient than solution B for a specific variation of the problem, or for a specific class of inputs to the problem, while solution B might be more efficient than A for another variation or class of inputs. For example, one sorting algorithm might be the best for sorting a small collection of integers (which is important if you need to do this many times). Another might be the best for sorting a large collection of integers. A third might be the best for sorting a collection of variable-length strings.

By definition, something can only be called an algorithm if it has all of the following properties.

1. It must be *correct*. In other words, it must compute the desired function, converting each input to the correct output. Note that every algorithm implements some function, because every algorithm maps every input to some output (even if that output is a program crash). At issue here is whether a given algorithm implements the *intended* function.
2. It is composed of a series of *concrete steps*. Concrete means that the action described by that step is completely understood — and doable — by the person or machine that must perform the algorithm. Each step must also be doable in a finite amount of time. Thus, the algorithm gives us a "recipe" for solving the problem by performing a series of steps, where each such step is within our capacity to perform. The ability to perform a step can depend on who or what is intended to execute the recipe. For example, the steps of a cookie recipe in a cookbook might be considered sufficiently concrete for instructing a human cook, but not for programming an automated cookie-making factory.
3. There can be *no ambiguity* as to which step will be performed next. Often it is the next step of the algorithm description. Selection (e.g., the `if` statement in C++) is normally a part of any language for describing algorithms. Selection allows a choice for which step will be performed next, but the selection process is unambiguous at the time when the choice is made.
4. It must be composed of a *finite* number of steps. If the description for the algorithm were made up of an infinite number of steps, we could never hope to write it down, nor implement it as a computer program. Most languages for describing algorithms (including English and "pseudocode") provide some

way to perform repeated actions, known as iteration. Examples of iteration in programming languages include the **`while`** and **`for`** loop constructs of C++. Iteration allows for short descriptions, with the number of steps actually performed controlled by the input.

5. It must *terminate*. In other words, it may not go into an infinite loop.

Programs: We often think of a **computer program** as an instance, or concrete representation, of an algorithm in some programming language. In this book, nearly all of the algorithms are presented in terms of programs, or parts of programs. Naturally, there are many programs that are instances of the same algorithm, because any modern computer programming language can be used to implement the same collection of algorithms (although some programming languages can make life easier for the programmer). To simplify presentation, I often use the terms "algorithm" and "program" interchangeably, despite the fact that they are really separate concepts. By definition, an algorithm must provide sufficient detail that it can be converted into a program when needed.

The requirement that an algorithm must terminate means that not all computer programs meet the technical definition of an algorithm. Your operating system is one such program. However, you can think of the various tasks for an operating system (each with associated inputs and outputs) as individual problems, each solved by specific algorithms implemented by a part of the operating system program, and each one of which terminates once its output is produced.

To summarize: A **problem** is a function or a mapping of inputs to outputs. An **algorithm** is a recipe for solving a problem whose steps are concrete and unambiguous. Algorithms must be correct, of finite length, and must terminate for all inputs. A **program** is an instantiation of an algorithm in a programming language.

1.5 Further Reading

An early authoritative work on data structures and algorithms was the series of books *The Art of Computer Programming* by Donald E. Knuth, with Volumes 1 and 3 being most relevant to the study of data structures [Knu97, Knu98]. A modern encyclopedic approach to data structures and algorithms that should be easy to understand once you have mastered this book is *Algorithms* by Robert Sedgewick [Sed11]. For an excellent and highly readable (but more advanced) teaching introduction to algorithms, their design, and their analysis, see *Introduction to Algorithms: A Creative Approach* by Udi Manber [Man89]. For an advanced, encyclopedic approach, see *Introduction to Algorithms* by Cormen, Leiserson, and Rivest [CLRS09]. Steven S. Skiena's *The Algorithm Design Manual* [Ski10] provides pointers to many implementations for data structures and algorithms that are available on the Web.

The claim that all modern programming languages can implement the same algorithms (stated more precisely, any function that is computable by one programming language is computable by any programming language with certain standard capabilities) is a key result from computability theory. For an easy introduction to this field see James L. Hein, *Discrete Structures, Logic, and Computability* [Hei09].

Much of computer science is devoted to problem solving. Indeed, this is what attracts many people to the field. *How to Solve It* by George Pólya [Pól57] is considered to be the classic work on how to improve your problem-solving abilities. If you want to be a better student (as well as a better problem solver in general), see *Strategies for Creative Problem Solving* by Folger and LeBlanc [FL95], *Effective Problem Solving* by Marvin Levine [Lev94], and *Problem Solving & Comprehension* by Arthur Whimbey and Jack Lochhead [WL99], and *Puzzle-Based Learning* by Zbigniew and Matthew Michaelewicz [MM08].

See *The Origin of Consciousness in the Breakdown of the Bicameral Mind* by Julian Jaynes [Jay90] for a good discussion on how humans use the concept of metaphor to handle complexity. More directly related to computer science education and programming, see "Cogito, Ergo Sum! Cognitive Processes of Students Dealing with Data Structures" by Dan Aharoni [Aha00] for a discussion on moving from programming-context thinking to higher-level (and more design-oriented) programming-free thinking.

On a more pragmatic level, most people study data structures to write better programs. If you expect your program to work correctly and efficiently, it must first be understandable to yourself and your co-workers. Kernighan and Pike's *The Practice of Programming* [KP99] discusses a number of practical issues related to programming, including good coding and documentation style. For an excellent (and entertaining!) introduction to the difficulties involved with writing large programs, read the classic *The Mythical Man-Month: Essays on Software Engineering* by Frederick P. Brooks [Bro95].

If you want to be a successful **C**++ programmer, you need good reference manuals close at hand. The standard reference for **C**++ is *The C++ Programming Language* by Bjarne Stroustrup [Str00], with further information provided in *The Annotated C++ Reference Manual* by Ellis and Stroustrup [ES90]. No **C**++ programmer should be without Stroustrup's book, as it provides the definitive description of the language and also includes a great deal of information about the principles of object-oriented design. Unfortunately, it is a poor text for learning how to program in **C**++. A good, gentle introduction to the basics of the language is Patrick Henry Winston's *On to C++* [Win94]. A good introductory teaching text for a wider range of **C**++ is Deitel and Deitel's *C++ How to Program* [DD08].

After gaining proficiency in the mechanics of program writing, the next step is to become proficient in program design. Good design is difficult to learn in any discipline, and good design for object-oriented software is one of the most difficult

of arts. The novice designer can jump-start the learning process by studying well-known and well-used design patterns. The classic reference on design patterns is *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides [GHJV95] (this is commonly referred to as the "gang of four" book). Unfortunately, this is an extremely difficult book to understand, in part because the concepts are inherently difficult. A number of Web sites are available that discuss design patterns, and which provide study guides for the *Design Patterns* book. Two other books that discuss object-oriented software design are *Object-Oriented Software Design and Construction with* **C++** by Dennis Kafura [Kaf98], and *Object-Oriented Design Heuristics* by Arthur J. Riel [Rie96].

1.6 Exercises

The exercises for this chapter are different from those in the rest of the book. Most of these exercises are answered in the following chapters. However, you should *not* look up the answers in other parts of the book. These exercises are intended to make you think about some of the issues to be covered later on. Answer them to the best of your ability with your current knowledge.

1.1 Think of a program you have used that is unacceptably slow. Identify the specific operations that make the program slow. Identify other basic operations that the program performs quickly enough.

1.2 Most programming languages have a built-in integer data type. Normally this representation has a fixed size, thus placing a limit on how large a value can be stored in an integer variable. Describe a representation for integers that has no size restriction (other than the limits of the computer's available main memory), and thus no practical limit on how large an integer can be stored. Briefly show how your representation can be used to implement the operations of addition, multiplication, and exponentiation.

1.3 Define an ADT for character strings. Your ADT should consist of typical functions that can be performed on strings, with each function defined in terms of its input and output. Then define two different physical representations for strings.

1.4 Define an ADT for a list of integers. First, decide what functionality your ADT should provide. Example 1.4 should give you some ideas. Then, specify your ADT in **C++** in the form of an abstract class declaration, showing the functions, their parameters, and their return types.

1.5 Briefly describe how integer variables are typically represented on a computer. (Look up one's complement and two's complement arithmetic in an introductory computer science textbook if you are not familiar with these.)

Why does this representation for integers qualify as a data structure as defined in Section 1.2?

1.6 Define an ADT for a two-dimensional array of integers. Specify precisely the basic operations that can be performed on such arrays. Next, imagine an application that stores an array with 1000 rows and 1000 columns, where less than 10,000 of the array values are non-zero. Describe two different implementations for such arrays that would be more space efficient than a standard two-dimensional array implementation requiring one million positions.

1.7 Imagine that you have been assigned to implement a sorting program. The goal is to make this program general purpose, in that you don't want to define in advance what record or key types are used. Describe ways to generalize a simple sorting algorithm (such as insertion sort, or any other sort you are familiar with) to support this generalization.

1.8 Imagine that you have been assigned to implement a simple sequential search on an array. The problem is that you want the search to be as general as possible. This means that you need to support arbitrary record and key types. Describe ways to generalize the search function to support this goal. Consider the possibility that the function will be used multiple times in the same program, on differing record types. Consider the possibility that the function will need to be used on different keys (possibly with the same or different types) of the same record. For example, a student data record might be searched by zip code, by name, by salary, or by GPA.

1.9 Does every problem have an algorithm?

1.10 Does every algorithm have a **C**++ program?

1.11 Consider the design for a spelling checker program meant to run on a home computer. The spelling checker should be able to handle quickly a document of less than twenty pages. Assume that the spelling checker comes with a dictionary of about 20,000 words. What primitive operations must be implemented on the dictionary, and what is a reasonable time constraint for each operation?

1.12 Imagine that you have been hired to design a database service containing information about cities and towns in the United States, as described in Example 1.2. Suggest two possible implementations for the database.

1.13 Imagine that you are given an array of records that is sorted with respect to some key field contained in each record. Give two different algorithms for searching the array to find the record with a specified key value. Which one do you consider "better" and why?

1.14 How would you go about comparing two proposed algorithms for sorting an array of integers? In particular,

(a) What would be appropriate measures of cost to use as a basis for comparing the two sorting algorithms?

(b) What tests or analysis would you conduct to determine how the two algorithms perform under these cost measures?

1.15 A common problem for compilers and text editors is to determine if the parentheses (or other brackets) in a string are balanced and properly nested. For example, the string "((())())()" contains properly nested pairs of parentheses, but the string ")()(" does not; and the string "())" does not contain properly matching parentheses.

(a) Give an algorithm that returns **`true`** if a string contains properly nested and balanced parentheses, and **`false`** if otherwise. *Hint*: At no time while scanning a legal string from left to right will you have encountered more right parentheses than left parentheses.

(b) Give an algorithm that returns the position in the string of the first offending parenthesis if the string is not properly nested and balanced. That is, if an excess right parenthesis is found, return its position; if there are too many left parentheses, return the position of the first excess left parenthesis. Return -1 if the string is properly balanced and nested.

1.16 A graph consists of a set of objects (called vertices) and a set of edges, where each edge connects two vertices. Any given pair of vertices can be connected by only one edge. Describe at least two different ways to represent the connections defined by the vertices and edges of a graph.

1.17 Imagine that you are a shipping clerk for a large company. You have just been handed about 1000 invoices, each of which is a single sheet of paper with a large number in the upper right corner. The invoices must be sorted by this number, in order from lowest to highest. Write down as many different approaches to sorting the invoices as you can think of.

1.18 How would you sort an array of about 1000 integers from lowest value to highest value? Write down at least five approaches to sorting the array. Do not write algorithms in **C**++ or pseudocode. Just write a sentence or two for each approach to describe how it would work.

1.19 Think of an algorithm to find the maximum value in an (unsorted) array. Now, think of an algorithm to find the second largest value in the array. Which is harder to implement? Which takes more time to run (as measured by the number of comparisons performed)? Now, think of an algorithm to find the third largest value. Finally, think of an algorithm to find the middle value. Which is the most difficult of these problems to solve?

1.20 An unsorted list allows for constant-time insert by adding a new element at the end of the list. Unfortunately, searching for the element with key value X requires a sequential search through the unsorted list until X is found, which on average requires looking at half the list element. On the other hand, a

sorted array-based list of n elements can be searched in $\log n$ time with a binary search. Unfortunately, inserting a new element requires a lot of time because many elements might be shifted in the array if we want to keep it sorted. How might data be organized to support both insertion and search in $\log n$ time?

2

Mathematical Preliminaries

This chapter presents mathematical notation, background, and techniques used throughout the book. This material is provided primarily for review and reference. You might wish to return to the relevant sections when you encounter unfamiliar notation or mathematical techniques in later chapters.

Section 2.7 on estimation might be unfamiliar to many readers. Estimation is not a mathematical technique, but rather a general engineering skill. It is enormously useful to computer scientists doing design work, because any proposed solution whose estimated resource requirements fall well outside the problem's resource constraints can be discarded immediately, allowing time for greater analysis of more promising solutions.

2.1 Sets and Relations

The concept of a set in the mathematical sense has wide application in computer science. The notations and techniques of set theory are commonly used when describing and implementing algorithms because the abstractions associated with sets often help to clarify and simplify algorithm design.

A **set** is a collection of distinguishable **members** or **elements**. The members are typically drawn from some larger population known as the **base type**. Each member of a set is either a **primitive element** of the base type or is a set itself. There is no concept of duplication in a set. Each value from the base type is either in the set or not in the set. For example, a set named **P** might consist of the three integers 7, 11, and 42. In this case, **P**'s members are 7, 11, and 42, and the base type is integer.

Figure 2.1 shows the symbols commonly used to express sets and their relationships. Here are some examples of this notation in use. First define two sets, **P** and **Q**.

$$\mathbf{P} = \{2, 3, 5\}, \qquad \mathbf{Q} = \{5, 10\}.$$

$\{1, 4\}$	A set composed of the members 1 and 4
$\{x \mid x$ is a positive integer$\}$	A set definition using a **set former** Example: the set of all positive integers
$x \in \mathbf{P}$	x is a member of set **P**
$x \notin \mathbf{P}$	x is not a member of set **P**
$\emptyset$	The null or empty set
$\lvert\mathbf{P}\rvert$	Cardinality: size of set **P** or number of members for set **P**
$\mathbf{P} \subseteq \mathbf{Q}$, $\mathbf{Q} \supseteq \mathbf{P}$	Set **P** is included in set **Q**, set **P** is a subset of set **Q**, set **Q** is a superset of set **P**
$\mathbf{P} \cup \mathbf{Q}$	Set Union: all elements appearing in **P** OR **Q**
$\mathbf{P} \cap \mathbf{Q}$	Set Intersection: all elements appearing in **P** AND **Q**
$\mathbf{P} - \mathbf{Q}$	Set difference: all elements of set **P** NOT in set **Q**

Figure 2.1 Set notation.

$|\mathbf{P}| = 3$ (because **P** has three members) and $|\mathbf{Q}| = 2$ (because **Q** has two members). The union of **P** and **Q**, written $\mathbf{P} \cup \mathbf{Q}$, is the set of elements in either **P** or **Q**, which is $\{2, 3, 5, 10\}$. The intersection of **P** and **Q**, written $\mathbf{P} \cap \mathbf{Q}$, is the set of elements that appear in both **P** and **Q**, which is $\{5\}$. The set difference of **P** and **Q**, written $\mathbf{P} - \mathbf{Q}$, is the set of elements that occur in **P** but not in **Q**, which is $\{2, 3\}$. Note that $\mathbf{P} \cup \mathbf{Q} = \mathbf{Q} \cup \mathbf{P}$ and that $\mathbf{P} \cap \mathbf{Q} = \mathbf{Q} \cap \mathbf{P}$, but in general $\mathbf{P} - \mathbf{Q} \neq \mathbf{Q} - \mathbf{P}$. In this example, $\mathbf{Q} - \mathbf{P} = \{10\}$. Note that the set $\{4, 3, 5\}$ is indistinguishable from set **P**, because sets have no concept of order. Likewise, set $\{4, 3, 4, 5\}$ is also indistinguishable from **P**, because sets have no concept of duplicate elements.

The **powerset** of a set **S** is the set of all possible subsets for **S**. Consider the set $\mathbf{S} = \{a, b, c\}$. The powerset of **S** is

$$\{\emptyset,\ \{a\},\ \{b\},\ \{c\},\ \{a, b\},\ \{a, c\},\ \{b, c\},\ \{a, b, c\}\}.$$

A collection of elements with no order (like a set), but with duplicate-valued elements is called a **bag**.[1] To distinguish bags from sets, I use square brackets [] around a bag's elements. For example, bag [3, 4, 5, 4] is distinct from bag [3, 4, 5], while set $\{3, 4, 5, 4\}$ is indistinguishable from set $\{3, 4, 5\}$. However, bag [3, 4, 5, 4] is indistinguishable from bag [3, 4, 4, 5].

[1]The object referred to here as a bag is sometimes called a **multilist**. But, I reserve the term multilist for a list that may contain sublists (see Section 12.1).

A **sequence** is a collection of elements with an order, and which may contain duplicate-valued elements. A sequence is also sometimes called a **tuple** or a **vector**. In a sequence, there is a 0th element, a 1st element, 2nd element, and so on. I indicate a sequence by using angle brackets $\langle\rangle$ to enclose its elements. For example, $\langle 3, 4, 5, 4\rangle$ is a sequence. Note that sequence $\langle 3, 5, 4, 4\rangle$ is distinct from sequence $\langle 3, 4, 5, 4\rangle$, and both are distinct from sequence $\langle 3, 4, 5\rangle$.

A **relation** R over set **S** is a set of ordered pairs from **S**. As an example of a relation, if **S** is $\{a, b, c\}$, then

$$\{\langle a, c\rangle, \langle b, c\rangle, \langle c, b\rangle\}$$

is a relation, and

$$\{\langle a, a\rangle, \langle a, c\rangle, \langle b, b\rangle, \langle b, c\rangle, \langle c, c\rangle\}$$

is a different relation. If tuple $\langle x, y\rangle$ is in relation R, we may use the infix notation xRy. We often use relations such as the less than operator ($<$) on the natural numbers, which includes ordered pairs such as $\langle 1, 3\rangle$ and $\langle 2, 23\rangle$, but not $\langle 3, 2\rangle$ or $\langle 2, 2\rangle$. Rather than writing the relationship in terms of ordered pairs, we typically use an infix notation for such relations, writing $1 < 3$.

Define the properties of relations as follows, with R a binary relation over set **S**.

- R is **reflexive** if aRa for all $a \in$ **S**.
- R is **symmetric** if whenever aRb, then bRa, for all $a, b \in$ **S**.
- R is **antisymmetric** if whenever aRb and bRa, then $a = b$, for all $a, b \in$ **S**.
- R is **transitive** if whenever aRb and bRc, then aRc, for all $a, b, c \in$ **S**.

As examples, for the natural numbers, $<$ is antisymmetric (because there is no case where aRb and bRa) and transitive; $\leq$ is reflexive, antisymmetric, and transitive, and $=$ is reflexive, symmetric (and antisymmetric!), and transitive. For people, the relation "is a sibling of" is symmetric and transitive. If we define a person to be a sibling of himself, then it is reflexive; if we define a person not to be a sibling of himself, then it is not reflexive.

R is an **equivalence relation** on set **S** if it is reflexive, symmetric, and transitive. An equivalence relation can be used to partition a set into **equivalence classes**. If two elements a and b are equivalent to each other, we write $a \equiv b$. A **partition** of a set **S** is a collection of subsets that are disjoint from each other and whose union is **S**. An equivalence relation on set **S** partitions the set into subsets whose elements are equivalent. See Section 6.2 for a discussion on how to represent equivalence classes on a set. One application for disjoint sets appears in Section 11.5.2.

Example 2.1 For the integers, $=$ is an equivalence relation that partitions each element into a distinct subset. In other words, for any integer a, three things are true.

1. $a = a$,

2. if $a = b$ then $b = a$, and
3. if $a = b$ and $b = c$, then $a = c$.

Of course, for distinct integers a, b, and c there are never cases where $a = b$, $b = a$, or $b = c$. So the claims that $=$ is symmetric and transitive are vacuously true (there are never examples in the relation where these events occur). But because the requirements for symmetry and transitivity are not violated, the relation is symmetric and transitive.

Example 2.2 If we clarify the definition of sibling to mean that a person is a sibling of him- or herself, then the sibling relation is an equivalence relation that partitions the set of people.

Example 2.3 We can use the modulus function (defined in the next section) to define an equivalence relation. For the set of integers, use the modulus function to define a binary relation such that two numbers x and y are in the relation if and only if $x \bmod m = y \bmod m$. Thus, for $m = 4$, $\langle 1, 5\rangle$ is in the relation because $1 \bmod 4 = 5 \bmod 4$. We see that modulus used in this way defines an equivalence relation on the integers, and this relation can be used to partition the integers into m equivalence classes. This relation is an equivalence relation because

1. $x \bmod m = x \bmod m$ for all x;
2. if $x \bmod m = y \bmod m$, then $y \bmod m = x \bmod m$; and
3. if $x \bmod m = y \bmod m$ and $y \bmod m = z \bmod m$, then $x \bmod m = z \bmod m$.

A binary relation is called a **partial order** if it is antisymmetric and transitive.[2] The set on which the partial order is defined is called a **partially ordered set** or a **poset**. Elements x and y of a set are **comparable** under a given relation if either xRy or yRx. If every pair of distinct elements in a partial order are comparable, then the order is called a **total order** or **linear order**.

Example 2.4 For the integers, relations $<$ and $\leq$ define partial orders. Operation $<$ is a total order because, for every pair of integers x and y such that $x \neq y$, either $x < y$ or $y < x$. Likewise, $\leq$ is a total order because, for every pair of integers x and y such that $x \neq y$, either $x \leq y$ or $y \leq x$.

---

[2] Not all authors use this definition for partial order. I have seen at least three significantly different definitions in the literature. I have selected the one that lets $<$ and $\leq$ both define partial orders on the integers, because this seems the most natural to me.

Example 2.5 For the powerset of the integers, the subset operator defines a partial order (because it is antisymmetric and transitive). For example, $\{1, 2\} \subseteq \{1, 2, 3\}$. However, sets $\{1, 2\}$ and $\{1, 3\}$ are not comparable by the subset operator, because neither is a subset of the other. Therefore, the subset operator does not define a total order on the powerset of the integers.

2.2 Miscellaneous Notation

Units of measure: I use the following notation for units of measure. "B" will be used as an abbreviation for bytes, "b" for bits, "KB" for kilobytes ($2^{10} = 1024$ bytes), "MB" for megabytes (2^{20} bytes), "GB" for gigabytes (2^{30} bytes), and "ms" for milliseconds (a millisecond is $\frac{1}{1000}$ of a second). Spaces are not placed between the number and the unit abbreviation when a power of two is intended. Thus a disk drive of size 25 gigabytes (where a gigabyte is intended as 2^{30} bytes) will be written as "25GB." Spaces are used when a decimal value is intended. An amount of 2000 bits would therefore be written "2 Kb" while "2Kb" represents 2048 bits. 2000 milliseconds is written as 2000 ms. Note that in this book large amounts of storage are nearly always measured in powers of two and times in powers of ten.

Factorial function: The **factorial** function, written $n!$ for n an integer greater than 0, is the product of the integers between 1 and n, inclusive. Thus, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. As a special case, $0! = 1$. The factorial function grows quickly as n becomes larger. Because computing the factorial function directly is a time-consuming process, it can be useful to have an equation that provides a good approximation. Stirling's approximation states that $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$, where $e \approx 2.71828$ (e is the base for the system of natural logarithms).[3] Thus we see that while $n!$ grows slower than n^n (because $\sqrt{2\pi n}/e^n < 1$), it grows faster than c^n for any positive integer constant c.

Permutations: A **permutation** of a sequence **S** is simply the members of **S** arranged in some order. For example, a permutation of the integers 1 through n would be those values arranged in some order. If the sequence contains n distinct members, then there are $n!$ different permutations for the sequence. This is because there are n choices for the first member in the permutation; for each choice of first member there are $n - 1$ choices for the second member, and so on. Sometimes one would like to obtain a **random permutation** for a sequence, that is, one of the $n!$ possible permutations is selected in such a way that each permutation has equal probability of being selected. A simple **C**++ function for generating a random permutation is as follows. Here, the n values of the sequence are stored in positions 0

[3] The symbol "≈" means "approximately equal."

through $n-1$ of array **A**, function **swap(A, i, j)** exchanges elements **i** and **j** in array **A**, and **Random(n)** returns an integer value in the range 0 to $n-1$ (see the Appendix for more information on **swap** and **Random**).

```
// Randomly permute the "n" values of array "A"
template<typename E>
void permute(E A[], int n) {
  for (int i=n; i>0; i--)
    swap(A, i-1, Random(i));
}
```

Boolean variables: A **Boolean variable** is a variable (of type **bool** in **C**++) that takes on one of the two values **true** and **false**. These two values are often associated with the values 1 and 0, respectively, although there is no reason why this needs to be the case. It is poor programming practice to rely on the correspondence between 0 and **false**, because these are logically distinct objects of different types.

Logic Notation: We will occasionally make use of the notation of symbolic or Boolean logic. $A \Rightarrow B$ means "A implies B" or "If A then B." $A \Leftrightarrow B$ means "A if and only if B" or "A is equivalent to B." $A \vee B$ means "A or B" (useful both in the context of symbolic logic or when performing a Boolean operation). $A \wedge B$ means "A and B." $\sim A$ and $\overline{A}$ both mean "not A" or the negation of A where A is a Boolean variable.

Floor and ceiling: The **floor** of x (written $\lfloor x \rfloor$) takes real value x and returns the greatest integer $\leq x$. For example, $\lfloor 3.4 \rfloor = 3$, as does $\lfloor 3.0 \rfloor$, while $\lfloor -3.4 \rfloor = -4$ and $\lfloor -3.0 \rfloor = -3$. The **ceiling** of x (written $\lceil x \rceil$) takes real value x and returns the least integer $\geq x$. For example, $\lceil 3.4 \rceil = 4$, as does $\lceil 4.0 \rceil$, while $\lceil -3.4 \rceil = \lceil -3.0 \rceil = -3$.

Modulus operator: The **modulus** (or **mod**) function returns the remainder of an integer division. Sometimes written $n \bmod m$ in mathematical expressions, the syntax for the **C**++ modulus operator is **n % m**. From the definition of remainder, $n \bmod m$ is the integer r such that $n = qm + r$ for q an integer, and $|r| < |m|$. Therefore, the result of $n \bmod m$ must be between 0 and $m-1$ when n and m are positive integers. For example, $5 \bmod 3 = 2$; $25 \bmod 3 = 1$, $5 \bmod 7 = 5$, and $5 \bmod 5 = 0$.

There is more than one way to assign values to q and r, depending on how integer division is interpreted. The most common mathematical definition computes the mod function as $n \bmod m = n - m\lfloor n/m \rfloor$. In this case, $-3 \bmod 5 = 2$. However, Java and **C**++ compilers typically use the underlying processor's machine instruction for computing integer arithmetic. On many computers this is done by truncating the resulting fraction, meaning $n \bmod m = n - m(\text{trunc}(n/m))$. Under this definition, $-3 \bmod 5 = -3$.

Unfortunately, for many applications this is not what the user wants or expects. For example, many hash systems will perform some computation on a record's key value and then take the result modulo the hash table size. The expectation here would be that the result is a legal index into the hash table, not a negative number. Implementers of hash functions must either insure that the result of the computation is always positive, or else add the hash table size to the result of the modulo function when that result is negative.

2.3 Logarithms

A **logarithm** of base b for value y is the power to which b is raised to get y. Normally, this is written as $\log_b y = x$. Thus, if $\log_b y = x$ then $b^x = y$, and $b^{log_b y} = y$.

Logarithms are used frequently by programmers. Here are two typical uses.

Example 2.6 Many programs require an encoding for a collection of objects. What is the minimum number of bits needed to represent n distinct code values? The answer is $\lceil \log_2 n \rceil$ bits. For example, if you have 1000 codes to store, you will require at least $\lceil \log_2 1000 \rceil = 10$ bits to have 1000 different codes (10 bits provide 1024 distinct code values).

Example 2.7 Consider the binary search algorithm for finding a given value within an array sorted by value from lowest to highest. Binary search first looks at the middle element and determines if the value being searched for is in the upper half or the lower half of the array. The algorithm then continues splitting the appropriate subarray in half until the desired value is found. (Binary search is described in more detail in Section 3.5.) How many times can an array of size n be split in half until only one element remains in the final subarray? The answer is $\lceil \log_2 n \rceil$ times.

In this book, nearly all logarithms used have a base of two. This is because data structures and algorithms most often divide things in half, or store codes with binary bits. Whenever you see the notation $\log n$ in this book, either $\log_2 n$ is meant or else the term is being used asymptotically and so the actual base does not matter. Logarithms using any base other than two will show the base explicitly.

Logarithms have the following properties, for any positive values of m, n, and r, and any positive integers a and b.

1. $\log(nm) = \log n + \log m$.
2. $\log(n/m) = \log n - \log m$.
3. $\log(n^r) = r \log n$.
4. $\log_a n = \log_b n / \log_b a$.

The first two properties state that the logarithm of two numbers multiplied (or divided) can be found by adding (or subtracting) the logarithms of the two numbers.[4] Property (3) is simply an extension of property (1). Property (4) tells us that, for variable n and any two integer constants a and b, $\log_a n$ and $\log_b n$ differ by the constant factor $\log_b a$, regardless of the value of n. Most runtime analyses in this book are of a type that ignores constant factors in costs. Property (4) says that such analyses need not be concerned with the base of the logarithm, because this can change the total cost only by a constant factor. Note that $2^{\log n} = n$.

When discussing logarithms, exponents often lead to confusion. Property (3) tells us that $\log n^2 = 2 \log n$. How do we indicate the square of the logarithm (as opposed to the logarithm of n^2)? This could be written as $(\log n)^2$, but it is traditional to use $\log^2 n$. On the other hand, we might want to take the logarithm of the logarithm of n. This is written $\log \log n$.

A special notation is used in the rare case when we need to know how many times we must take the log of a number before we reach a value ≤ 1. This quantity is written $\log^* n$. For example, $\log^* 1024 = 4$ because $\log 1024 = 10$, $\log 10 \approx 3.33$, $\log 3.33 \approx 1.74$, and $\log 1.74 < 1$, which is a total of 4 log operations.

2.4 Summations and Recurrences

Most programs contain loop constructs. When analyzing running time costs for programs with loops, we need to add up the costs for each time the loop is executed. This is an example of a **summation**. Summations are simply the sum of costs for some function applied to a range of parameter values. Summations are typically written with the following "Sigma" notation:

$$\sum_{i=1}^{n} f(i).$$

This notation indicates that we are summing the value of $f(i)$ over some range of (integer) values. The parameter to the expression and its initial value are indicated below the $\sum$ symbol. Here, the notation $i = 1$ indicates that the parameter is i and that it begins with the value 1. At the top of the $\sum$ symbol is the expression n. This indicates the maximum value for the parameter i. Thus, this notation means to sum the values of $f(i)$ as i ranges across the integers from 1 through n. This can also be

[4] These properties are the idea behind the slide rule. Adding two numbers can be viewed as joining two lengths together and measuring their combined length. Multiplication is not so easily done. However, if the numbers are first converted to the lengths of their logarithms, then those lengths can be added and the inverse logarithm of the resulting length gives the answer for the multiplication (this is simply logarithm property (1)). A slide rule measures the length of the logarithm for the numbers, lets you slide bars representing these lengths to add up the total length, and finally converts this total length to the correct numeric answer by taking the inverse of the logarithm for the result.

written $f(1) + f(2) + \cdots + f(n-1) + f(n)$. Within a sentence, Sigma notation is typeset as $\sum_{i=1}^{n} f(i)$.

Given a summation, you often wish to replace it with an algebraic equation with the same value as the summation. This is known as a **closed-form solution**, and the process of replacing the summation with its closed-form solution is known as **solving** the summation. For example, the summation $\sum_{i=1}^{n} 1$ is simply the expression "1" summed n times (remember that i ranges from 1 to n). Because the sum of n 1s is n, the closed-form solution is n. The following is a list of useful summations, along with their closed-form solutions.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \tag{2.1}$$

$$\sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6} = \frac{n(2n+1)(n+1)}{6}. \tag{2.2}$$

$$\sum_{i=1}^{\log n} n = n \log n. \tag{2.3}$$

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \text{ for } 0 < a < 1. \tag{2.4}$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} \text{ for } a \neq 1. \tag{2.5}$$

As special cases to Equation 2.5,

$$\sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n}, \tag{2.6}$$

and

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1. \tag{2.7}$$

As a corollary to Equation 2.7,

$$\sum_{i=0}^{\log n} 2^i = 2^{\log n + 1} - 1 = 2n - 1. \tag{2.8}$$

Finally,

$$\sum_{i=1}^{n} \frac{i}{2^i} = 2 - \frac{n+2}{2^n}. \tag{2.9}$$

The sum of reciprocals from 1 to n, called the **Harmonic Series** and written $\mathcal{H}_n$, has a value between $\log_e n$ and $\log_e n + 1$. To be more precise, as n grows, the

summation grows closer to

$$\mathcal{H}_n \approx \log_e n + \gamma + \frac{1}{2n}, \tag{2.10}$$

where γ is Euler's constant and has the value 0.5772...

Most of these equalities can be proved easily by mathematical induction (see Section 2.6.3). Unfortunately, induction does not help us derive a closed-form solution. It only confirms when a proposed closed-form solution is correct. Techniques for deriving closed-form solutions are discussed in Section 14.1.

The running time for a recursive algorithm is most easily expressed by a recursive expression because the total time for the recursive algorithm includes the time to run the recursive call(s). A **recurrence relation** defines a function by means of an expression that includes one or more (smaller) instances of itself. A classic example is the recursive definition for the factorial function:

$$n! = (n-1)! \cdot n \text{ for } n > 1; \quad 1! = 0! = 1.$$

Another standard example of a recurrence is the Fibonacci sequence:

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) \text{ for } n > 2; \quad \text{Fib}(1) = \text{Fib}(2) = 1.$$

From this definition, the first seven numbers of the Fibonacci sequence are

$$1, 1, 2, 3, 5, 8, \text{ and } 13.$$

Notice that this definition contains two parts: the general definition for $\text{Fib}(n)$ and the base cases for $\text{Fib}(1)$ and $\text{Fib}(2)$. Likewise, the definition for factorial contains a recursive part and base cases.

Recurrence relations are often used to model the cost of recursive functions. For example, the number of multiplications required by function **`fact`** of Section 2.5 for an input of size n will be zero when $n = 0$ or $n = 1$ (the base cases), and it will be one plus the cost of calling **`fact`** on a value of $n - 1$. This can be defined using the following recurrence:

$$\mathbf{T}(n) = \mathbf{T}(n-1) + 1 \text{ for } n > 1; \quad \mathbf{T}(0) = \mathbf{T}(1) = 0.$$

As with summations, we typically wish to replace the recurrence relation with a closed-form solution. One approach is to **expand** the recurrence by replacing any occurrences of $\mathbf{T}$ on the right-hand side with its definition.

Example 2.8 If we expand the recurrence $\mathbf{T}(n) = \mathbf{T}(n-1) + 1$, we get

$$\begin{aligned} \mathbf{T}(n) &= \mathbf{T}(n-1) + 1 \\ &= (\mathbf{T}(n-2) + 1) + 1. \end{aligned}$$

We can expand the recurrence as many steps as we like, but the goal is to detect some pattern that will permit us to rewrite the recurrence in terms of a summation. In this example, we might notice that

$$(\mathbf{T}(n-2)+1)+1 = \mathbf{T}(n-2)+2$$

and if we expand the recurrence again, we get

$$\mathbf{T}(n) = \mathbf{T}(n-2)+2 = \mathbf{T}(n-3)+1+2 = \mathbf{T}(n-3)+3$$

which generalizes to the pattern $\mathbf{T}(n) = \mathbf{T}(n-i)+i$. We might conclude that

$$\begin{aligned}\mathbf{T}(n) &= \mathbf{T}(n-(n-1))+(n-1)\\ &= \mathbf{T}(1)+n-1\\ &= n-1.\end{aligned}$$

Because we have merely guessed at a pattern and not actually proved that this is the correct closed form solution, we should use an induction proof to complete the process (see Example 2.13).

Example 2.9 A slightly more complicated recurrence is

$$\mathbf{T}(n) = \mathbf{T}(n-1)+n; \quad T(1) = 1.$$

Expanding this recurrence a few steps, we get

$$\begin{aligned}\mathbf{T}(n) &= \mathbf{T}(n-1)+n\\ &= \mathbf{T}(n-2)+(n-1)+n\\ &= \mathbf{T}(n-3)+(n-2)+(n-1)+n.\end{aligned}$$

We should then observe that this recurrence appears to have a pattern that leads to

$$\begin{aligned}\mathbf{T}(n) &= \mathbf{T}(n-(n-1))+(n-(n-2))+\cdots+(n-1)+n\\ &= 1+2+\cdots+(n-1)+n.\end{aligned}$$

This is equivalent to the summation $\sum_{i=1}^{n} i$, for which we already know the closed-form solution.

Techniques to find closed-form solutions for recurrence relations are discussed in Section 14.2. Prior to Chapter 14, recurrence relations are used infrequently in this book, and the corresponding closed-form solution and an explanation for how it was derived will be supplied at the time of use.

2.5 Recursion

An algorithm is **recursive** if it calls itself to do part of its work. For this approach to be successful, the "call to itself" must be on a smaller problem then the one originally attempted. In general, a recursive algorithm must have two parts: the **base case**, which handles a simple input that can be solved without resorting to a recursive call, and the recursive part which contains one or more recursive calls to the algorithm where the parameters are in some sense "closer" to the base case than those of the original call. Here is a recursive C++ function to compute the factorial of n. A trace of **fact**'s execution for a small value of n is presented in Section 4.2.4.

```
long fact(int n) {       // Compute n! recursively
  // To fit n! into a long variable, we require n <= 12
  Assert((n >= 0) && (n <= 12), "Input out of range");
  if (n <= 1)  return 1; // Base case: return base solution
  return n * fact(n-1);  // Recursive call for n > 1
}
```

The first two lines of the function constitute the base cases. If $n \leq 1$, then one of the base cases computes a solution for the problem. If $n > 1$, then **fact** calls a function that knows how to find the factorial of $n - 1$. Of course, the function that knows how to compute the factorial of $n - 1$ happens to be **fact** itself. But we should not think too hard about this while writing the algorithm. The design for recursive algorithms can always be approached in this way. First write the base cases. Then think about solving the problem by combining the results of one or more smaller — but similar — subproblems. If the algorithm you write is correct, then certainly you can rely on it (recursively) to solve the smaller subproblems. The secret to success is: Do not worry about *how* the recursive call solves the subproblem. Simply accept that it *will* solve it correctly, and use this result to in turn correctly solve the original problem. What could be simpler?

Recursion has no counterpart in everyday, physical-world problem solving. The concept can be difficult to grasp because it requires you to think about problems in a new way. To use recursion effectively, it is necessary to train yourself to stop analyzing the recursive process beyond the recursive call. The subproblems will take care of themselves. You just worry about the base cases and how to recombine the subproblems.

The recursive version of the factorial function might seem unnecessarily complicated to you because the same effect can be achieved by using a **while** loop. Here is another example of recursion, based on a famous puzzle called "Towers of Hanoi." The natural algorithm to solve this problem has multiple recursive calls. It cannot be rewritten easily using **while** loops.

The Towers of Hanoi puzzle begins with three poles and n rings, where all rings start on the leftmost pole (labeled Pole 1). The rings each have a different size, and

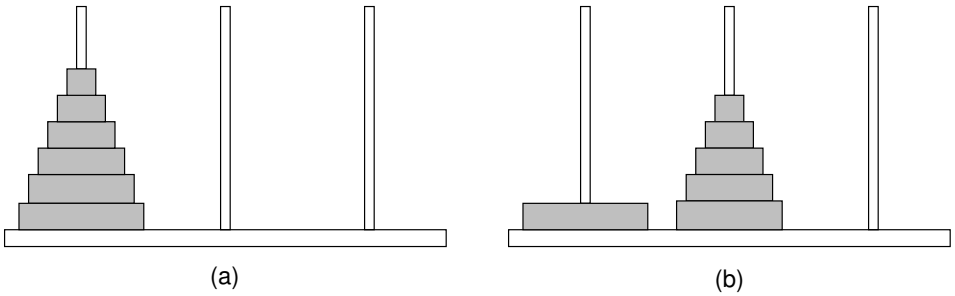


Figure 2.2 Towers of Hanoi example. (a) The initial conditions for a problem with six rings. (b) A necessary intermediate step on the road to a solution.

are stacked in order of decreasing size with the largest ring at the bottom, as shown in Figure 2.2(a). The problem is to move the rings from the leftmost pole to the rightmost pole (labeled Pole 3) in a series of steps. At each step the top ring on some pole is moved to another pole. There is one limitation on where rings may be moved: A ring can never be moved on top of a smaller ring.

How can you solve this problem? It is easy if you don't think too hard about the details. Instead, consider that all rings are to be moved from Pole 1 to Pole 3. It is not possible to do this without first moving the bottom (largest) ring to Pole 3. To do that, Pole 3 must be empty, and only the bottom ring can be on Pole 1. The remaining $n - 1$ rings must be stacked up in order on Pole 2, as shown in Figure 2.2(b). How can you do this? Assume that a function X is available to solve the problem of moving the top $n - 1$ rings from Pole 1 to Pole 2. Then move the bottom ring from Pole 1 to Pole 3. Finally, again use function X to move the remaining $n - 1$ rings from Pole 2 to Pole 3. In both cases, "function X" is simply the Towers of Hanoi function called on a smaller version of the problem.

The secret to success is relying on the Towers of Hanoi algorithm to do the work for you. You need not be concerned about the gory details of *how* the Towers of Hanoi subproblem will be solved. That will take care of itself provided that two things are done. First, there must be a base case (what to do if there is only one ring) so that the recursive process will not go on forever. Second, the recursive call to Towers of Hanoi can only be used to solve a smaller problem, and then only one of the proper form (one that meets the original definition for the Towers of Hanoi problem, assuming appropriate renaming of the poles).

Here is an implementation for the recursive Towers of Hanoi algorithm. Function **`move(start, goal)`** takes the top ring from Pole **`start`** and moves it to Pole **`goal`**. If **`move`** were to print the values of its parameters, then the result of calling **`TOH`** would be a list of ring-moving instructions that solves the problem.

```
void TOH(int n, Pole start, Pole goal, Pole temp) {
  if (n == 0) return;           // Base case
  TOH(n-1, start, temp, goal); // Recursive call: n-1 rings
  move(start, goal);            // Move bottom disk to goal
  TOH(n-1, temp, goal, start); // Recursive call: n-1 rings
}
```

Those who are unfamiliar with recursion might find it hard to accept that it is used primarily as a tool for simplifying the design and description of algorithms. A recursive algorithm usually does not yield the most efficient computer program for solving the problem because recursion involves function calls, which are typically more expensive than other alternatives such as a **`while`** loop. However, the recursive approach usually provides an algorithm that is reasonably efficient in the sense discussed in Chapter 3. (But not always! See Exercise 2.11.) If necessary, the clear, recursive solution can later be modified to yield a faster implementation, as described in Section 4.2.4.

Many data structures are naturally recursive, in that they can be defined as being made up of self-similar parts. Tree structures are an example of this. Thus, the algorithms to manipulate such data structures are often presented recursively. Many searching and sorting algorithms are based on a strategy of **divide and conquer**. That is, a solution is found by breaking the problem into smaller (similar) subproblems, solving the subproblems, then combining the subproblem solutions to form the solution to the original problem. This process is often implemented using recursion. Thus, recursion plays an important role throughout this book, and many more examples of recursive functions will be given.

2.6 Mathematical Proof Techniques

Solving any problem has two distinct parts: the investigation and the argument. Students are too used to seeing only the argument in their textbooks and lectures. But to be successful in school (and in life after school), one needs to be good at both, and to understand the differences between these two phases of the process. To solve the problem, you must investigate successfully. That means engaging the problem, and working through until you find a solution. Then, to give the answer to your client (whether that "client" be your instructor when writing answers on a homework assignment or exam, or a written report to your boss), you need to be able to make the argument in a way that gets the solution across clearly and succinctly. The argument phase involves good technical writing skills — the ability to make a clear, logical argument.

Being conversant with standard proof techniques can help you in this process. Knowing how to write a good proof helps in many ways. First, it clarifies your thought process, which in turn clarifies your explanations. Second, if you use one of the standard proof structures such as proof by contradiction or an induction proof,

then both you and your reader are working from a shared understanding of that structure. That makes for less complexity to your reader to understand your proof, because the reader need not decode the structure of your argument from scratch.

This section briefly introduces three commonly used proof techniques: (i) deduction, or direct proof; (ii) proof by contradiction, and (iii) proof by mathematical induction.

2.6.1 Direct Proof

In general, a **direct proof** is just a "logical explanation." A direct proof is sometimes referred to as an argument by deduction. This is simply an argument in terms of logic. Often written in English with words such as "if ... then," it could also be written with logic notation such as "$P \Rightarrow Q$." Even if we don't wish to use symbolic logic notation, we can still take advantage of fundamental theorems of logic to structure our arguments. For example, if we want to prove that P and Q are equivalent, we can first prove $P \Rightarrow Q$ and then prove $Q \Rightarrow P$.

In some domains, proofs are essentially a series of state changes from a start state to an end state. Formal predicate logic can be viewed in this way, with the various "rules of logic" being used to make the changes from one formula or combining a couple of formulas to make a new formula on the route to the destination. Symbolic manipulations to solve integration problems in introductory calculus classes are similar in spirit, as are high school geometry proofs.

2.6.2 Proof by Contradiction

The simplest way to *disprove* a theorem or statement is to find a counterexample to the theorem. Unfortunately, no number of examples supporting a theorem is sufficient to prove that the theorem is correct. However, there is an approach that is vaguely similar to disproving by counterexample, called Proof by Contradiction. To prove a theorem by contradiction, we first *assume* that the theorem is *false*. We then find a logical contradiction stemming from this assumption. If the logic used to find the contradiction is correct, then the only way to resolve the contradiction is to recognize that the assumption that the theorem is false must be incorrect. That is, we conclude that the theorem must be true.

Example 2.10 Here is a simple proof by contradiction.

Theorem 2.1 *There is no largest integer.*

Proof: Proof by contradiction.

Step 1. Contrary assumption: Assume that there *is* a largest integer. Call it B (for "biggest").

Step 2. Show this assumption leads to a contradiction: Consider $C = B + 1$. C is an integer because it is the sum of two integers. Also,

$C > B$, which means that B is not the largest integer after all. Thus, we have reached a contradiction. The only flaw in our reasoning is the initial assumption that the theorem is false. Thus, we conclude that the theorem is correct. □

A related proof technique is proving the contrapositive. We can prove that $P \Rightarrow Q$ by proving $(\text{not } Q) \Rightarrow (\text{not } P)$.

2.6.3 Proof by Mathematical Induction

Mathematical induction can be used to prove a wide variety of theorems. Induction also provides a useful way to think about algorithm design, because it encourages you to think about solving a problem by building up from simple subproblems. Induction can help to prove that a recursive function produces the correct result.. Understanding recursion is a big step toward understanding induction, and vice versa, since they work by essentially the same process.

Within the context of algorithm analysis, one of the most important uses for mathematical induction is as a method to test a hypothesis. As explained in Section 2.4, when seeking a closed-form solution for a summation or recurrence we might first guess or otherwise acquire evidence that a particular formula is the correct solution. If the formula is indeed correct, it is often an easy matter to prove that fact with an induction proof.

Let **Thrm** be a theorem to prove, and express **Thrm** in terms of a positive integer parameter n. Mathematical induction states that **Thrm** is true for any value of parameter n (for $n \geq c$, where c is some constant) if the following two conditions are true:

1. **Base Case: Thrm** holds for $n = c$, and
2. **Induction Step:** If **Thrm** holds for $n - 1$, then **Thrm** holds for n.

Proving the base case is usually easy, typically requiring that some small value such as 1 be substituted for n in the theorem and applying simple algebra or logic as necessary to verify the theorem. Proving the induction step is sometimes easy, and sometimes difficult. An alternative formulation of the induction step is known as **strong induction**. The induction step for strong induction is:

2a. Induction Step: If **Thrm** holds for all k, $c \leq k < n$, then **Thrm** holds for n.

Proving either variant of the induction step (in conjunction with verifying the base case) yields a satisfactory proof by mathematical induction.

The two conditions that make up the induction proof combine to demonstrate that **Thrm** holds for $n = 2$ as an extension of the fact that **Thrm** holds for $n = 1$. This fact, combined again with condition (2) or (2a), indicates that **Thrm** also holds

for $n = 3$, and so on. Thus, **Thrm** holds for all values of n (larger than the base cases) once the two conditions have been proved.

What makes mathematical induction so powerful (and so mystifying to most people at first) is that we can take advantage of the *assumption* that **Thrm** holds for all values less than n as a tool to help us prove that **Thrm** holds for n. This is known as the **induction hypothesis**. Having this assumption to work with makes the induction step easier to prove than tackling the original theorem itself. Being able to rely on the induction hypothesis provides extra information that we can bring to bear on the problem.

Recursion and induction have many similarities. Both are anchored on one or more base cases. A recursive function relies on the ability to call itself to get the answer for smaller instances of the problem. Likewise, induction proofs rely on the truth of the induction hypothesis to prove the theorem. The induction hypothesis does not come out of thin air. It is true if and only if the theorem itself is true, and therefore is reliable within the proof context. Using the induction hypothesis it do work is exactly the same as using a recursive call to do work.

Example 2.11 Here is a sample proof by mathematical induction. Call the sum of the first n positive integers $\mathbf{S}(n)$.

Theorem 2.2 $S(n) = n(n+1)/2$.

Proof: The proof is by mathematical induction.

1. **Check the base case.** For $n = 1$, verify that $\mathbf{S}(1) = 1(1+1)/2$. $\mathbf{S}(1)$ is simply the sum of the first positive number, which is 1. Because $1(1+1)/2 = 1$, the formula is correct for the base case.
2. **State the induction hypothesis.** The induction hypothesis is

$$\mathbf{S}(n-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)((n-1)+1)}{2} = \frac{(n-1)(n)}{2}.$$

3. **Use the assumption from the induction hypothesis for $n-1$ to show that the result is true for n.** The induction hypothesis states that $\mathbf{S}(n-1) = (n-1)(n)/2$, and because $\mathbf{S}(n) = \mathbf{S}(n-1) + n$, we can substitute for $\mathbf{S}(n-1)$ to get

$$\begin{aligned} \sum_{i=1}^{n} i &= \left(\sum_{i=1}^{n-1} i\right) + n = \frac{(n-1)(n)}{2} + n \\ &= \frac{n^2 - n + 2n}{2} = \frac{n(n+1)}{2}. \end{aligned}$$

Thus, by mathematical induction,

$$\mathbf{S}(n) = \sum_{i=1}^{n} i = n(n+1)/2.$$

□

Note carefully what took place in this example. First we cast $\mathbf{S}(n)$ in terms of a smaller occurrence of the problem: $\mathbf{S}(n) = \mathbf{S}(n-1) + n$. This is important because once $\mathbf{S}(n-1)$ comes into the picture, we can use the induction hypothesis to replace $\mathbf{S}(n-1)$ with $(n-1)(n)/2$. From here, it is simple algebra to prove that $\mathbf{S}(n-1) + n$ equals the right-hand side of the original theorem.

Example 2.12 Here is another simple proof by induction that illustrates choosing the proper variable for induction. We wish to prove by induction that the sum of the first n positive odd numbers is n^2. First we need a way to describe the nth odd number, which is simply $2n-1$. This also allows us to cast the theorem as a summation.

Theorem 2.3 $\sum_{i=1}^{n}(2i-1) = n^2$.

Proof: The base case of $n = 1$ yields $1 = 1^2$, which is true. The induction hypothesis is

$$\sum_{i=1}^{n-1}(2i-1) = (n-1)^2.$$

We now use the induction hypothesis to show that the theorem holds true for n. The sum of the first n odd numbers is simply the sum of the first $n-1$ odd numbers plus the nth odd number. In the second line below, we will use the induction hypothesis to replace the partial summation (shown in brackets in the first line) with its closed-form solution. After that, algebra takes care of the rest.

$$\begin{aligned}\sum_{i=1}^{n}(2i-1) &= \left[\sum_{i=1}^{n-1}(2i-1)\right] + 2n - 1 \\ &= [(n-1)^2] + 2n - 1 \\ &= n^2 - 2n + 1 + 2n - 1 \\ &= n^2.\end{aligned}$$

Thus, by mathematical induction, $\sum_{i=1}^{n}(2i-1) = n^2$. □

Example 2.13 This example shows how we can use induction to prove that a proposed closed-form solution for a recurrence relation is correct.

Theorem 2.4 *The recurrence relation* $\mathbf{T}(n) = \mathbf{T}(n-1)+1; \quad \mathbf{T}(1) = 0$ *has closed-form solution* $\mathbf{T}(n) = n - 1$.

Proof: To prove the base case, we observe that $\mathbf{T}(1) = 1 - 1 = 0$. The induction hypothesis is that $\mathbf{T}(n-1) = n - 2$. Combining the definition of the recurrence with the induction hypothesis, we see immediately that

$$\mathbf{T}(n) = \mathbf{T}(n-1) + 1 = n - 2 + 1 = n - 1$$

for $n > 1$. Thus, we have proved the theorem correct by mathematical induction. □

Example 2.14 This example uses induction without involving summations or other equations. It also illustrates a more flexible use of base cases.

Theorem 2.5 *2¢ and 5¢ stamps can be used to form any value (for values* ≥ 4*).*

Proof: The theorem defines the problem for values ≥ 4 because it does not hold for the values 1 and 3. Using 4 as the base case, a value of 4¢ can be made from two 2¢ stamps. The induction hypothesis is that a value of $n - 1$ can be made from some combination of 2¢ and 5¢ stamps. We now use the induction hypothesis to show how to get the value n from 2¢ and 5¢ stamps. Either the makeup for value $n - 1$ includes a 5¢ stamp, or it does not. If so, then replace a 5¢ stamp with three 2¢ stamps. If not, then the makeup must have included at least two 2¢ stamps (because it is at least of size 4 and contains only 2¢ stamps). In this case, replace two of the 2¢ stamps with a single 5¢ stamp. In either case, we now have a value of n made up of 2¢ and 5¢ stamps. Thus, by mathematical induction, the theorem is correct. □

Example 2.15 Here is an example using strong induction.

Theorem 2.6 *For* $\mathrm{n} > 1$*,* n *is divisible by some prime number.*

Proof: For the base case, choose $n = 2$. 2 is divisible by the prime number 2. The induction hypothesis is that *any* value a, $2 \leq a < n$, is divisible by some prime number. There are now two cases to consider when proving the theorem for n. If n is a prime number, then n is divisible by itself. If n is not a prime number, then $n = a \times b$ for a and b, both integers less than n but greater than 1. The induction hypothesis tells us that a is divisible by some prime number. That same prime number must also divide n. Thus, by mathematical induction, the theorem is correct. □

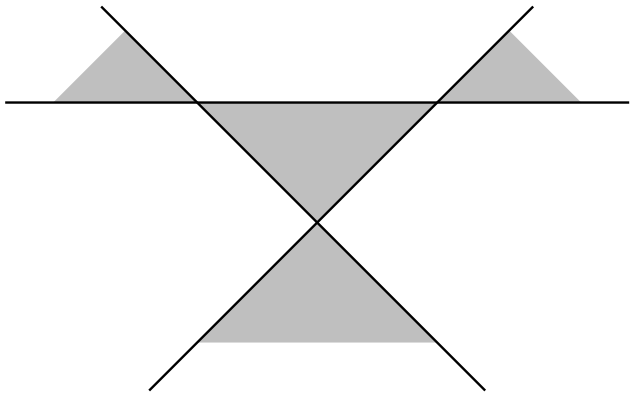


Figure 2.3 A two-coloring for the regions formed by three lines in the plane.

Our next example of mathematical induction proves a theorem from geometry. It also illustrates a standard technique of induction proof where we take n objects and remove some object to use the induction hypothesis.

Example 2.16 Define a **two-coloring** for a set of regions as a way of assigning one of two colors to each region such that no two regions sharing a side have the same color. For example, a chessboard is two-colored. Figure 2.3 shows a two-coloring for the plane with three lines. We will assume that the two colors to be used are black and white.

Theorem 2.7 *The set of regions formed by* n *infinite lines in the plane can be two-colored.*

Proof: Consider the base case of a single infinite line in the plane. This line splits the plane into two regions. One region can be colored black and the other white to get a valid two-coloring. The induction hypothesis is that the set of regions formed by $n-1$ infinite lines can be two-colored. To prove the theorem for n, consider the set of regions formed by the $n-1$ lines remaining when any one of the n lines is removed. By the induction hypothesis, this set of regions can be two-colored. Now, put the nth line back. This splits the plane into two half-planes, each of which (independently) has a valid two-coloring inherited from the two-coloring of the plane with $n-1$ lines. Unfortunately, the regions newly split by the nth line violate the rule for a two-coloring. Take all regions on one side of the nth line and reverse their coloring (after doing so, this half-plane is still two-colored). Those regions split by the nth line are now properly two-colored, because the part of the region to one side of the line is now black and the region to the other side is now white. Thus, by mathematical induction, the entire plane is two-colored. □

Compare the proof of Theorem 2.7 with that of Theorem 2.5. For Theorem 2.5, we took a collection of stamps of size $n-1$ (which, by the induction hypothesis, must have the desired property) and from that "built" a collection of size n that has the desired property. We therefore proved the existence of *some* collection of stamps of size n with the desired property.

For Theorem 2.7 we must prove that *any* collection of n lines has the desired property. Thus, our strategy is to take an *arbitrary* collection of n lines, and "reduce" it so that we have a set of lines that must have the desired property because it matches the induction hypothesis. From there, we merely need to show that reversing the original reduction process preserves the desired property.

In contrast, consider what is required if we attempt to "build" from a set of lines of size $n-1$ to one of size n. We would have great difficulty justifying that *all* possible collections of n lines are covered by our building process. By reducing from an arbitrary collection of n lines to something less, we avoid this problem.

This section's final example shows how induction can be used to prove that a recursive function produces the correct result.

Example 2.17 We would like to prove that function `fact` does indeed compute the factorial function. There are two distinct steps to such a proof. The first is to prove that the function always terminates. The second is to prove that the function returns the correct value.

Theorem 2.8 *Function* `fact` *will terminate for any value of* n.

Proof: For the base case, we observe that `fact` will terminate directly whenever $n \leq 0$. The induction hypothesis is that `fact` will terminate for $n-1$. For n, we have two possibilities. One possibility is that $n \geq 12$. In that case, `fact` will terminate directly because it will fail its assertion test. Otherwise, `fact` will make a recursive call to `fact(n-1)`. By the induction hypothesis, `fact(n-1)` must terminate. □

Theorem 2.9 *Function* `fact` *does compute the factorial function for any value in the range 0 to 12.*

Proof: To prove the base case, observe that when $n = 0$ or $n = 1$, `fact(n)` returns the correct value of 1. The induction hypothesis is that `fact(n-1)` returns the correct value of $(n-1)!$. For any value n within the legal range, `fact(n)` returns $n *$ `fact(n-1)`. By the induction hypothesis, `fact(n-1)` $= (n-1)!$, and because $n*(n-1)! = n!$, we have proved that `fact(n)` produces the correct result. □

We can use a similar process to prove many recursive programs correct. The general form is to show that the base cases perform correctly, and then to use the induction hypothesis to show that the recursive step also produces the correct result.

Prior to this, we must prove that the function always terminates, which might also be done using an induction proof.

2.7 Estimation

One of the most useful life skills that you can gain from your computer science training is the ability to perform quick estimates. This is sometimes known as "back of the napkin" or "back of the envelope" calculation. Both nicknames suggest that only a rough estimate is produced. Estimation techniques are a standard part of engineering curricula but are often neglected in computer science. Estimation is no substitute for rigorous, detailed analysis of a problem, but it can serve to indicate when a rigorous analysis is warranted: If the initial estimate indicates that the solution is unworkable, then further analysis is probably unnecessary.

Estimation can be formalized by the following three-step process:

1. Determine the major parameters that affect the problem.
2. Derive an equation that relates the parameters to the problem.
3. Select values for the parameters, and apply the equation to yield an estimated solution.

When doing estimations, a good way to reassure yourself that the estimate is reasonable is to do it in two different ways. In general, if you want to know what comes out of a system, you can either try to estimate that directly, or you can estimate what goes into the system (assuming that what goes in must later come out). If both approaches (independently) give similar answers, then this should build confidence in the estimate.

When calculating, be sure that your units match. For example, do not add feet and pounds. Verify that the result is in the correct units. Always keep in mind that the output of a calculation is only as good as its input. The more uncertain your valuation for the input parameters in Step 3, the more uncertain the output value. However, back of the envelope calculations are often meant only to get an answer within an order of magnitude, or perhaps within a factor of two. Before doing an estimate, you should decide on acceptable error bounds, such as within 25%, within a factor of two, and so forth. Once you are confident that an estimate falls within your error bounds, leave it alone! Do not try to get a more precise estimate than necessary for your purpose.

Example 2.18 How many library bookcases does it take to store books containing one million pages? I estimate that a 500-page book requires one inch on the library shelf (it will help to look at the size of any handy book), yielding about 200 feet of shelf space for one million pages. If a shelf is 4 feet wide, then 50 shelves are required. If a bookcase contains

5 shelves, this yields about 10 library bookcases. To reach this conclusion, I estimated the number of pages per inch, the width of a shelf, and the number of shelves in a bookcase. None of my estimates are likely to be precise, but I feel confident that my answer is correct to within a factor of two. (After writing this, I went to Virginia Tech's library and looked at some real bookcases. They were only about 3 feet wide, but typically had 7 shelves for a total of 21 shelf-feet. So I was correct to within 10% on bookcase capacity, far better than I expected or needed. One of my selected values was too high, and the other too low, which canceled out the errors.)

Example 2.19 Is it more economical to buy a car that gets 20 miles per gallon, or one that gets 30 miles per gallon but costs $3000 more? The typical car is driven about 12,000 miles per year. If gasoline costs $3/gallon, then the yearly gas bill is $1800 for the less efficient car and $1200 for the more efficient car. If we ignore issues such as the payback that would be received if we invested $3000 in a bank, it would take 5 years to make up the difference in price. At this point, the buyer must decide if price is the only criterion and if a 5-year payback time is acceptable. Naturally, a person who drives more will make up the difference more quickly, and changes in gasoline prices will also greatly affect the outcome.

Example 2.20 When at the supermarket doing the week's shopping, can you estimate about how much you will have to pay at the checkout? One simple way is to round the price of each item to the nearest dollar, and add this value to a mental running total as you put the item in your shopping cart. This will likely give an answer within a couple of dollars of the true total.

2.8 Further Reading

Most of the topics covered in this chapter are considered part of Discrete Mathematics. An introduction to this field is *Discrete Mathematics with Applications* by Susanna S. Epp [Epp10]. An advanced treatment of many mathematical topics useful to computer scientists is *Concrete Mathematics: A Foundation for Computer Science* by Graham, Knuth, and Patashnik [GKP94].

See "Technically Speaking" from the February 1995 issue of *IEEE Spectrum* [Sel95] for a discussion on the standard for indicating units of computer storage used in this book.

Introduction to Algorithms by Udi Manber [Man89] makes extensive use of mathematical induction as a technique for developing algorithms.

For more information on recursion, see *Thinking Recursively* by Eric S. Roberts [Rob86]. To learn recursion properly, it is worth your while to learn the programming languages LISP or Scheme, even if you never intend to write a program in either language. In particular, Friedman and Felleisen's "Little" books (including *The Little LISPer*[FF89] and *The Little Schemer*[FFBS95]) are designed to teach you how to think recursively as well as teach you the language. These books are entertaining reading as well.

A good book on writing mathematical proofs is Daniel Solow's *How to Read and Do Proofs* [Sol09]. To improve your general mathematical problem-solving abilities, see *The Art and Craft of Problem Solving* by Paul Zeitz [Zei07]. Zeitz also discusses the three proof techniques presented in Section 2.6, and the roles of investigation and argument in problem solving.

For more about estimation techniques, see two Programming Pearls by John Louis Bentley entitled *The Back of the Envelope* and *The Envelope is Back* [Ben84, Ben00, Ben86, Ben88]. *Genius: The Life and Science of Richard Feynman* by James Gleick [Gle92] gives insight into how important back of the envelope calculation was to the developers of the atomic bomb, and to modern theoretical physics in general.

2.9 Exercises

2.1 For each relation below, explain why the relation does or does not satisfy each of the properties reflexive, symmetric, antisymmetric, and transitive.

- **(a)** "isBrotherOf" on the set of people.
- **(b)** "isFatherOf" on the set of people.
- **(c)** The relation $R = \{\langle x, y\rangle \mid x^2 + y^2 = 1\}$ for real numbers x and y.
- **(d)** The relation $R = \{\langle x, y\rangle \mid x^2 = y^2\}$ for real numbers x and y.
- **(e)** The relation $R = \{\langle x, y\rangle \mid x \bmod y = 0\}$ for $x, y \in \{1, 2, 3, 4\}$.
- **(f)** The empty relation $\emptyset$ (i.e., the relation with no ordered pairs for which it is true) on the set of integers.
- **(g)** The empty relation $\emptyset$ (i.e., the relation with no ordered pairs for which it is true) on the empty set.

2.2 For each of the following relations, either prove that it is an equivalence relation or prove that it is not an equivalence relation.

- **(a)** For integers a and b, $a \equiv b$ if and only if $a + b$ is even.
- **(b)** For integers a and b, $a \equiv b$ if and only if $a + b$ is odd.
- **(c)** For nonzero rational numbers a and b, $a \equiv b$ if and only if $a \times b > 0$.
- **(d)** For nonzero rational numbers a and b, $a \equiv b$ if and only if a/b is an integer.

(e) For rational numbers a and b, $a \equiv b$ if and only if $a - b$ is an integer.
(f) For rational numbers a and b, $a \equiv b$ if and only if $|a - b| \leq 2$.

2.3 State whether each of the following relations is a partial ordering, and explain why or why not.
(a) "isFatherOf" on the set of people.
(b) "isAncestorOf" on the set of people.
(c) "isOlderThan" on the set of people.
(d) "isSisterOf" on the set of people.
(e) $\{\langle a, b\rangle, \langle a, a\rangle, \langle b, a\rangle\}$ on the set $\{a, b\}$.
(f) $\{\langle 2, 1\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$ on the set $\{1, 2, 3\}$.

2.4 How many total orderings can be defined on a set with n elements? Explain your answer.

2.5 Define an ADT for a set of integers (remember that a set has no concept of duplicate elements, and has no concept of order). Your ADT should consist of the functions that can be performed on a set to control its membership, check the size, check if a given element is in the set, and so on. Each function should be defined in terms of its input and output.

2.6 Define an ADT for a bag of integers (remember that a bag may contain duplicates, and has no concept of order). Your ADT should consist of the functions that can be performed on a bag to control its membership, check the size, check if a given element is in the set, and so on. Each function should be defined in terms of its input and output.

2.7 Define an ADT for a sequence of integers (remember that a sequence may contain duplicates, and supports the concept of position for its elements). Your ADT should consist of the functions that can be performed on a sequence to control its membership, check the size, check if a given element is in the set, and so on. Each function should be defined in terms of its input and output.

2.8 An investor places $30,000 into a stock fund. 10 years later the account has a value of $69,000. Using logarithms and anti-logarithms, present a formula for calculating the average annual rate of increase. Then use your formula to determine the average annual growth rate for this fund.

2.9 Rewrite the factorial function of Section 2.5 without using recursion.

2.10 Rewrite the **`for`** loop for the random permutation generator of Section 2.2 as a recursive function.

2.11 Here is a simple recursive function to compute the Fibonacci sequence:

```
long fibr(int n) { // Recursive Fibonacci generator
  // fibr(46) is largest value that fits in a long
  Assert((n > 0) && (n < 47), "Input out of range");
  if ((n == 1) || (n == 2)) return 1; // Base cases
  return fibr(n-1) + fibr(n-2);       // Recursion
}
```

This algorithm turns out to be very slow, calling **Fibr** a total of Fib(n) times. Contrast this with the following iterative algorithm:

```
long fibi(int n) { // Iterative Fibonacci generator
  // fibi(46) is largest value that fits in a long
  Assert((n > 0) && (n < 47), "Input out of range");
  long past, prev, curr;  // Store temporary values
  past = prev = curr = 1;    // initialize
  for (int i=3; i<=n; i++) { // Compute next value
    past = prev;             // past holds fibi(i-2)
    prev = curr;             // prev holds fibi(i-1)
    curr = past + prev;      // curr now holds fibi(i)
  }
  return curr;
}
```

Function **Fibi** executes the **for** loop $n - 2$ times.

(a) Which version is easier to understand? Why?

(b) Explain why **Fibr** is so much slower than **Fibi**.

2.12 Write a recursive function to solve a generalization of the Towers of Hanoi problem where each ring may begin on any pole so long as no ring sits on top of a smaller ring.

2.13 Revise the recursive implementation for Towers of Hanoi from Section 2.5 to return the list of moves needed to solve the problem.

2.14 Consider the following function:

```
void foo (double val) {
  if (val != 0.0)
    foo(val/2.0);
}
```

This function makes progress towards the base case on every recursive call. In theory (that is, if **double** variables acted like true real numbers), would this function ever terminate for input **val** a nonzero number? In practice (an actual computer implementation), will it terminate?

2.15 Write a function to print all of the permutations for the elements of an array containing n distinct integer values.

2.16 Write a recursive algorithm to print all of the subsets for the set of the first n positive integers.

2.17 The Largest Common Factor (LCF) for two positive integers n and m is the largest integer that divides both n and m evenly. LCF(n, m) is at least one, and at most m, assuming that $n \geq m$. Over two thousand years ago, Euclid provided an efficient algorithm based on the observation that, when $n \bmod m \neq 0$, LCF(n, m) = LCF(m, $n \bmod m$). Use this fact to write two algorithms to find the LCF for two positive integers. The first version should compute the value iteratively. The second version should compute the value using recursion.

2.18 Prove by contradiction that the number of primes is infinite.

2.19 **(a)** Use induction to show that $n^2 - n$ is always even.
(b) Give a direct proof in one or two sentences that $n^2 - n$ is always even.
(c) Show that $n^3 - n$ is always divisible by three.
(d) Is $n^5 - n$ aways divisible by 5? Explain your answer.

2.20 Prove that $\sqrt{2}$ is irrational.

2.21 Explain why

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n} (n - i + 1) = \sum_{i=0}^{n-1} (n - i).$$

2.22 Prove Equation 2.2 using mathematical induction.

2.23 Prove Equation 2.6 using mathematical induction.

2.24 Prove Equation 2.7 using mathematical induction.

2.25 Find a closed-form solution and prove (using induction) that your solution is correct for the summation

$$\sum_{i=1}^{n} 3^i.$$

2.26 Prove that the sum of the first n even numbers is $n^2 + n$

(a) by assuming that the sum of the first n odd numbers is n^2.
(b) by mathematical induction.

2.27 Give a closed-form formula for the summation $\sum_{i=a}^{n} i$ where a is an integer between 1 and n.

2.28 Prove that $\text{Fib}(n) < (\frac{5}{3})^n$.

2.29 Prove, for $n \geq 1$, that

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}.$$

2.30 The following theorem is called the **Pigeonhole Principle**.

Theorem 2.10 *When* $\mathrm{n} + 1$ *pigeons roost in* n *holes, there must be some hole containing at least two pigeons.*

(a) Prove the Pigeonhole Principle using proof by contradiction.
(b) Prove the Pigeonhole Principle using mathematical induction.

2.31 For this problem, you will consider arrangements of infinite lines in the plane such that three or more lines never intersect at a single point and no two lines are parallel.

(a) Give a recurrence relation that expresses the number of regions formed by n lines, and explain why your recurrence is correct.
(b) Give the summation that results from expanding your recurrence.

(c) Give a closed-form solution for the summation.

2.32 Prove (using induction) that the recurrence $\mathbf{T}(n) = \mathbf{T}(n-1) + n;\ \mathbf{T}(1) = 1$ has as its closed-form solution $\mathbf{T}(n) = n(n+1)/2$.

2.33 Expand the following recurrence to help you find a closed-form solution, and then use induction to prove your answer is correct.

$$\mathbf{T}(n) = 2\mathbf{T}(n-1) + 1 \text{ for } n > 0;\ \mathbf{T}(0) = 0.$$

2.34 Expand the following recurrence to help you find a closed-form solution, and then use induction to prove your answer is correct.

$$\mathbf{T}(n) = \mathbf{T}(n-1) + 3n + 1 \text{ for } n > 0;\ \mathbf{T}(0) = 1.$$

2.35 Assume that an n-bit integer (represented by standard binary notation) takes any value in the range 0 to $2^n - 1$ with equal probability.

(a) For each bit position, what is the probability of its value being 1 and what is the probability of its value being 0?

(b) What is the average number of "1" bits for an n-bit random number?

(c) What is the expected value for the position of the leftmost "1" bit? In other words, how many positions on average must we examine when moving from left to right before encountering a "1" bit? Show the appropriate summation.

2.36 What is the total volume of your body in liters (or, if you prefer, gallons)?

2.37 An art historian has a database of 20,000 full-screen color images.

(a) About how much space will this require? How many CDs would be required to store the database? (A CD holds about 600MB of data). Be sure to explain all assumptions you made to derive your answer.

(b) Now, assume that you have access to a good image compression technique that can store the images in only 1/10 of the space required for an uncompressed image. Will the entire database fit onto a single CD if the images are compressed?

2.38 How many cubic miles of water flow out of the mouth of the Mississippi River each day? DO NOT look up the answer or any supplemental facts. Be sure to describe all assumptions made in arriving at your answer.

2.39 When buying a home mortgage, you often have the option of paying some money in advance (called "discount points") to get a lower interest rate. Assume that you have the choice between two 15-year fixed-rate mortgages: one at 8% with no up-front charge, and the other at $7\frac{3}{4}\%$ with an up-front charge of 1% of the mortgage value. How long would it take to recover the 1% charge when you take the mortgage at the lower rate? As a second, more

precise estimate, how long would it take to recover the charge plus the interest you would have received if you had invested the equivalent of the 1% charge in the bank at 5% interest while paying the higher rate? DO NOT use a calculator to help you answer this question.

2.40 When you build a new house, you sometimes get a "construction loan" which is a temporary line of credit out of which you pay construction costs as they occur. At the end of the construction period, you then replace the construction loan with a regular mortgage on the house. During the construction loan, you only pay each month for the interest charged against the actual amount borrowed so far. Assume that your house construction project starts at the beginning of April, and is complete at the end of six months. Assume that the total construction cost will be $300,000 with the costs occurring at the beginning of each month in $50,000 increments. The construction loan charges 6% interest. Estimate the total interest payments that must be paid over the life of the construction loan.

2.41 Here are some questions that test your working knowledge of how fast computers operate. Is disk drive access time normally measured in milliseconds (thousandths of a second) or microseconds (millionths of a second)? Does your RAM memory access a word in more or less than one microsecond? How many instructions can your CPU execute in one year if the machine is left running at full speed all the time? DO NOT use paper or a calculator to derive your answers.

2.42 Does your home contain enough books to total one million pages? How many total pages are stored in your school library building? Explain how you got your answer.

2.43 How many words are in this book? Explain how you got your answer.

2.44 How many hours are one million seconds? How many days? Answer these questions doing all arithmetic in your head. Explain how you got your answer.

2.45 How many cities and towns are there in the United States? Explain how you got your answer.

2.46 How many steps would it take to walk from Boston to San Francisco? Explain how you got your answer.

2.47 A man begins a car trip to visit his in-laws. The total distance is 60 miles, and he starts off at a speed of 60 miles per hour. After driving exactly 1 mile, he loses some of his enthusiasm for the journey, and (instantaneously) slows down to 59 miles per hour. After traveling another mile, he again slows to 58 miles per hour. This continues, progressively slowing by 1 mile per hour for each mile traveled until the trip is complete.

(a) How long does it take the man to reach his in-laws?

(b) How long would the trip take in the continuous case where the speed smoothly diminishes with the distance yet to travel?

3

Algorithm Analysis

How long will it take to process the company payroll once we complete our planned merger? Should I buy a new payroll program from vendor X or vendor Y? If a particular program is slow, is it badly implemented or is it solving a hard problem? Questions like these ask us to consider the difficulty of a problem, or the relative efficiency of two or more approaches to solving a problem.

This chapter introduces the motivation, basic notation, and fundamental techniques of algorithm analysis. We focus on a methodology known as **asymptotic algorithm analysis**, or simply **asymptotic analysis**. Asymptotic analysis attempts to estimate the resource consumption of an algorithm. It allows us to compare the relative costs of two or more algorithms for solving the same problem. Asymptotic analysis also gives algorithm designers a tool for estimating whether a proposed solution is likely to meet the resource constraints for a problem before they implement an actual program. After reading this chapter, you should understand

- the concept of a growth rate, the rate at which the cost of an algorithm grows as the size of its input grows;
- the concept of upper and lower bounds for a growth rate, and how to estimate these bounds for a simple program, algorithm, or problem; and
- the difference between the cost of an algorithm (or program) and the cost of a problem.

The chapter concludes with a brief discussion of the practical difficulties encountered when empirically measuring the cost of a program, and some principles for code tuning to improve program efficiency.

3.1 Introduction

How do you compare two algorithms for solving some problem in terms of efficiency? We could implement both algorithms as computer programs and then run

them on a suitable range of inputs, measuring how much of the resources in question each program uses. This approach is often unsatisfactory for four reasons. First, there is the effort involved in programming and testing two algorithms when at best you want to keep only one. Second, when empirically comparing two algorithms there is always the chance that one of the programs was "better written" than the other, and therefor the relative qualities of the underlying algorithms are not truly represented by their implementations. This can easily occur when the programmer has a bias regarding the algorithms. Third, the choice of empirical test cases might unfairly favor one algorithm. Fourth, you could find that even the better of the two algorithms does not fall within your resource budget. In that case you must begin the entire process again with yet another program implementing a new algorithm. But, how would you know if any algorithm can meet the resource budget? Perhaps the problem is simply too difficult for any implementation to be within budget.

These problems can often be avoided by using asymptotic analysis. Asymptotic analysis measures the efficiency of an algorithm, or its implementation as a program, as the input size becomes large. It is actually an estimating technique and does not tell us anything about the relative merits of two programs where one is always "slightly faster" than the other. However, asymptotic analysis has proved useful to computer scientists who must determine if a particular algorithm is worth considering for implementation.

The critical resource for a program is most often its running time. However, you cannot pay attention to running time alone. You must also be concerned with other factors such as the space required to run the program (both main memory and disk space). Typically you will analyze the *time* required for an *algorithm* (or the instantiation of an algorithm in the form of a program), and the *space* required for a *data structure*.

Many factors affect the running time of a program. Some relate to the environment in which the program is compiled and run. Such factors include the speed of the computer's CPU, bus, and peripheral hardware. Competition with other users for the computer's (or the network's) resources can make a program slow to a crawl. The programming language and the quality of code generated by a particular compiler can have a significant effect. The "coding efficiency" of the programmer who converts the algorithm to a program can have a tremendous impact as well.

If you need to get a program working within time and space constraints on a particular computer, all of these factors can be relevant. Yet, none of these factors address the differences between two algorithms or data structures. To be fair, programs derived from two algorithms for solving the same problem should both be compiled with the same compiler and run on the same computer under the same conditions. As much as possible, the same amount of care should be taken in the programming effort devoted to each program to make the implementations "equally

efficient." In this sense, all of the factors mentioned above should cancel out of the comparison because they apply to both algorithms equally.

If you truly wish to understand the running time of an algorithm, there are other factors that are more appropriate to consider than machine speed, programming language, compiler, and so forth. Ideally we would measure the running time of the algorithm under standard benchmark conditions. However, we have no way to calculate the running time reliably other than to run an implementation of the algorithm on some computer. The only alternative is to use some other measure as a surrogate for running time.

Of primary consideration when estimating an algorithm's performance is the number of **basic operations** required by the algorithm to process an input of a certain **size**. The terms "basic operations" and "size" are both rather vague and depend on the algorithm being analyzed. Size is often the number of inputs processed. For example, when comparing sorting algorithms, the size of the problem is typically measured by the number of records to be sorted. A basic operation must have the property that its time to complete does not depend on the particular values of its operands. Adding or comparing two integer variables are examples of basic operations in most programming languages. Summing the contents of an array containing n integers is not, because the cost depends on the value of n (i.e., the size of the input).

Example 3.1 Consider a simple algorithm to solve the problem of finding the largest value in an array of n integers. The algorithm looks at each integer in turn, saving the position of the largest value seen so far. This algorithm is called the *largest-value sequential search* and is illustrated by the following function:

```
// Return position of largest value in "A" of size "n"
int largest(int A[], int n) {
  int currlarge = 0; // Holds largest element position
  for (int i=1; i<n; i++)   // For each array element
   if (A[currlarge] < A[i]) // if A[i] is larger
      currlarge = i;        //   remember its position
  return currlarge;         // Return largest position
}
```

Here, the size of the problem is **`A.length`**, the number of integers stored in array **`A`**. The basic operation is to compare an integer's value to that of the largest value seen so far. It is reasonable to assume that it takes a fixed amount of time to do one such comparison, regardless of the value of the two integers or their positions in the array.

Because the most important factor affecting running time is normally size of the input, for a given input size n we often express the time **T** to run

the algorithm as a function of n, written as $\mathbf{T}(n)$. We will always assume $\mathbf{T}(n)$ is a non-negative value.

Let us call c the amount of time required to compare two integers in function **largest**. We do not care right now what the precise value of c might be. Nor are we concerned with the time required to increment variable i because this must be done for each value in the array, or the time for the actual assignment when a larger value is found, or the little bit of extra time taken to initialize **currlarge**. We just want a reasonable approximation for the time taken to execute the algorithm. The total time to run **largest** is therefore approximately cn, because we must make n comparisons, with each comparison costing c time. We say that function **largest** (and by extension ,the largest-value sequential search algorithm for any typical implementation) has a running time expressed by the equation

$$\mathbf{T}(n) = cn.$$

This equation describes the growth rate for the running time of the largest-value sequential search algorithm.

Example 3.2 The running time of a statement that assigns the first value of an integer array to a variable is simply the time required to copy the value of the first array value. We can assume this assignment takes a constant amount of time regardless of the value. Let us call c_1 the amount of time necessary to copy an integer. No matter how large the array on a typical computer (given reasonable conditions for memory and array size), the time to copy the value from the first position of the array is always c_1. Thus, the equation for this algorithm is simply

$$\mathbf{T}(n) = c_1,$$

indicating that the size of the input n has no effect on the running time. This is called a **constant** running time.

Example 3.3 Consider the following code:

```
sum = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=n; j++)
      sum++;
```

What is the running time for this code fragment? Clearly it takes longer to run when n is larger. The basic operation in this example is the increment

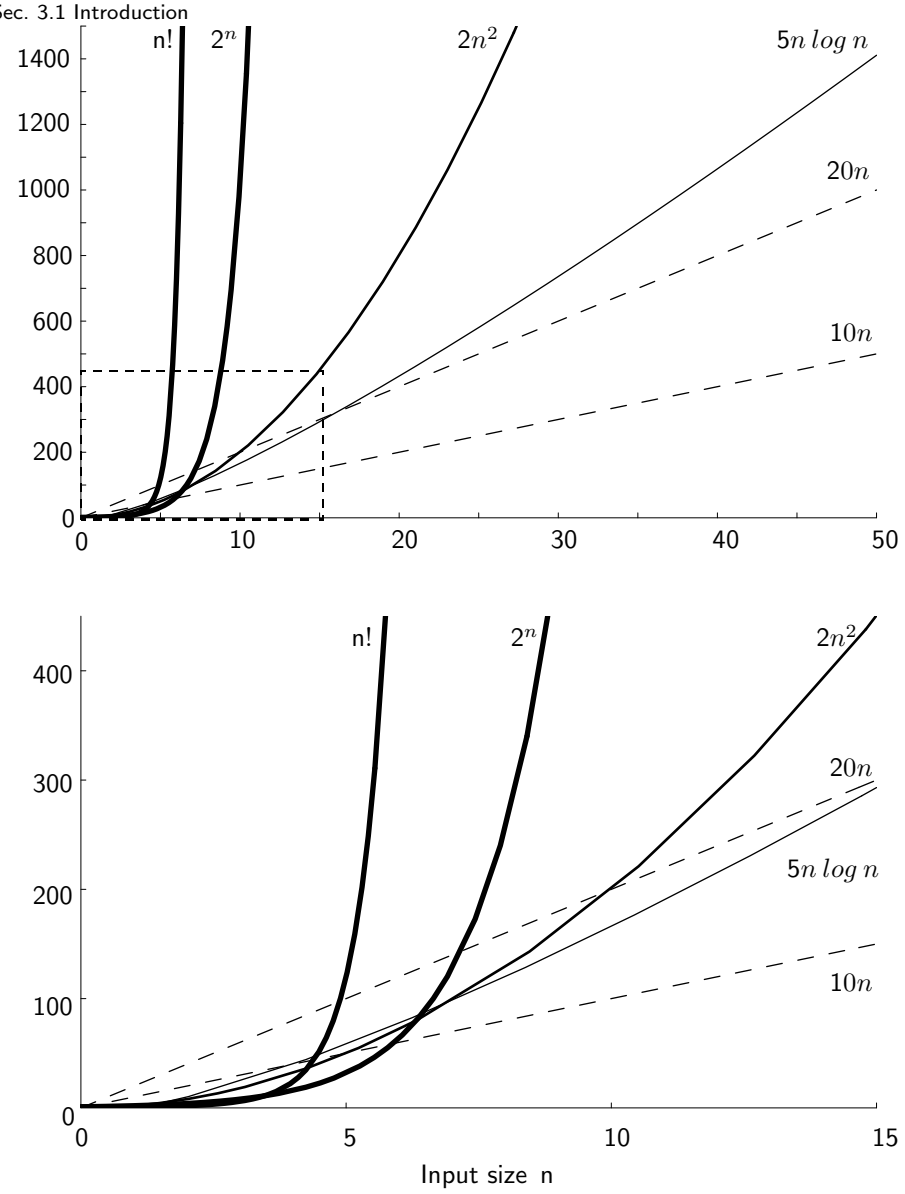


Figure 3.1 Two views of a graph illustrating the growth rates for six equations. The bottom view shows in detail the lower-left portion of the top view. The horizontal axis represents input size. The vertical axis can represent time, space, or any other measure of cost.

operation for variable *sum*. We can assume that incrementing takes constant time; call this time c_2. (We can ignore the time required to initialize *sum*, and to increment the loop counters i and j. In practice, these costs can safely be bundled into time c_2.) The total number of increment operations is n^2. Thus, we say that the running time is $\mathbf{T}(n) = c_2 n^2$.

n	$\log\log n$	$\log n$	n	$n \log n$	n^2	n^3	2^n
16	2	4	2^4	$4 \cdot 2^4 = 2^6$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	$2^{64\mathrm{K}}$
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	$2^{1\mathrm{M}}$
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	$2^{1\mathrm{G}}$

Figure 3.2 Costs for growth rates representative of most computer algorithms.

The **growth rate** for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows. Figure 3.1 shows a graph for six equations, each meant to describe the running time for a particular program or algorithm. A variety of growth rates representative of typical algorithms are shown. The two equations labeled $10n$ and $20n$ are graphed by straight lines. A growth rate of cn (for c any positive constant) is often referred to as a **linear** growth rate or running time. This means that as the value of n grows, the running time of the algorithm grows in the same proportion. Doubling the value of n roughly doubles the running time. An algorithm whose running-time equation has a highest-order term containing a factor of n^2 is said to have a **quadratic** growth rate. In Figure 3.1, the line labeled $2n^2$ represents a quadratic growth rate. The line labeled 2^n represents an **exponential** growth rate. This name comes from the fact that n appears in the exponent. The line labeled $n!$ is also growing exponentially.

As you can see from Figure 3.1, the difference between an algorithm whose running time has cost $\mathbf{T}(n) = 10n$ and another with cost $\mathbf{T}(n) = 2n^2$ becomes tremendous as n grows. For $n > 5$, the algorithm with running time $\mathbf{T}(n) = 2n^2$ is already much slower. This is despite the fact that $10n$ has a greater constant factor than $2n^2$. Comparing the two curves marked $20n$ and $2n^2$ shows that changing the constant factor for one of the equations only shifts the point at which the two curves cross. For $n > 10$, the algorithm with cost $\mathbf{T}(n) = 2n^2$ is slower than the algorithm with cost $\mathbf{T}(n) = 20n$. This graph also shows that the equation $\mathbf{T}(n) = 5n \log n$ grows somewhat more quickly than both $\mathbf{T}(n) = 10n$ and $\mathbf{T}(n) = 20n$, but not nearly so quickly as the equation $\mathbf{T}(n) = 2n^2$. For constants $a, b > 1$, n^a grows faster than either $\log^b n$ or $\log n^b$. Finally, algorithms with cost $\mathbf{T}(n) = 2^n$ or $\mathbf{T}(n) = n!$ are prohibitively expensive for even modest values of n. Note that for constants $a, b \geq 1$, a^n grows faster than n^b.

We can get some further insight into relative growth rates for various algorithms from Figure 3.2. Most of the growth rates that appear in typical algorithms are shown, along with some representative input sizes. Once again, we see that the growth rate has a tremendous effect on the resources consumed by an algorithm.

3.2 Best, Worst, and Average Cases

Consider the problem of finding the factorial of n. For this problem, there is only one input of a given "size" (that is, there is only a single instance for each size of n). Now consider our largest-value sequential search algorithm of Example 3.1, which always examines every array value. This algorithm works on many inputs of a given size n. That is, there are many possible arrays of any given size. However, no matter what array of size n that the algorithm looks at, its cost will always be the same in that it always looks at every element in the array one time.

For some algorithms, different inputs of a given size require different amounts of time. For example, consider the problem of searching an array containing n integers to find the one with a particular value K (assume that K appears exactly once in the array). The **sequential search** algorithm begins at the first position in the array and looks at each value in turn until K is found. Once K is found, the algorithm stops. This is different from the largest-value sequential search algorithm of Example 3.1, which always examines every array value.

There is a wide range of possible running times for the sequential search algorithm. The first integer in the array could have value K, and so only one integer is examined. In this case the running time is short. This is the **best case** for this algorithm, because it is not possible for sequential search to look at less than one value. Alternatively, if the last position in the array contains K, then the running time is relatively long, because the algorithm must examine n values. This is the **worst case** for this algorithm, because sequential search never looks at more than n values. If we implement sequential search as a program and run it many times on many different arrays of size n, or search for many different values of K within the same array, we expect the algorithm on average to go halfway through the array before finding the value we seek. On average, the algorithm examines about $n/2$ values. We call this the **average case** for this algorithm.

When analyzing an algorithm, should we study the best, worst, or average case? Normally we are not interested in the best case, because this might happen only rarely and generally is too optimistic for a fair characterization of the algorithm's running time. In other words, analysis based on the best case is not likely to be representative of the behavior of the algorithm. However, there are rare instances where a best-case analysis is useful — in particular, when the best case has high probability of occurring. In Chapter 7 you will see some examples where taking advantage of the best-case running time for one sorting algorithm makes a second more efficient.

How about the worst case? The advantage to analyzing the worst case is that you know for certain that the algorithm must perform at least that well. This is especially important for real-time applications, such as for the computers that monitor an air traffic control system. Here, it would not be acceptable to use an algorithm

that can handle n airplanes quickly enough *most of the time*, but which fails to perform quickly enough when all n airplanes are coming from the same direction.

For other applications — particularly when we wish to aggregate the cost of running the program many times on many different inputs — worst-case analysis might not be a representative measure of the algorithm's performance. Often we prefer to know the average-case running time. This means that we would like to know the *typical* behavior of the algorithm on inputs of size n. Unfortunately, average-case analysis is not always possible. Average-case analysis first requires that we understand how the actual inputs to the program (and their costs) are distributed with respect to the set of all possible inputs to the program. For example, it was stated previously that the sequential search algorithm on average examines half of the array values. This is only true if the element with value K is equally likely to appear in any position in the array. If this assumption is not correct, then the algorithm does *not* necessarily examine half of the array values in the average case. See Section 9.2 for further discussion regarding the effects of data distribution on the sequential search algorithm.

The characteristics of a data distribution have a significant effect on many search algorithms, such as those based on hashing (Section 9.4) and search trees (e.g., see Section 5.4). Incorrect assumptions about data distribution can have disastrous consequences on a program's space or time performance. Unusual data distributions can also be used to advantage, as shown in Section 9.2.

In summary, for real-time applications we are likely to prefer a worst-case analysis of an algorithm. Otherwise, we often desire an average-case analysis if we know enough about the distribution of our input to compute the average case. If not, then we must resort to worst-case analysis.

3.3 A Faster Computer, or a Faster Algorithm?

Imagine that you have a problem to solve, and you know of an algorithm whose running time is proportional to n^2. Unfortunately, the resulting program takes ten times too long to run. If you replace your current computer with a new one that is ten times faster, will the n^2 algorithm become acceptable? If the problem size remains the same, then perhaps the faster computer will allow you to get your work done quickly enough even with an algorithm having a high growth rate. But a funny thing happens to most people who get a faster computer. They don't run the same problem faster. They run a bigger problem! Say that on your old computer you were content to sort 10,000 records because that could be done by the computer during your lunch break. On your new computer you might hope to sort 100,000 records in the same time. You won't be back from lunch any sooner, so you are better off solving a larger problem. And because the new machine is ten times faster, you would like to sort ten times as many records.

f(n)	n	n′	Change	n′/n
10n	1000	10,000	$n' = 10n$	10
20n	500	5000	$n' = 10n$	10
5n log n	250	1842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	−−

Figure 3.3 The increase in problem size that can be run in a fixed period of time on a computer that is ten times faster. The first column lists the right-hand sides for each of five growth rate equations from Figure 3.1. For the purpose of this example, arbitrarily assume that the old machine can run 10,000 basic operations in one hour. The second column shows the maximum value for n that can be run in 10,000 basic operations on the old machine. The third column shows the value for n', the new maximum size for the problem that can be run in the same time on the new machine that is ten times faster. Variable n' is the greatest size for the problem that can run in 100,000 basic operations. The fourth column shows how the size of n changed to become n' on the new machine. The fifth column shows the increase in the problem size as the ratio of n' to n.

If your algorithm's growth rate is linear (i.e., if the equation that describes the running time on input size n is $\mathbf{T}(n) = cn$ for some constant c), then 100,000 records on the new machine will be sorted in the same time as 10,000 records on the old machine. If the algorithm's growth rate is greater than cn, such as c_1n^2, then you will *not* be able to do a problem ten times the size in the same amount of time on a machine that is ten times faster.

How much larger a problem can be solved in a given amount of time by a faster computer? Assume that the new machine is ten times faster than the old. Say that the old machine could solve a problem of size n in an hour. What is the largest problem that the new machine can solve in one hour? Figure 3.3 shows how large a problem can be solved on the two machines for five of the running-time functions from Figure 3.1.

This table illustrates many important points. The first two equations are both linear; only the value of the constant factor has changed. In both cases, the machine that is ten times faster gives an increase in problem size by a factor of ten. In other words, while the value of the constant does affect the absolute size of the problem that can be solved in a fixed amount of time, it does not affect the *improvement* in problem size (as a proportion to the original size) gained by a faster computer. This relationship holds true regardless of the algorithm's growth rate: Constant factors never affect the relative improvement gained by a faster computer.

An algorithm with time equation $\mathbf{T}(n) = 2n^2$ does not receive nearly as great an improvement from the faster machine as an algorithm with linear growth rate. Instead of an improvement by a factor of ten, the improvement is only the square

root of that: $\sqrt{10} \approx 3.16$. Thus, the algorithm with higher growth rate not only solves a smaller problem in a given time in the first place, it *also* receives less of a speedup from a faster computer. As computers get ever faster, the disparity in problem sizes becomes ever greater.

The algorithm with growth rate $\mathbf{T}(n) = 5n \log n$ improves by a greater amount than the one with quadratic growth rate, but not by as great an amount as the algorithms with linear growth rates.

Note that something special happens in the case of the algorithm whose running time grows exponentially. In Figure 3.1, the curve for the algorithm whose time is proportional to 2^n goes up very quickly. In Figure 3.3, the increase in problem size on the machine ten times as fast is shown to be about $n + 3$ (to be precise, it is $n + \log_2 10$). The increase in problem size for an algorithm with exponential growth rate is by a constant addition, not by a multiplicative factor. Because the old value of n was 13, the new problem size is 16. If next year you buy another computer ten times faster yet, then the new computer (100 times faster than the original computer) will only run a problem of size 19. If you had a second program whose growth rate is 2^n and for which the original computer could run a problem of size 1000 in an hour, than a machine ten times faster can run a problem only of size 1003 in an hour! Thus, an exponential growth rate is radically different than the other growth rates shown in Figure 3.3. The significance of this difference is explored in Chapter 17.

Instead of buying a faster computer, consider what happens if you replace an algorithm whose running time is proportional to n^2 with a new algorithm whose running time is proportional to $n \log n$. In the graph of Figure 3.1, a fixed amount of time would appear as a horizontal line. If the line for the amount of time available to solve your problem is above the point at which the curves for the two growth rates in question meet, then the algorithm whose running time grows less quickly is faster. An algorithm with running time $\mathbf{T}(n) = n^2$ requires $1024 \times 1024 = 1,048,576$ time steps for an input of size $n = 1024$. An algorithm with running time $\mathbf{T}(n) = n \log n$ requires $1024 \times 10 = 10,240$ time steps for an input of size $n = 1024$, which is an improvement of much more than a factor of ten when compared to the algorithm with running time $\mathbf{T}(n) = n^2$. Because $n^2 > 10n \log n$ whenever $n > 58$, if the typical problem size is larger than 58 for this example, then you would be much better off changing algorithms instead of buying a computer ten times faster. Furthermore, when you do buy a faster computer, an algorithm with a slower growth rate provides a greater benefit in terms of larger problem size that can run in a certain time on the new computer.

3.4 Asymptotic Analysis

Despite the larger constant for the curve labeled $10n$ in Figure 3.1, $2n^2$ crosses it at the relatively small value of $n = 5$. What if we double the value of the constant in front of the linear equation? As shown in the graph, $20n$ is surpassed by $2n^2$ once $n = 10$. The additional factor of two for the linear growth rate does not much matter. It only doubles the x-coordinate for the intersection point. In general, changes to a constant factor in either equation only shift *where* the two curves cross, not *whether* the two curves cross.

When you buy a faster computer or a faster compiler, the new problem size that can be run in a given amount of time for a given growth rate is larger by the same factor, regardless of the constant on the running-time equation. The time curves for two algorithms with different growth rates still cross, regardless of their running-time equation constants. For these reasons, we usually ignore the constants when we want an estimate of the growth rate for the running time or other resource requirements of an algorithm. This simplifies the analysis and keeps us thinking about the most important aspect: the growth rate. This is called **asymptotic algorithm analysis**. To be precise, asymptotic analysis refers to the study of an algorithm as the input size "gets big" or reaches a limit (in the calculus sense). However, it has proved to be so useful to ignore all constant factors that asymptotic analysis is used for most algorithm comparisons.

It is not always reasonable to ignore the constants. When comparing algorithms meant to run on small values of n, the constant can have a large effect. For example, if the problem is to sort a collection of exactly five records, then an algorithm designed for sorting thousands of records is probably not appropriate, even if its asymptotic analysis indicates good performance. There are rare cases where the constants for two algorithms under comparison can differ by a factor of 1000 or more, making the one with lower growth rate impractical for most purposes due to its large constant. Asymptotic analysis is a form of "back of the envelope" estimation for algorithm resource consumption. It provides a simplified model of the running time or other resource needs of an algorithm. This simplification usually helps you understand the behavior of your algorithms. Just be aware of the limitations to asymptotic analysis in the rare situation where the constant is important.

3.4.1 Upper Bounds

Several terms are used to describe the running-time equation for an algorithm. These terms — and their associated symbols — indicate precisely what aspect of the algorithm's behavior is being described. One is the **upper bound** for the growth of the algorithm's running time. It indicates the upper or highest growth rate that the algorithm can have.

Because the phrase "has an upper bound to its growth rate of $f(n)$" is long and often used when discussing algorithms, we adopt a special notation, called **big-Oh notation**. If the upper bound for an algorithm's growth rate (for, say, the worst case) is $f(n)$, then we would write that this algorithm is "in the set $\mathrm{O}(f(n))$in the worst case" (or just "in $\mathrm{O}(f(n))$in the worst case"). For example, if n^2 grows as fast as $\mathbf{T}(n)$ (the running time of our algorithm) for the worst-case input, we would say the algorithm is "in $\mathrm{O}(n^2)$ in the worst case."

The following is a precise definition for an upper bound. $\mathbf{T}(n)$ represents the true running time of the algorithm. $f(n)$ is some expression for the upper bound.

> For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $\mathrm{O}(f(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \leq cf(n)$ for all $n > n_0$.

Constant n_0 is the smallest value of n for which the claim of an upper bound holds true. Usually n_0 is small, such as 1, but does not need to be. You must also be able to pick some constant c, but it is irrelevant what the value for c actually is. In other words, the definition says that for *all* inputs of the type in question (such as the worst case for all inputs of size n) that are large enough (i.e., $n > n_0$), the algorithm *always* executes in less than $cf(n)$ steps for some constant c.

Example 3.4 Consider the sequential search algorithm for finding a specified value in an array of integers. If visiting and examining one value in the array requires c_s steps where c_s is a positive number, and if the value we search for has equal probability of appearing in any position in the array, then in the average case $\mathbf{T}(n) = c_s n/2$. For all values of $n > 1$, $c_s n/2 \leq c_s n$. Therefore, by the definition, $\mathbf{T}(n)$ is in $\mathrm{O}(n)$ for $n_0 = 1$ and $c = c_s$.

Example 3.5 For a particular algorithm, $\mathbf{T}(n) = c_1 n^2 + c_2 n$ in the average case where c_1 and c_2 are positive numbers. Then, $c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$ for all $n > 1$. So, $\mathbf{T}(n) \leq cn^2$ for $c = c_1 + c_2$, and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\mathrm{O}(n^2)$ by the second definition.

Example 3.6 Assigning the value from the first position of an array to a variable takes constant time regardless of the size of the array. Thus, $\mathbf{T}(n) = c$ (for the best, worst, and average cases). We could say in this case that $\mathbf{T}(n)$ is in $\mathrm{O}(c)$. However, it is traditional to say that an algorithm whose running time has a constant upper bound is in $\mathrm{O}(1)$.

If someone asked you out of the blue "Who is the best?" your natural reaction should be to reply "Best at what?" In the same way, if you are asked "What is the growth rate of this algorithm," you would need to ask "When? Best case? Average case? Or worst case?" Some algorithms have the same behavior no matter which input instance they receive. An example is finding the maximum in an array of integers. But for many algorithms, it makes a big difference, such as when searching an unsorted array for a particular value. So any statement about the upper bound of an algorithm must be in the context of some class of inputs of size n. We measure this upper bound nearly always on the best-case, average-case, or worst-case inputs. Thus, we cannot say, "this algorithm has an upper bound to its growth rate of n^2." We must say something like, "this algorithm has an upper bound to its growth rate of n^2 *in the average case*."

Knowing that something is in $\mathrm{O}(f(n))$ says only how bad things can be. Perhaps things are not nearly so bad. Because sequential search is in $\mathrm{O}(n)$ in the worst case, it is also true to say that sequential search is in $\mathrm{O}(n^2)$. But sequential search is practical for large n, in a way that is not true for some other algorithms in $\mathrm{O}(n^2)$. We always seek to define the running time of an algorithm with the tightest (lowest) possible upper bound. Thus, we prefer to say that sequential search is in $\mathrm{O}(n)$. This also explains why the phrase "is in $\mathrm{O}(f(n))$" or the notation "$\in \mathrm{O}(f(n))$" is used instead of "is $\mathrm{O}(f(n))$" or "$= \mathrm{O}(f(n))$." There is no strict equality to the use of big-Oh notation. $\mathrm{O}(n)$ is in $\mathrm{O}(n^2)$, but $\mathrm{O}(n^2)$ is not in $\mathrm{O}(n)$.

3.4.2 Lower Bounds

Big-Oh notation describes an upper bound. In other words, big-Oh notation states a claim about the greatest amount of some resource (usually time) that is required by an algorithm for some class of inputs of size n (typically the worst such input, the average of all possible inputs, or the best such input).

Similar notation is used to describe the least amount of a resource that an algorithm needs for some class of input. Like big-Oh notation, this is a measure of the algorithm's growth rate. Like big-Oh notation, it works for any resource, but we most often measure the least amount of time required. And again, like big-Oh notation, we are measuring the resource required for some particular class of inputs: the worst-, average-, or best-case input of size n.

The lower bound for an algorithm (or a problem, as explained later) is denoted by the symbol Ω, pronounced "big-Omega" or just "Omega." The following definition for Ω is symmetric with the definition of big-Oh.

> For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \geq cg(n)$ for all $n > n_0$.[1]

[1] An alternate (non-equivalent) definition for Ω is

Example 3.7 Assume $\mathbf{T}(n) = c_1n^2 + c_2n$ for c_1 and $c_2 > 0$. Then,

$$c_1n^2 + c_2n \geq c_1n^2$$

for all $n > 1$. So, $\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

It is also true that the equation of Example 3.7 is in $\Omega(n)$. However, as with big-Oh notation, we wish to get the "tightest" (for Ω notation, the largest) bound possible. Thus, we prefer to say that this running time is in $\Omega(n^2)$.

Recall the sequential search algorithm to find a value K within an array of integers. In the average and worst cases this algorithm is in $\Omega(n)$, because in both the average and worst cases we must examine *at least* cn values (where c is $1/2$ in the average case and 1 in the worst case).

3.4.3 Θ Notation

The definitions for big-Oh and Ω give us ways to describe the upper bound for an algorithm (if we can find an equation for the maximum cost of a particular class of inputs of size n) and the lower bound for an algorithm (if we can find an equation for the minimum cost for a particular class of inputs of size n). When the upper and lower bounds are the same within a constant factor, we indicate this by using Θ (big-Theta) notation. An algorithm is said to be $\Theta(h(n))$ if it is in $\mathrm{O}(h(n))$ *and*

$\mathbf{T}(n)$ is in the set $\Omega(g(n))$ if there exists a positive constant c such that $\mathbf{T}(n) \geq cg(n)$ for an infinite number of values for n.

This definition says that for an "interesting" number of cases, the algorithm takes at least $cg(n)$ time. Note that this definition is *not* symmetric with the definition of big-Oh. For $g(n)$ to be a lower bound, this definition *does not* require that $\mathbf{T}(n) \geq cg(n)$ for all values of n greater than some constant. It only requires that this happen often enough, in particular that it happen for an infinite number of values for n. Motivation for this alternate definition can be found in the following example.

Assume a particular algorithm has the following behavior:

$$\mathbf{T}(n) = \begin{cases} n & \text{for all odd } n \geq 1 \\ n^2/100 & \text{for all even } n \geq 0 \end{cases}$$

From this definition, $n^2/100 \geq \frac{1}{100}n^2$ for all even $n \geq 0$. So, $\mathbf{T}(n) \geq cn^2$ for an infinite number of values of n (i.e., for all even n) for $c = 1/100$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

For this equation for $\mathbf{T}(n)$, it is true that all inputs of size n take at least cn time. But an infinite number of inputs of size n take cn^2 time, so we would like to say that the algorithm is in $\Omega(n^2)$. Unfortunately, using our first definition will yield a lower bound of $\Omega(n)$ because it is not possible to pick constants c and n_0 such that $\mathbf{T}(n) \geq cn^2$ for all $n > n_0$. The alternative definition does result in a lower bound of $\Omega(n^2)$ for this algorithm, which seems to fit common sense more closely. Fortunately, few real algorithms or computer programs display the pathological behavior of this example. Our first definition for Ω generally yields the expected result.

As you can see from this discussion, asymptotic bounds notation is not a law of nature. It is merely a powerful modeling tool used to describe the behavior of algorithms.

it is in $\Omega(h(n))$. Note that we drop the word "in" for Θ notation, because there is a strict equality for two equations with the same Θ. In other words, if $f(n)$ is $\Theta(g(n))$, then $g(n)$ is $\Theta(f(n))$.

Because the sequential search algorithm is both in $\mathrm{O}(n)$ and in $\Omega(n)$ in the average case, we say it is $\Theta(n)$ in the average case.

Given an algebraic equation describing the time requirement for an algorithm, the upper and lower bounds always meet. That is because in some sense we have a perfect analysis for the algorithm, embodied by the running-time equation. For many algorithms (or their instantiations as programs), it is easy to come up with the equation that defines their runtime behavior. Most algorithms presented in this book are well understood and we can almost always give a Θ analysis for them. However, Chapter 17 discusses a whole class of algorithms for which we have no Θ analysis, just some unsatisfying big-Oh and Ω analyses. Exercise 3.14 presents a short, simple program fragment for which nobody currently knows the true upper or lower bounds.

While some textbooks and programmers will casually say that an algorithm is "order of" or "big-Oh" of some cost function, it is generally better to use Θ notation rather than big-Oh notation whenever we have sufficient knowledge about an algorithm to be sure that the upper and lower bounds indeed match. Throughout this book, Θ notation will be used in preference to big-Oh notation whenever our state of knowledge makes that possible. Limitations on our ability to analyze certain algorithms may require use of big-Oh or Ω notations. In rare occasions when the discussion is explicitly about the upper or lower bound of a problem or algorithm, the corresponding notation will be used in preference to Θ notation.

3.4.4 Simplifying Rules

Once you determine the running-time equation for an algorithm, it really is a simple matter to derive the big-Oh, Ω, and Θ expressions from the equation. You do not need to resort to the formal definitions of asymptotic analysis. Instead, you can use the following rules to determine the simplest form.

1. If $f(n)$ is in $\mathrm{O}(g(n))$ and $g(n)$ is in $\mathrm{O}(h(n))$, then $f(n)$ is in $\mathrm{O}(h(n))$.
2. If $f(n)$ is in $\mathrm{O}(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $\mathrm{O}(g(n))$.
3. If $f_1(n)$ is in $\mathrm{O}(g_1(n))$ and $f_2(n)$ is in $\mathrm{O}(g_2(n))$, then $f_1(n) + f_2(n)$ is in $\mathrm{O}(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $\mathrm{O}(g_1(n))$ and $f_2(n)$ is in $\mathrm{O}(g_2(n))$, then $f_1(n)f_2(n)$ is in $\mathrm{O}(g_1(n)g_2(n))$.

The first rule says that if some function $g(n)$ is an upper bound for your cost function, then any upper bound for $g(n)$ is also an upper bound for your cost function. A similar property holds true for Ω notation: If $g(n)$ is a lower bound for your

cost function, then any lower bound for $g(n)$ is also a lower bound for your cost function. Likewise for Θ notation.

The significance of rule (2) is that you can ignore any multiplicative constants in your equations when using big-Oh notation. This rule also holds true for Ω and Θ notations.

Rule (3) says that given two parts of a program run in sequence (whether two statements or two sections of code), you need consider only the more expensive part. This rule applies to Ω and Θ notations as well: For both, you need consider only the more expensive part.

Rule (4) is used to analyze simple loops in programs. If some action is repeated some number of times, and each repetition has the same cost, then the total cost is the cost of the action multiplied by the number of times that the action takes place. This rule applies to Ω and Θ notations as well.

Taking the first three rules collectively, you can ignore all constants and all lower-order terms to determine the asymptotic growth rate for any cost function. The advantages and dangers of ignoring constants were discussed near the beginning of this section. Ignoring lower-order terms is reasonable when performing an asymptotic analysis. The higher-order terms soon swamp the lower-order terms in their contribution to the total cost as n becomes larger. Thus, if $\mathbf{T}(n) = 3n^4 + 5n^2$, then $\mathbf{T}(n)$ is in $\mathrm{O}(n^4)$. The n^2 term contributes relatively little to the total cost for large n.

Throughout the rest of this book, these simplifying rules are used when discussing the cost for a program or algorithm.

3.4.5 Classifying Functions

Given functions $f(n)$ and $g(n)$ whose growth rates are expressed as algebraic equations, we might like to determine if one grows faster than the other. The best way to do this is to take the limit of the two functions as n grows towards infinity,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}.$$

If the limit goes to ∞, then $f(n)$ is in $\Omega(g(n))$ because $f(n)$ grows faster. If the limit goes to zero, then $f(n)$ is in $\mathrm{O}(g(n))$ because $g(n)$ grows faster. If the limit goes to some constant other than zero, then $f(n) = \Theta(g(n))$ because both grow at the same rate.

Example 3.8 If $f(n) = 2n \log n$ and $g(n) = n^2$, is $f(n)$ in $\mathrm{O}(g(n))$, $\Omega(g(n))$, or $\Theta(g(n))$? Because

$$\frac{n^2}{2n \log n} = \frac{n}{2 \log n},$$

we easily see that

$$\lim_{n\to\infty} \frac{n^2}{2n\log n} = \infty$$

because n grows faster than $2\log n$. Thus, n^2 is in $\Omega(2n\log n)$.

3.5 Calculating the Running Time for a Program

This section presents the analysis for several simple code fragments.

Example 3.9 We begin with an analysis of a simple assignment to an integer variable.

```
a = b;
```

Because the assignment statement takes constant time, it is $\Theta(1)$.

Example 3.10 Consider a simple **for** loop.

```
sum = 0;
for (i=1; i<=n; i++)
   sum += n;
```

The first line is $\Theta(1)$. The **for** loop is repeated n times. The third line takes constant time so, by simplifying rule (4) of Section 3.4.4, the total cost for executing the two lines making up the **for** loop is $\Theta(n)$. By rule (3), the cost of the entire code fragment is also $\Theta(n)$.

Example 3.11 We now analyze a code fragment with several **for** loops, some of which are nested.

```
sum = 0;
for (i=1; i<=n; i++)      // First for loop
   for (j=1; j<=i; j++)   //   is a double loop
      sum++;
for (k=0; k<n; k++)       // Second for loop
   A[k] = k;
```

This code fragment has three separate statements: the first assignment statement and the two **for** loops. Again the assignment statement takes constant time; call it c_1. The second **for** loop is just like the one in Example 3.10 and takes $c_2 n = \Theta(n)$ time.

The first **for** loop is a double loop and requires a special technique. We work from the inside of the loop outward. The expression **sum++** requires constant time; call it c_3. Because the inner **for** loop is executed i times, by

simplifying rule (4) it has cost $c_3 i$. The outer **for** loop is executed n times, but each time the cost of the inner loop is different because it costs $c_3 i$ with i changing each time. You should see that for the first execution of the outer loop, i is 1. For the second execution of the outer loop, i is 2. Each time through the outer loop, i becomes one greater, until the last time through the loop when $i = n$. Thus, the total cost of the loop is c_3 times the sum of the integers 1 through n. From Equation 2.1, we know that

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

which is $\Theta(n^2)$. By simplifying rule (3), $\Theta(c_1 + c_2 n + c_3 n^2)$ is simply $\Theta(n^2)$.

Example 3.12 Compare the asymptotic analysis for the following two code fragments:

```
sum1 = 0;
for (i=1; i<=n; i++)      // First double loop
   for (j=1; j<=n; j++)   //   do n times
      sum1++;

sum2 = 0;
for (i=1; i<=n; i++)      // Second double loop
   for (j=1; j<=i; j++)   //   do i times
      sum2++;
```

In the first double loop, the inner **for** loop always executes n times. Because the outer loop executes n times, it should be obvious that the statement **sum1++** is executed precisely n^2 times. The second loop is similar to the one analyzed in the previous example, with cost $\sum_{j=1}^{n} j$. This is approximately $\frac{1}{2}n^2$. Thus, both double loops cost $\Theta(n^2)$, though the second requires about half the time of the first.

Example 3.13 Not all doubly nested **for** loops are $\Theta(n^2)$. The following pair of nested loops illustrates this fact.

```
sum1 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
   for (j=1; j<=n; j++)   // Do n times
      sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)     // Do log n times
   for (j=1; j<=k; j++)   // Do k times
      sum2++;
```

When analyzing these two code fragments, we will assume that n is a power of two. The first code fragment has its outer **for** loop executed $\log n + 1$ times because on each iteration k is multiplied by two until it reaches n. Because the inner loop always executes n times, the total cost for the first code fragment can be expressed as $\sum_{i=0}^{\log n} n$. Note that a variable substitution takes place here to create the summation, with $k = 2^i$. From Equation 2.3, the solution for this summation is $\Theta(n \log n)$. In the second code fragment, the outer loop is also executed $\log n + 1$ times. The inner loop has cost k, which doubles each time. The summation can be expressed as $\sum_{i=0}^{\log n} 2^i$ where n is assumed to be a power of two and again $k = 2^i$. From Equation 2.8, we know that this summation is simply $\Theta(n)$.

What about other control statements? **While** loops are analyzed in a manner similar to **for** loops. The cost of an **if** statement in the worst case is the greater of the costs for the **then** and **else** clauses. This is also true for the average case, assuming that the size of n does not affect the probability of executing one of the clauses (which is usually, but not necessarily, true). For **switch** statements, the worst-case cost is that of the most expensive branch. For subroutine calls, simply add the cost of executing the subroutine.

There are rare situations in which the probability for executing the various branches of an **if** or **switch** statement are functions of the input size. For example, for input of size n, the **then** clause of an **if** statement might be executed with probability $1/n$. An example would be an **if** statement that executes the **then** clause only for the smallest of n values. To perform an average-case analysis for such programs, we cannot simply count the cost of the **if** statement as being the cost of the more expensive branch. In such situations, the technique of amortized analysis (see Section 14.3) can come to the rescue.

Determining the execution time of a recursive subroutine can be difficult. The running time for a recursive subroutine is typically best expressed by a recurrence relation. For example, the recursive factorial function **fact** of Section 2.5 calls itself with a value one less than its input value. The result of this recursive call is then multiplied by the input value, which takes constant time. Thus, the cost of the factorial function, if we wish to measure cost in terms of the number of multiplication operations, is one more than the number of multiplications made by the recursive call on the smaller input. Because the base case does no multiplications, its cost is zero. Thus, the running time for this function can be expressed as

$$\mathbf{T}(n) = \mathbf{T}(n-1) + 1 \text{ for } n > 1; \;\; T(1) = 0.$$

We know from Examples 2.8 and 2.13 that the closed-form solution for this recurrence relation is $\Theta(n)$.

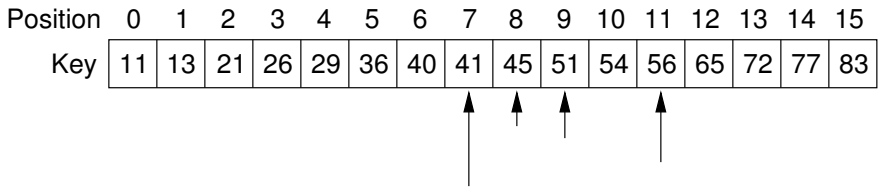


Figure 3.4 An illustration of binary search on a sorted array of 16 positions. Consider a search for the position with value $K = 45$. Binary search first checks the value at position 7. Because $41 < K$, the desired value cannot appear in any position below 7 in the array. Next, binary search checks the value at position 11. Because $56 > K$, the desired value (if it exists) must be between positions 7 and 11. Position 9 is checked next. Again, its value is too great. The final search is at position 8, which contains the desired value. Thus, function **`binary`** returns position 8. Alternatively, if K were 44, then the same series of record accesses would be made. After checking position 8, **`binary`** would return a value of n, indicating that the search is unsuccessful.

The final example of algorithm analysis for this section will compare two algorithms for performing search in an array. Earlier, we determined that the running time for sequential search on an array where the search value K is equally likely to appear in any location is $\Theta(n)$ in both the average and worst cases. We would like to compare this running time to that required to perform a **binary search** on an array whose values are stored in order from lowest to highest.

Binary search begins by examining the value in the middle position of the array; call this position mid and the corresponding value k_{mid}. If $k_{mid} = K$, then processing can stop immediately. This is unlikely to be the case, however. Fortunately, knowing the middle value provides useful information that can help guide the search process. In particular, if $k_{mid} > K$, then you know that the value K cannot appear in the array at any position greater than mid. Thus, you can eliminate future search in the upper half of the array. Conversely, if $k_{mid} < K$, then you know that you can ignore all positions in the array less than mid. Either way, half of the positions are eliminated from further consideration. Binary search next looks at the middle position in that part of the array where value K may exist. The value at this position again allows us to eliminate half of the remaining positions from consideration. This process repeats until either the desired value is found, or there are no positions remaining in the array that might contain the value K. Figure 3.4 illustrates the binary search method. Figure 3.5 shows an implementation for binary search.

To find the cost of this algorithm in the worst case, we can model the running time as a recurrence and then find the closed-form solution. Each recursive call to **`binary`** cuts the size of the array approximately in half, so we can model the worst-case cost as follows, assuming for simplicity that n is a power of two.

$$\mathbf{T}(n) = \mathbf{T}(n/2) + 1 \text{ for } n > 1; \quad \mathbf{T}(1) = 1.$$

```
// Return the position of an element in sorted array "A" of
// size "n" with value "K".  If "K" is not in "A", return
// the value "n".
int binary(int A[], int n, int K) {
  int l = -1;
  int r = n;           // l and r are beyond array bounds
  while (l+1 != r) {  // Stop when l and r meet
    int i = (l+r)/2;  // Check middle of remaining subarray
    if (K < A[i]) r = i;      // In left half
    if (K == A[i]) return i; // Found it
    if (K > A[i]) l = i;      // In right half
  }
  return n; // Search value not in A
}
```

Figure 3.5 Implementation for binary search.

If we expand the recurrence, we find that we can do so only $\log n$ times before we reach the base case, and each expansion adds one to the cost. Thus, the closed-form solution for the recurrence is $\mathbf{T}(n) = \log n$.

Function **binary** is designed to find the (single) occurrence of K and return its position. A special value is returned if K does not appear in the array. This algorithm can be modified to implement variations such as returning the position of the first occurrence of K in the array if multiple occurrences are allowed, and returning the position of the greatest value less than K when K is not in the array.

Comparing sequential search to binary search, we see that as n grows, the $\Theta(n)$ running time for sequential search in the average and worst cases quickly becomes much greater than the $\Theta(\log n)$ running time for binary search. Taken in isolation, binary search appears to be much more efficient than sequential search. This is despite the fact that the constant factor for binary search is greater than that for sequential search, because the calculation for the next search position in binary search is more expensive than just incrementing the current position, as sequential search does.

Note however that the running time for sequential search will be roughly the same regardless of whether or not the array values are stored in order. In contrast, binary search requires that the array values be ordered from lowest to highest. Depending on the context in which binary search is to be used, this requirement for a sorted array could be detrimental to the running time of a complete program, because maintaining the values in sorted order requires to greater cost when inserting new elements into the array. This is an example of a tradeoff between the advantage of binary search during search and the disadvantage related to maintaining a sorted array. Only in the context of the complete problem to be solved can we know whether the advantage outweighs the disadvantage.

3.6 Analyzing Problems

You most often use the techniques of "algorithm" analysis to analyze an algorithm, or the instantiation of an algorithm as a program. You can also use these same techniques to analyze the cost of a problem. It should make sense to you to say that the upper bound for a problem cannot be worse than the upper bound for the best algorithm that we know for that problem. But what does it mean to give a lower bound for a problem?

Consider a graph of cost over all inputs of a given size n for some algorithm for a given problem. Define $\mathcal{A}$ to be the collection of all algorithms that solve the problem (theoretically, there are an infinite number of such algorithms). Now, consider the collection of all the graphs for all of the (infinitely many) algorithms in $\mathcal{A}$. The worst case lower bound is the *least* of all the *highest* points on all the graphs.

It is much easier to show that an algorithm (or program) is in $\Omega(f(n))$ than it is to show that a problem is in $\Omega(f(n))$. For a problem to be in $\Omega(f(n))$ means that *every* algorithm that solves the problem is in $\Omega(f(n))$, even algorithms that we have not thought of!

So far all of our examples of algorithm analysis give "obvious" results, with big-Oh always matching Ω. To understand how big-Oh, Ω, and Θ notations are properly used to describe our understanding of a problem or an algorithm, it is best to consider an example where you do not already know a lot about the problem.

Let us look ahead to analyzing the problem of sorting to see how this process works. What is the least possible cost for any sorting algorithm in the worst case? The algorithm must at least look at every element in the input, just to determine that the input is truly sorted. Thus, any sorting algorithm must take at least cn time. For many problems, this observation that each of the n inputs must be looked at leads to an easy $\Omega(n)$ lower bound.

In your previous study of computer science, you have probably seen an example of a sorting algorithm whose running time is in $\mathrm{O}(n^2)$ in the worst case. The simple Bubble Sort and Insertion Sort algorithms typically given as examples in a first year programming course have worst case running times in $\mathrm{O}(n^2)$. Thus, the problem of sorting can be said to have an upper bound in $\mathrm{O}(n^2)$. How do we close the gap between $\Omega(n)$ and $\mathrm{O}(n^2)$? Can there be a better sorting algorithm? If you can think of no algorithm whose worst-case growth rate is better than $\mathrm{O}(n^2)$, and if you have discovered no analysis technique to show that the least cost for the problem of sorting in the worst case is greater than $\Omega(n)$, then you cannot know for sure whether or not there is a better algorithm.

Chapter 7 presents sorting algorithms whose running time is in $\mathrm{O}(n \log n)$ for the worst case. This greatly narrows the gap. With this new knowledge, we now have a lower bound in $\Omega(n)$ and an upper bound in $\mathrm{O}(n \log n)$. Should we search

for a faster algorithm? Many have tried, without success. Fortunately (or perhaps unfortunately?), Chapter 7 also includes a proof that any sorting algorithm must have running time in $\Omega(n \log n)$ in the worst case.[2] This proof is one of the most important results in the field of algorithm analysis, and it means that no sorting algorithm can possibly run faster than $cn \log n$ for the worst-case input of size n. Thus, we can conclude that the problem of sorting is $\Theta(n \log n)$ in the worst case, because the upper and lower bounds have met.

Knowing the lower bound for a problem does not give you a good algorithm. But it does help you to know when to stop looking. If the lower bound for the problem matches the upper bound for the algorithm (within a constant factor), then we know that we can find an algorithm that is better only by a constant factor.

3.7 Common Misunderstandings

Asymptotic analysis is one of the most intellectually difficult topics that undergraduate computer science majors are confronted with. Most people find growth rates and asymptotic analysis confusing and so develop misconceptions about either the concepts or the terminology. It helps to know what the standard points of confusion are, in hopes of avoiding them.

One problem with differentiating the concepts of upper and lower bounds is that, for most algorithms that you will encounter, it is easy to recognize the true growth rate for that algorithm. Given complete knowledge about a cost function, the upper and lower bound for that cost function are always the same. Thus, the distinction between an upper and a lower bound is only worthwhile when you have incomplete knowledge about the thing being measured. If this distinction is still not clear, reread Section 3.6. We use Θ-notation to indicate that there is no meaningful difference between what we know about the growth rates of the upper and lower bound (which is usually the case for simple algorithms).

It is a common mistake to confuse the concepts of upper bound or lower bound on the one hand, and worst case or best case on the other. The best, worst, or average cases each give us a concrete input instance (or concrete set of instances) that we can apply to an algorithm description to get a cost measure. The upper and lower bounds describe our understanding of the *growth rate* for that cost measure. So to define the growth rate for an algorithm or problem, we need to determine what we are measuring (the best, worst, or average case) and also our description for what we know about the growth rate of that cost measure (big-Oh, Ω, or Θ).

The upper bound for an algorithm is not the same as the worst case for that algorithm for a given input of size n. What is being bounded is not the actual cost (which you can determine for a given value of n), but rather the *growth rate* for the

[2] While it is fortunate to know the truth, it is unfortunate that sorting is $\Theta(n \log n)$ rather than $\Theta(n)$!

cost. There cannot be a growth rate for a single point, such as a particular value of n. The growth *rate* applies to the *change* in cost as a *change* in input size occurs. Likewise, the lower bound is not the same as the best case for a given size n.

Another common misconception is thinking that the best case for an algorithm occurs when the input size is as small as possible, or that the worst case occurs when the input size is as large as possible. What is correct is that best- and worse-case instances exist for each possible size of input. That is, for all inputs of a given size, say i, one (or more) of the inputs of size i is the best and one (or more) of the inputs of size i is the worst. Often (but not always!), we can characterize the best input case for an arbitrary size, and we can characterize the worst input case for an arbitrary size. Ideally, we can determine the growth rate for the characterized best, worst, and average cases as the input size grows.

Example 3.14 What is the growth rate of the best case for sequential search? For any array of size n, the best case occurs when the value we are looking for appears in the first position of the array. This is true regardless of the size of the array. Thus, the best case (for arbitrary size n) occurs when the desired value is in the first of n positions, and its cost is 1. It is *not* correct to say that the best case occurs when $n = 1$.

Example 3.15 Imagine drawing a graph to show the cost of finding the maximum value among n values, as n grows. That is, the x axis would be n, and the y value would be the cost. Of course, this is a diagonal line going up to the right, as n increases (you might want to sketch this graph for yourself before reading further).

Now, imagine the graph showing the cost for *each* instance of the problem of finding the maximum value among (say) 20 elements in an array. The first position along the x axis of the graph might correspond to having the maximum element in the first position of the array. The second position along the x axis of the graph might correspond to having the maximum element in the second position of the array, and so on. Of course, the cost is always 20. Therefore, the graph would be a horizontal line with value 20. You should sketch this graph for yourself.

Now, let us switch to the problem of doing a sequential search for a given value in an array. Think about the graph showing all the problem instances of size 20. The first problem instance might be when the value we search for is in the first position of the array. This has cost 1. The second problem instance might be when the value we search for is in the second position of the array. This has cost 2. And so on. If we arrange the problem instances of size 20 from least expensive on the left to most expensive on

the right, we see that the graph forms a diagonal line from lower left (with value 0) to upper right (with value 20). Sketch this graph for yourself.

Finally, let us consider the cost for performing sequential search as the size of the array n gets bigger. What will this graph look like? Unfortunately, there's not one simple answer, as there was for finding the maximum value. The shape of this graph depends on whether we are considering the best case cost (that would be a horizontal line with value 1), the worst case cost (that would be a diagonal line with value i at position i along the x axis), or the average cost (that would be a a diagonal line with value $i/2$ at position i along the x axis). This is why we must always say that function $f(n)$ is in $\mathrm{O}(g(n))$ in the best, average, or worst case! If we leave off which class of inputs we are discussing, we cannot know which cost measure we are referring to for most algorithms.

3.8 Multiple Parameters

Sometimes the proper analysis for an algorithm requires multiple parameters to describe the cost. To illustrate the concept, consider an algorithm to compute the rank ordering for counts of all pixel values in a picture. Pictures are often represented by a two-dimensional array, and a pixel is one cell in the array. The value of a pixel is either the code value for the color, or a value for the intensity of the picture at that pixel. Assume that each pixel can take any integer value in the range 0 to $C-1$. The problem is to find the number of pixels of each color value and then sort the color values with respect to the number of times each value appears in the picture. Assume that the picture is a rectangle with P pixels. A pseudocode algorithm to solve the problem follows.

```
for (i=0; i<C; i++)   // Initialize count
   count[i] = 0;
for (i=0; i<P; i++)   // Look at all of the pixels
   count[value(i)]++; // Increment a pixel value count
sort(count, C);       // Sort pixel value counts
```

In this example, **count** is an array of size C that stores the number of pixels for each color value. Function **value(i)** returns the color value for pixel i.

The time for the first **for** loop (which initializes **count**) is based on the number of colors, C. The time for the second loop (which determines the number of pixels with each color) is $\Theta(P)$. The time for the final line, the call to **sort**, depends on the cost of the sorting algorithm used. From the discussion of Section 3.6, we can assume that the sorting algorithm has cost $\Theta(P \log P)$ if P items are sorted, thus yielding $\Theta(P \log P)$ as the total algorithm cost.

Is this a good representation for the cost of this algorithm? What is actually being sorted? It is not the pixels, but rather the colors. What if C is much smaller than P? Then the estimate of $\Theta(P \log P)$ is pessimistic, because much fewer than P items are being sorted. Instead, we should use P as our analysis variable for steps that look at each pixel, and C as our analysis variable for steps that look at colors. Then we get $\Theta(C)$ for the initialization loop, $\Theta(P)$ for the pixel count loop, and $\Theta(C \log C)$ for the sorting operation. This yields a total cost of $\Theta(P + C \log C)$.

Why can we not simply use the value of C for input size and say that the cost of the algorithm is $\Theta(C \log C)$? Because, C is typically much less than P. For example, a picture might have 1000×1000 pixels and a range of 256 possible colors. So, P is one million, which is much larger than $C \log C$. But, if P is smaller, or C larger (even if it is still less than P), then $C \log C$ can become the larger quantity. Thus, neither variable should be ignored.

3.9 Space Bounds

Besides time, space is the other computing resource that is commonly of concern to programmers. Just as computers have become much faster over the years, they have also received greater allotments of memory. Even so, the amount of available disk space or main memory can be significant constraints for algorithm designers.

The analysis techniques used to measure space requirements are similar to those used to measure time requirements. However, while time requirements are normally measured for an algorithm that manipulates a particular data structure, space requirements are normally determined for the data structure itself. The concepts of asymptotic analysis for growth rates on input size apply completely to measuring space requirements.

Example 3.16 What are the space requirements for an array of n integers? If each integer requires c bytes, then the array requires cn bytes, which is $\Theta(n)$.

Example 3.17 Imagine that we want to keep track of friendships between n people. We can do this with an array of size $n \times n$. Each row of the array represents the friends of an individual, with the columns indicating who has that individual as a friend. For example, if person j is a friend of person i, then we place a mark in column j of row i in the array. Likewise, we should also place a mark in column i of row j if we assume that friendship works both ways. For n people, the total size of the array is $\Theta(n^2)$.

A data structure's primary purpose is to store data in a way that allows efficient access to those data. To provide efficient access, it may be necessary to store additional information about where the data are within the data structure. For example, each node of a linked list must store a pointer to the next value on the list. All such information stored in addition to the actual data values is referred to as **overhead**. Ideally, overhead should be kept to a minimum while allowing maximum access. The need to maintain a balance between these opposing goals is what makes the study of data structures so interesting.

One important aspect of algorithm design is referred to as the **space/time tradeoff** principle. The space/time tradeoff principle says that one can often achieve a reduction in time if one is willing to sacrifice space or vice versa. Many programs can be modified to reduce storage requirements by "packing" or encoding information. "Unpacking" or decoding the information requires additional time. Thus, the resulting program uses less space but runs slower. Conversely, many programs can be modified to pre-store results or reorganize information to allow faster running time at the expense of greater storage requirements. Typically, such changes in time and space are both by a constant factor.

A classic example of a space/time tradeoff is the **lookup table**. A lookup table pre-stores the value of a function that would otherwise be computed each time it is needed. For example, 12! is the greatest value for the factorial function that can be stored in a 32-bit `int` variable. If you are writing a program that often computes factorials, it is likely to be much more time efficient to simply pre-compute and store the 12 values in a table. Whenever the program needs the value of $n!$ it can simply check the lookup table. (If $n > 12$, the value is too large to store as an `int` variable anyway.) Compared to the time required to compute factorials, it may be well worth the small amount of additional space needed to store the lookup table.

Lookup tables can also store approximations for an expensive function such as sine or cosine. If you compute this function only for exact degrees or are willing to approximate the answer with the value for the nearest degree, then a lookup table storing the computation for exact degrees can be used instead of repeatedly computing the sine function. Note that initially building the lookup table requires a certain amount of time. Your application must use the lookup table often enough to make this initialization worthwhile.

Another example of the space/time tradeoff is typical of what a programmer might encounter when trying to optimize space. Here is a simple code fragment for sorting an array of integers. We assume that this is a special case where there are n integers whose values are a permutation of the integers from 0 to $n - 1$. This is an example of a Binsort, which is discussed in Section 7.7. Binsort assigns each value to an array position corresponding to its value.

```
for (i=0; i<n; i++)
   B[A[i]] = A[i];
```

This is efficient and requires $\Theta(n)$ time. However, it also requires two arrays of size n. Next is a code fragment that places the permutation in order but does so within the same array (thus it is an example of an "in place" sort).

```
for (i=0; i<n; i++)
   while (A[i] != i)
      swap(A, i, A[i]);
```

Function **swap(A, i, j)** exchanges elements **i** and **j** in array **A**. It may not be obvious that the second code fragment actually sorts the array. To see that this does work, notice that each pass through the **for** loop will at least move the integer with value i to its correct position in the array, and that during this iteration, the value of **A[i]** must be greater than or equal to i. A total of at most n **swap** operations take place, because an integer cannot be moved out of its correct position once it has been placed there, and each swap operation places at least one integer in its correct position. Thus, this code fragment has cost $\Theta(n)$. However, it requires more time to run than the first code fragment. On my computer the second version takes nearly twice as long to run as the first, but it only requires half the space.

A second principle for the relationship between a program's space and time requirements applies to programs that process information stored on disk, as discussed in Chapter 8 and thereafter. Strangely enough, the disk-based space/time tradeoff principle is almost the reverse of the space/time tradeoff principle for programs using main memory.

The **disk-based space/time tradeoff** principle states that the smaller you can make your disk storage requirements, the faster your program will run. This is because the time to read information from disk is enormous compared to computation time, so almost any amount of additional computation needed to unpack the data is going to be less than the disk-reading time saved by reducing the storage requirements. Naturally this principle does not hold true in all cases, but it is good to keep in mind when designing programs that process information stored on disk.

3.10 Speeding Up Your Programs

In practice, there is not such a big difference in running time between an algorithm with growth rate $\Theta(n)$ and another with growth rate $\Theta(n \log n)$. There is, however, an enormous difference in running time between algorithms with growth rates of $\Theta(n \log n)$ and $\Theta(n^2)$. As you shall see during the course of your study of common data structures and algorithms, it is not unusual that a problem whose obvious solution requires $\Theta(n^2)$ time also has a solution requiring $\Theta(n \log n)$ time. Examples include sorting and searching, two of the most important computer problems.

Example 3.18 The following is a true story. A few years ago, one of my graduate students had a big problem. His thesis work involved several

intricate operations on a large database. He was now working on the final step. "Dr. Shaffer," he said, "I am running this program and it seems to be taking a long time." After examining the algorithm we realized that its running time was $\Theta(n^2)$, and that it would likely take one to two weeks to complete. Even if we could keep the computer running uninterrupted for that long, he was hoping to complete his thesis and graduate before then. Fortunately, we realized that there was a fairly easy way to convert the algorithm so that its running time was $\Theta(n \log n)$. By the next day he had modified the program. It ran in only a few hours, and he finished his thesis on time.

While not nearly so important as changing an algorithm to reduce its growth rate, "code tuning" can also lead to dramatic improvements in running time. Code tuning is the art of hand-optimizing a program to run faster or require less storage. For many programs, code tuning can reduce running time by a factor of ten, or cut the storage requirements by a factor of two or more. I once tuned a critical function in a program — without changing its basic algorithm — to achieve a factor of 200 speedup. To get this speedup, however, I did make major changes in the representation of the information, converting from a symbolic coding scheme to a numeric coding scheme on which I was able to do direct computation.

Here are some suggestions for ways to speed up your programs by code tuning. The most important thing to realize is that most statements in a program do not have much effect on the running time of that program. There are normally just a few key subroutines, possibly even key lines of code within the key subroutines, that account for most of the running time. There is little point to cutting in half the running time of a subroutine that accounts for only 1% of the total running time. Focus your attention on those parts of the program that have the most impact.

When tuning code, it is important to gather good timing statistics. Many compilers and operating systems include profilers and other special tools to help gather information on both time and space use. These are invaluable when trying to make a program more efficient, because they can tell you where to invest your effort.

A lot of code tuning is based on the principle of avoiding work rather than speeding up work. A common situation occurs when we can test for a condition that lets us skip some work. However, such a test is never completely free. Care must be taken that the cost of the test does not exceed the amount of work saved. While one test might be cheaper than the work potentially saved, the test must always be made and the work can be avoided only some fraction of the time.

Example 3.19 A common operation in computer graphics applications is to find which among a set of complex objects contains a given point in space. Many useful data structures and algorithms have been developed to

deal with variations of this problem. Most such implementations involve the following tuning step. Directly testing whether a given complex object contains the point in question is relatively expensive. Instead, we can screen for whether the point is contained within a **bounding box** for the object. The bounding box is simply the smallest rectangle (usually defined to have sides perpendicular to the x and y axes) that contains the object. If the point is not in the bounding box, then it cannot be in the object. If the point is in the bounding box, only then would we conduct the full comparison of the object versus the point. Note that if the point is outside the bounding box, we saved time because the bounding box test is cheaper than the comparison of the full object versus the point. But if the point is inside the bounding box, then that test is redundant because we still have to compare the point against the object. Typically the amount of work avoided by making this test is greater than the cost of making the test on every object.

Example 3.20 Section 7.2.3 presents a sorting algorithm named Selection Sort. The chief distinguishing characteristic of this algorithm is that it requires relatively few swaps of records stored in the array to be sorted. However, it sometimes performs an unnecessary swap operation where it tries to swap a record with itself. This work could be avoided by testing whether the two indices being swapped are the same. However, this event does not occurr often. Because the cost of the test is high enough compared to the work saved when the test is successful, adding the test typically will slow down the program rather than speed it up.

Be careful not to use tricks that make the program unreadable. Most code tuning is simply cleaning up a carelessly written program, not taking a clear program and adding tricks. In particular, you should develop an appreciation for the capabilities of modern compilers to make extremely good optimizations of expressions. "Optimization of expressions" here means a rearrangement of arithmetic or logical expressions to run more efficiently. Be careful not to damage the compiler's ability to do such optimizations for you in an effort to optimize the expression yourself. Always check that your "optimizations" really do improve the program by running the program before and after the change on a suitable benchmark set of input. Many times I have been wrong about the positive effects of code tuning in my own programs. Most often I am wrong when I try to optimize an expression. It is hard to do better than the compiler.

The greatest time and space improvements come from a better data structure or algorithm. The final thought for this section is

First tune the algorithm, then tune the code.

3.11 Empirical Analysis

This chapter has focused on asymptotic analysis. This is an analytic tool, whereby we model the key aspects of an algorithm to determine the growth rate of the algorithm as the input size grows. As pointed out previously, there are many limitations to this approach. These include the effects at small problem size, determining the finer distinctions between algorithms with the same growth rate, and the inherent difficulty of doing mathematical modeling for more complex problems.

An alternative to analytical approaches are empirical ones. The most obvious empirical approach is simply to run two competitors and see which performs better. In this way we might overcome the deficiencies of analytical approaches.

Be warned that comparative timing of programs is a difficult business, often subject to experimental errors arising from uncontrolled factors (system load, the language or compiler used, etc.). The most important point is not to be biased in favor of one of the programs. If you are biased, this is certain to be reflected in the timings. One look at competing software or hardware vendors' advertisements should convince you of this. The most common pitfall when writing two programs to compare their performance is that one receives more code-tuning effort than the other. As mentioned in Section 3.10, code tuning can often reduce running time by a factor of ten. If the running times for two programs differ by a constant factor regardless of input size (i.e., their growth rates are the same), then differences in code tuning might account for any difference in running time. Be suspicious of empirical comparisons in this situation.

Another approach to analysis is simulation. The idea of simulation is to model the problem with a computer program and then run it to get a result. In the context of algorithm analysis, simulation is distinct from empirical comparison of two competitors because the purpose of the simulation is to perform analysis that might otherwise be too difficult. A good example of this appears in Figure 9.10. This figure shows the cost for inserting or deleting a record from a hash table under two different assumptions for the policy used to find a free slot in the table. The y axes is the cost in number of hash table slots evaluated, and the x axes is the percentage of slots in the table that are full. The mathematical equations for these curves can be determined, but this is not so easy. A reasonable alternative is to write simple variations on hashing. By timing the cost of the program for various loading conditions, it is not difficult to construct a plot similar to Figure 9.10. The purpose of this analysis is not to determine which approach to hashing is most efficient, so we are not doing empirical comparison of hashing alternatives. Instead, the purpose is to analyze the proper loading factor that would be used in an efficient hashing system to balance time cost versus hash table size (space cost).

3.12 Further Reading

Pioneering works on algorithm analysis include *The Art of Computer Programming* by Donald E. Knuth [Knu97, Knu98], and *The Design and Analysis of Computer Algorithms* by Aho, Hopcroft, and Ullman [AHU74]. The alternate definition for Ω comes from [AHU83]. The use of the notation "$\mathbf{T}(n)$ is in $\mathrm{O}(f(n))$" rather than the more commonly used "$\mathbf{T}(n) = \mathrm{O}(f(n))$" I derive from Brassard and Bratley [BB96], though certainly this use predates them. A good book to read for further information on algorithm analysis techniques is *Compared to What?* by Gregory J.E. Rawlins [Raw92].

Bentley [Ben88] describes one problem in numerical analysis for which, between 1945 and 1988, the complexity of the best known algorithm had decreased from $\mathrm{O}(n^7)$ to $\mathrm{O}(n^3)$. For a problem of size $n = 64$, this is roughly equivalent to the speedup achieved from all advances in computer hardware during the same time period.

While the most important aspect of program efficiency is the algorithm, much improvement can be gained from efficient coding of a program. As cited by Frederick P. Brooks in *The Mythical Man-Month* [Bro95], an efficient programmer can often produce programs that run five times faster than an inefficient programmer, even when neither takes special efforts to speed up their code. For excellent and enjoyable essays on improving your coding efficiency, and ways to speed up your code when it really matters, see the books by Jon Bentley [Ben82, Ben00, Ben88]. The situation described in Example 3.18 arose when we were working on the project reported on in [SU92].

As an interesting aside, writing a correct binary search algorithm is not easy. Knuth [Knu98] notes that while the first binary search was published in 1946, the first bug-free algorithm was not published until 1962! Bentley ("Writing Correct Programs" in [Ben00]) has found that 90% of the computer professionals he tested could not write a bug-free binary search in two hours.

3.13 Exercises

3.1 For each of the six expressions of Figure 3.1, give the range of values of n for which that expression is most efficient.

3.2 Graph the following expressions. For each expression, state the range of values of n for which that expression is the most efficient.

$$4n^2 \qquad \log_3 n \qquad 3^n \qquad 20n \qquad 2 \qquad \log_2 n \qquad n^{2/3}$$

3.3 Arrange the following expressions by growth rate from slowest to fastest.

$$4n^2 \qquad \log_3 n \qquad n! \qquad 3^n \qquad 20n \qquad 2 \qquad \log_2 n \qquad n^{2/3}$$

See Stirling's approximation in Section 2.2 for help in classifying $n!$.

3.4 **(a)** Suppose that a particular algorithm has time complexity $\mathbf{T}(n) = 3 \times 2^n$, and that executing an implementation of it on a particular machine takes t seconds for n inputs. Now suppose that we are presented with a machine that is 64 times as fast. How many inputs could we process on the new machine in t seconds?

(b) Suppose that another algorithm has time complexity $\mathbf{T}(n) = n^2$, and that executing an implementation of it on a particular machine takes t seconds for n inputs. Now suppose that we are presented with a machine that is 64 times as fast. How many inputs could we process on the new machine in t seconds?

(c) A third algorithm has time complexity $\mathbf{T}(n) = 8n$. Executing an implementation of it on a particular machine takes t seconds for n inputs. Given a new machine that is 64 times as fast, how many inputs could we process in t seconds?

3.5 Hardware vendor XYZ Corp. claims that their latest computer will run 100 times faster than that of their competitor, Prunes, Inc. If the Prunes, Inc. computer can execute a program on input of size n in one hour, what size input can XYZ's computer execute in one hour for each algorithm with the following growth rate equations?

$$n \qquad n^2 \qquad n^3 \qquad 2^n$$

3.6 **(a)** Find a growth rate that squares the run time when we double the input size. That is, if $\mathbf{T}(n) = X$, then $\mathbf{T}(2n) = x^2$

(b) Find a growth rate that cubes the run time when we double the input size. That is, if $\mathbf{T}(n) = X$, then $\mathbf{T}(2n) = x^3$

3.7 Using the definition of big-Oh, show that 1 is in $\mathrm{O}(1)$ and that 1 is in $\mathrm{O}(n)$.

3.8 Using the definitions of big-Oh and Ω, find the upper and lower bounds for the following expressions. Be sure to state appropriate values for c and n_0.

(a) $c_1 n$

(b) $c_2 n^3 + c_3$

(c) $c_4 n \log n + c_5 n$

(d) $c_6 2^n + c_7 n^6$

3.9 **(a)** What is the smallest integer k such that $\sqrt{n} = O(n^k)$?

(b) What is the smallest integer k such that $n \log n = O(n^k)$?

3.10 **(a)** Is $2n = \Theta(3n)$? Explain why or why not.

(b) Is $2^n = \Theta(3^n)$? Explain why or why not.

3.11 For each of the following pairs of functions, either $f(n)$ is in $\mathrm{O}(g(n))$, $f(n)$ is in $\Omega(g(n))$, or $f(n) = \Theta(g(n))$. For each pair, determine which relationship is correct. Justify your answer, using the method of limits discussed in Section 3.4.5.

(a) $f(n) = \log n^2$; $g(n) = \log n + 5$.
(b) $f(n) = \sqrt{n}$; $g(n) = \log n^2$.
(c) $f(n) = \log^2 n$; $g(n) = \log n$.
(d) $f(n) = n$; $g(n) = log^2 n$.
(e) $f(n) = n \log n + n$; $g(n) = \log n$.
(f) $f(n) = \log n^2$; $g(n) = (\log n)^2$.
(g) $f(n) = 10$; $g(n) = \log 10$.
(h) $f(n) = 2^n$; $g(n) = 10n^2$.
(i) $f(n) = 2^n$; $g(n) = n \log n$.
(j) $f(n) = 2^n$; $g(n) = 3^n$.
(k) $f(n) = 2^n$; $g(n) = n^n$.

3.12 Determine Θ for the following code fragments in the average case. Assume that all variables are of type **int**.

(a)
```
a = b + c;
d = a + e;
```

(b)
```
sum = 0;
for (i=0; i<3; i++)
   for (j=0; j<n; j++)
      sum++;
```

(c)
```
sum=0;
for (i=0; i<n*n; i++)
   sum++;
```

(d)
```
for (i=0; i < n-1; i++)
  for (j=i+1; j < n; j++) {
    tmp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = tmp;
  }
```

(e)
```
sum = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j*=2)
    sum++;
```

(f)
```
sum = 0;
for (i=1; i<=n; i*=2)
  for (j=1; j<=n; j++)
    sum++;
```

(g) Assume that array **A** contains n values, **Random** takes constant time, and **sort** takes $n \log n$ steps.

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++)
    A[j] = Random(n);
  sort(A, n);
}
```

(h) Assume array **A** contains a random permutation of the values from 0 to $n-1$.

```
sum = 0;
for (i=0; i<n; i++)
  for (j=0; A[j]!=i; j++)
    sum++;
```

(i)

```
sum = 0;
if (EVEN(n))
  for (i=0; i<n; i++)
    sum++;
else
  sum = sum + n;
```

3.13 Show that big-Theta notation (Θ) defines an equivalence relation on the set of functions.

3.14 Give the best *lower* bound that you can for the following code fragment, as a function of the initial value of n.

```
while (n > 1)
  if (ODD(n))
    n = 3 * n + 1;
  else
    n = n / 2;
```

Do you think that the upper bound is likely to be the same as the answer you gave for the lower bound?

3.15 Does every algorithm have a Θ running-time equation? In other words, are the upper and lower bounds for the running time (on any specified class of inputs) always the same?

3.16 Does every problem for which there exists some algorithm have a Θ running-time equation? In other words, for every problem, and for any specified class of inputs, is there some algorithm whose upper bound is equal to the problem's lower bound?

3.17 Given an array storing integers ordered by value, modify the binary search routine to return the position of the first integer with value K in the situation where K can appear multiple times in the array. Be sure that your algorithm is $\Theta(\log n)$, that is, do *not* resort to sequential search once an occurrence of K is found.

3.18 Given an array storing integers ordered by value, modify the binary search routine to return the position of the integer with the greatest value less than K when K itself does not appear in the array. Return **ERROR** if the least value in the array is greater than K.

3.19 Modify the binary search routine to support search in an array of infinite size. In particular, you are given as input a sorted array and a key value K to search for. Call n the position of the smallest value in the array that

is equal to or larger than X. Provide an algorithm that can determine n in $O(\log n)$ comparisons in the worst case. Explain why your algorithm meets the required time bound.

3.20 It is possible to change the way that we pick the dividing point in a binary search, and still get a working search routine. However, where we pick the dividing point could affect the performance of the algorithm.

(a) If we change the dividing point computation in function **`binary`** from $i = (l + r)/2$ to $i = (l + ((r - l)/3))$, what will the worst-case running time be in asymptotic terms? If the difference is only a constant time factor, how much slower or faster will the modified program be compared to the original version of **`binary`**?

(b) If we change the dividing point computation in function **`binary`** from $i = (l + r)/2$ to $i = r - 2$, what will the worst-case running time be in asymptotic terms? If the difference is only a constant time factor, how much slower or faster will the modified program be compared to the original version of **`binary`**?

3.21 Design an algorithm to assemble a jigsaw puzzle. Assume that each piece has four sides, and that each piece's final orientation is known (top, bottom, etc.). Assume that you have available a function

```
bool compare(Piece a, Piece b, Side ad)
```

that can tell, in constant time, whether piece a connects to piece b on a's side ad and b's opposite side bd. The input to your algorithm should consist of an $n \times m$ array of random pieces, along with dimensions n and m. The algorithm should put the pieces in their correct positions in the array. Your algorithm should be as efficient as possible in the asymptotic sense. Write a summation for the running time of your algorithm on n pieces, and then derive a closed-form solution for the summation.

3.22 Can the average case cost for an algorithm be worse than the worst case cost? Can it be better than the best case cost? Explain why or why not.

3.23 Prove that if an algorithm is $\Theta(f(n))$ in the average case, then it is $\Omega(f(n))$ in the worst case.

3.24 Prove that if an algorithm is $\Theta(f(n))$ in the average case, then it is $O(f(n))$ in the best case.

3.14 Projects

3.1 Imagine that you are trying to store 32 Boolean values, and must access them frequently. Compare the time required to access Boolean values stored alternatively as a single bit field, a character, a short integer, or a long integer. There are two things to be careful of when writing your program. First, be

sure that your program does enough variable accesses to make meaningful measurements. A single access takes much less time than a single unit of measurement (typically milliseconds) for all four methods. Second, be sure that your program spends as much time as possible doing variable accesses rather than other things such as calling timing functions or incrementing **`for`** loop counters.

3.2 Implement sequential search and binary search algorithms on your computer. Run timings for each algorithm on arrays of size $n = 10^i$ for i ranging from 1 to as large a value as your computer's memory and compiler will allow. For both algorithms, store the values 0 through $n - 1$ in order in the array, and use a variety of random search values in the range 0 to $n - 1$ on each size n. Graph the resulting times. When is sequential search faster than binary search for a sorted array?

3.3 Implement a program that runs and gives timings for the two Fibonacci sequence functions provided in Exercise 2.11. Graph the resulting running times for as many values of n as your computer can handle.

PART II

Fundamental Data Structures

4

Lists, Stacks, and Queues

If your program needs to store a few things — numbers, payroll records, or job descriptions for example — the simplest and most effective approach might be to put them in a list. Only when you have to organize and search through a large number of things do more sophisticated data structures usually become necessary. (We will study how to organize and search through medium amounts of data in Chapters 5, 7, and 9, and discuss how to deal with large amounts of data in Chapters 8–10.) Many applications don't require any form of search, and they do not require that any ordering be placed on the objects being stored. Some applications require processing in a strict chronological order, processing objects in the order that they arrived, or perhaps processing objects in the reverse of the order that they arrived. For all these situations, a simple list structure is appropriate.

This chapter describes representations for lists in general, as well as two important list-like structures called the stack and the queue. Along with presenting these fundamental data structures, the other goals of the chapter are to: (1) Give examples of separating a logical representation in the form of an ADT from a physical implementation for a data structure. (2) Illustrate the use of asymptotic analysis in the context of some simple operations that you might already be familiar with. In this way you can begin to see how asymptotic analysis works, without the complications that arise when analyzing more sophisticated algorithms and data structures. (3) Introduce the concept and use of dictionaries.

We begin by defining an ADT for lists in Section 4.1. Two implementations for the list ADT — the array-based list and the linked list — are covered in detail and their relative merits discussed. Sections 4.2 and 4.3 cover stacks and queues, respectively. Sample implementations for each of these data structures are presented. Section 4.4 presents the Dictionary ADT for storing and retrieving data, which sets a context for implementing search structures such as the Binary Search Tree of Section 5.4.

4.1 Lists

We all have an intuitive understanding of what we mean by a "list." Our first step is to define precisely what is meant so that this intuitive understanding can eventually be converted into a concrete data structure and its operations. The most important concept related to lists is that of **position**. In other words, we perceive that there is a first element in the list, a second element, and so on. We should view a list as embodying the mathematical concepts of a sequence, as defined in Section 2.1.

We define a **list** to be a finite, ordered sequence of data items known as **elements**. "Ordered" in this definition means that each element has a position in the list. (We will not use "ordered" in this context to mean that the list elements are sorted by value.) Each list element has a data type. In the simple list implementations discussed in this chapter, all elements of the list have the same data type, although there is no conceptual objection to lists whose elements have differing data types if the application requires it (see Section 12.1). The operations defined as part of the list ADT do not depend on the elemental data type. For example, the list ADT can be used for lists of integers, lists of characters, lists of payroll records, even lists of lists.

A list is said to be **empty** when it contains no elements. The number of elements currently stored is called the **length** of the list. The beginning of the list is called the **head**, the end of the list is called the **tail**. There might or might not be some relationship between the value of an element and its position in the list. For example, **sorted lists** have their elements positioned in ascending order of value, while **unsorted lists** have no particular relationship between element values and positions. This section will consider only unsorted lists. Chapters 7 and 9 treat the problems of how to create and search sorted lists efficiently.

When presenting the contents of a list, we use the same notation as was introduced for sequences in Section 2.1. To be consistent with **C**++ array indexing, the first position on the list is denoted as 0. Thus, if there are n elements in the list, they are given positions 0 through $n-1$ as $\langle a_0,\ a_1,\ ...,\ a_{n-1}\rangle$. The subscript indicates an element's position within the list. Using this notation, the empty list would appear as $\langle\rangle$.

Before selecting a list implementation, a program designer should first consider what basic operations the implementation must support. Our common intuition about lists tells us that a list should be able to grow and shrink in size as we insert and remove elements. We should be able to insert and remove elements from anywhere in the list. We should be able to gain access to any element's value, either to read it or to change it. We must be able to create and clear (or reinitialize) lists. It is also convenient to access the next or previous element from the "current" one.

The next step is to define the ADT for a list object in terms of a set of operations on that object. We will use the **C**++ notation of anabstract class to formally define

the list ADT. An abstract class is one whose member functions are all declared to be "pure virtual" as indicated by the "**=0**" notation at the end of the member function declarations. Class **List** defines the member functions that any list implementation inheriting from it must support, along with their parameters and return types. We increase the flexibility of the list ADT by writing it as a **C**++ template.

True to the notion of an ADT, anabstract class does not specify how operations are implemented. Two complete implementations are presented later in this section, both of which use the same list ADT to define their operations, but they are considerably different in approaches and in their space/time tradeoffs.

Figure 4.1 presents our list ADT. Class **List** is a template of one parameter, named **E** for "element". **E** serves as a placeholder for whatever element type the user would like to store in a list. The comments given in Figure 4.1 describe precisely what each member function is intended to do. However, some explanation of the basic design is in order. Given that we wish to support the concept of a sequence, with access to any position in the list, the need for many of the member functions such as **insert** and **moveToPos** is clear. The key design decision embodied in this ADT is support for the concept of a **current position**. For example, member **moveToStart** sets the current position to be the first element on the list, while methods **next** and **prev** move the current position to the next and previous elements, respectively. The intention is that any implementation for this ADT support the concept of a current position. The current position is where any action such as insertion or deletion will take place.

Since insertions take place at the current position, and since we want to be able to insert to the front or the back of the list as well as anywhere in between, there are actually $n + 1$ possible "current positions" when there are n elements in the list.

It is helpful to modify our list display notation to show the position of the current element. I will use a vertical bar, such as $\langle 20,\ 23 \mid 12,\ 15\rangle$ to indicate the list of four elements, with the current position being to the right of the bar at element 12. Given this configuration, calling **insert** with value 10 will change the list to be $\langle 20,\ 23 \mid 10,\ 12,\ 15\rangle$.

If you examine Figure 4.1, you should find that the list member functions provided allow you to build a list with elements in any desired order, and to access any desired position in the list. You might notice that the **clear** method is not necessary, in that it could be implemented by means of the other member functions in the same asymptotic time. It is included merely for convenience.

Method **getValue** returns a pointer to the current element. It is considered a violation of **getValue**'s preconditions to ask for the value of a non-existent element (i.e., there must be something to the right of the vertical bar). In our concrete list implementations, assertions are used to enforce such preconditions. In a commercial implementation, such violations would be best implemented by the **C**++ exception mechanism.

```
template <typename E> class List { // List ADT
private:
  void operator =(const List&) {}      // Protect assignment
  List(const List&) {}           // Protect copy constructor
public:
  List() {}          // Default constructor
  virtual ~List() {} // Base destructor

  // Clear contents from the list, to make it empty.
  virtual void clear() = 0;

  // Insert an element at the current location.
  // item: The element to be inserted
  virtual void insert(const E& item) = 0;

  // Append an element at the end of the list.
  // item: The element to be appended.
  virtual void append(const E& item) = 0;

  // Remove and return the current element.
  // Return: the element that was removed.
  virtual E remove() = 0;

  // Set the current position to the start of the list
  virtual void moveToStart() = 0;

  // Set the current position to the end of the list
  virtual void moveToEnd() = 0;

  // Move the current position one step left. No change
  // if already at beginning.
  virtual void prev() = 0;

  // Move the current position one step right. No change
  // if already at end.
  virtual void next() = 0;

  // Return: The number of elements in the list.
  virtual int length() const = 0;

  // Return: The position of the current element.
  virtual int currPos() const = 0;

  // Set current position.
  // pos: The position to make current.
  virtual void moveToPos(int pos) = 0;

  // Return: The current element.
  virtual const E& getValue() const = 0;
};
```

Figure 4.1 The ADT for a list.

A list can be iterated through as shown in the following code fragment.

```
for (L.moveToStart(); L.currPos()<L.length(); L.next()) {
  it = L.getValue();
  doSomething(it);
}
```

In this example, each element of the list in turn is stored in **it**, and passed to the **doSomething** function. The loop terminates when the current position reaches the end of the list.

The declaration for abstract class **List** also makes private the class copy constructor and an overloading for the assignment operator. This protects the class from accidentally being copied. This is done in part to simplify the example code used in this book. A full-featured list implementation would likely support copying and assigning list objects.

The list class declaration presented here is just one of many possible interpretations for lists. Figure 4.1 provides most of the operations that one naturally expects to perform on lists and serves to illustrate the issues relevant to implementing the list data structure. As an example of using the list ADT, we can create a function to return **true** if there is an occurrence of a given integer in the list, and **false** otherwise. The **find** method needs no knowledge about the specific list implementation, just the list ADT.

```
// Return true if "K" is in list "L", false otherwise
bool find(List<int>& L, int K) {
  int it;
  for (L.moveToStart(); L.currPos()<L.length(); L.next()) {
    it = L.getValue();
    if (K == it) return true;  // Found K
  }
  return false;                // K not found
}
```

While this implementation for **find** could be written as a template with respect to the element type, it would still be limited in its ability to handle different data types stored on the list. In particular, it only works when the description for the object being searched for (**k** in the function) is of the same type as the objects themselves, and that can meaningfully be compared when using the **==** comparison operator. A more typical situation is that we are searching for a record that contains a key field who's value matches **k**. Similar functions to find and return a composite element based on a key value can be created using the list implementation, but to do so requires some agreement between the list ADT and the **find** function on the concept of a key, and on how keys may be compared. This topic will be discussed in Section 4.4.

4.1.1 Array-Based List Implementation

There are two standard approaches to implementing lists, the **array-based** list, and the **linked** list. This section discusses the array-based approach. The linked list is presented in Section 4.1.2. Time and space efficiency comparisons for the two are discussed in Section 4.1.3.

Figure 4.2 shows the array-based list implementation, named **`AList`**. **`AList`** inherits from abstract class **`List`** and so must implement all of the member functions of **`List`**.

Class **`AList`**'s private portion contains the data members for the array-based list. These include **`listArray`**, the array which holds the list elements. Because **`listArray`** must be allocated at some fixed size, the size of the array must be known when the list object is created. Note that an optional parameter is declared for the **`AList`** constructor. With this parameter, the user can indicate the maximum number of elements permitted in the list. The phrase "**`=defaultSize`**" indicates that the parameter is optional. If no parameter is given, then it takes the value **`defaultSize`**, which is assumed to be a suitably defined constant value.

Because each list can have a differently sized array, each list must remember its maximum permitted size. Data member **`maxSize`** serves this purpose. At any given time the list actually holds some number of elements that can be less than the maximum allowed by the array. This value is stored in **`listSize`**. Data member **`curr`** stores the current position. Because **`listArray`**, **`maxSize`**, **`listSize`**, and **`curr`** are all declared to be **`private`**, they may only be accessed by methods of Class **`AList`**.

Class **`AList`** stores the list elements in the first **`listSize`** contiguous array positions. Array positions correspond to list positions. In other words, the element at position i in the list is stored at array cell i. The head of the list is always at position 0. This makes random access to any element in the list quite easy. Given some position in the list, the value of the element in that position can be accessed directly. Thus, access to any element using the **`moveToPos`** method followed by the **`getValue`** method takes $\Theta(1)$ time.

Because the array-based list implementation is defined to store list elements in contiguous cells of the array, the **`insert`**, **`append`**, and **`remove`** methods must maintain this property. Inserting or removing elements at the tail of the list is easy, so the **`append`** operation takes $\Theta(1)$ time. But if we wish to insert an element at the head of the list, all elements currently in the list must shift one position toward the tail to make room, as illustrated by Figure 4.3. This process takes $\Theta(n)$ time if there are n elements already in the list. If we wish to insert at position i within a list of n elements, then $n - i$ elements must shift toward the tail. Removing an element from the head of the list is similar in that all remaining elements must shift toward the head by one position to fill in the gap. To remove the element at position

```
template <typename E> // Array-based list implementation
class AList : public List<E> {
private:
  int maxSize;        // Maximum size of list
  int listSize;       // Number of list items now
  int curr;           // Position of current element
  E* listArray;    // Array holding list elements

public:
  AList(int size=defaultSize) { // Constructor
    maxSize = size;
    listSize = curr = 0;
    listArray = new E[maxSize];
  }

  ~AList() { delete [] listArray; } // Destructor

  void clear() {                    // Reinitialize the list
    delete [] listArray;            // Remove the array
    listSize = curr = 0;            // Reset the size
    listArray = new E[maxSize];  // Recreate array
  }

  // Insert "it" at current position
  void insert(const E& it) {
    Assert(listSize < maxSize, "List capacity exceeded");
    for(int i=listSize; i>curr; i--)  // Shift elements up
      listArray[i] = listArray[i-1];  //   to make room
    listArray[curr] = it;
    listSize++;                       // Increment list size
  }

  void append(const E& it) {       // Append "it"
    Assert(listSize < maxSize, "List capacity exceeded");
    listArray[listSize++] = it;
  }

  // Remove and return the current element.
  E remove() {
    Assert((curr>=0) && (curr < listSize), "No element");
    E it = listArray[curr];           // Copy the element
    for(int i=curr; i<listSize-1; i++)  // Shift them down
      listArray[i] = listArray[i+1];
    listSize--;                           // Decrement size
    return it;
  }
```

Figure 4.2 An array-based list implementation.

```
  void moveToStart() { curr = 0; }        // Reset position
  void moveToEnd() { curr = listSize; }     // Set at end
  void prev() { if (curr != 0) curr--; }       // Back up
  void next() { if (curr < listSize) curr++; } // Next

  // Return list size
  int length() const  { return listSize; }

  // Return current position
  int currPos() const { return curr; }

  // Set current list position to "pos"
  void moveToPos(int pos) {
    Assert ((pos>=0)&&(pos<=listSize), "Pos out of range");
    curr = pos;
  }

  const E& getValue() const { // Return current element
    Assert((curr>=0)&&(curr<listSize),"No current element");
    return listArray[curr];
  }
};
```

Figure 4.2 (continued)

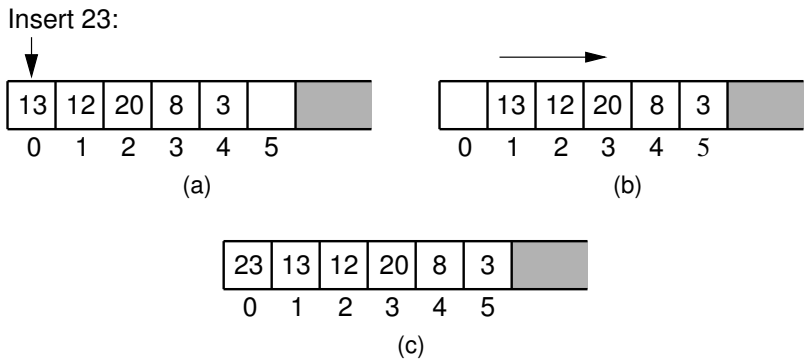


Figure 4.3 Inserting an element at the head of an array-based list requires shifting all existing elements in the array by one position toward the tail. (a) A list containing five elements before inserting an element with value 23. (b) The list after shifting all existing elements one position to the right. (c) The list after 23 has been inserted in array position 0. Shading indicates the unused part of the array.

i, $n - i - 1$ elements must shift toward the head. In the average case, insertion or removal requires moving half of the elements, which is $\Theta(n)$.

Most of the other member functions for Class **AList** simply access the current list element or move the current position. Such operations all require $\Theta(1)$ time. Aside from **insert** and **remove**, the only other operations that might require

```
// Singly linked list node
template <typename E> class Link {
public:
  E element;       // Value for this node
  Link *next;         // Pointer to next node in list
  // Constructors
  Link(const E& elemval, Link* nextval =NULL)
    { element = elemval;  next = nextval; }
  Link(Link* nextval =NULL) { next = nextval; }
};
```

Figure 4.4 A simple singly linked list node implementation.

more than constant time are the constructor, the destructor, and **clear**. These three member functions each make use of the system free-storeoperators **new** and **delete**. As discussed further in Section 4.1.2, system free-store operations can be expensive. In particular, the cost to delete **listArray** depends in part on the type of elements it stores, and whether the **delete** operator must call a destructor on each one.

4.1.2 Linked Lists

The second traditional approach to implementing lists makes use of pointers and is usually called a **linked list**. The linked list uses **dynamic memory allocation**, that is, it allocates memory for new list elements as needed.

A linked list is made up of a series of objects, called the **nodes** of the list. Because a list node is a distinct object (as opposed to simply a cell in an array), it is good practice to make a separate list node class. An additional benefit to creating a list node class is that it can be reused by the linked implementations for the stack and queue data structures presented later in this chapter. Figure 4.4 shows the implementation for list nodes, called the **Link** class. Objects in the **Link** class contain an **element** field to store the element value, and a **next** field to store a pointer to the next node on the list. The list built from such nodes is called a **singly linked list**, or a **one-way list**, because each list node has a single pointer to the next node on the list.

The **Link** class is quite simple. There are two forms for its constructor, one with an initial element value and one without. Because the **Link** class is also used by the stack and queue implementations presented later, its data members are made public. While technically this is breaking encapsulation, in practice the **Link** class should be implemented as a private class of the linked list (or stack or queue) implementation, and thus not visible to the rest of the program.

Figure 4.5(a) shows a graphical depiction for a linked list storing four integers. The value stored in a pointer variable is indicated by an arrow "pointing" to something. C++ uses the special symbol **NULL** for a pointer value that points nowhere, such as for the last list node's **next** field. A **NULL** pointer is indicated graphically

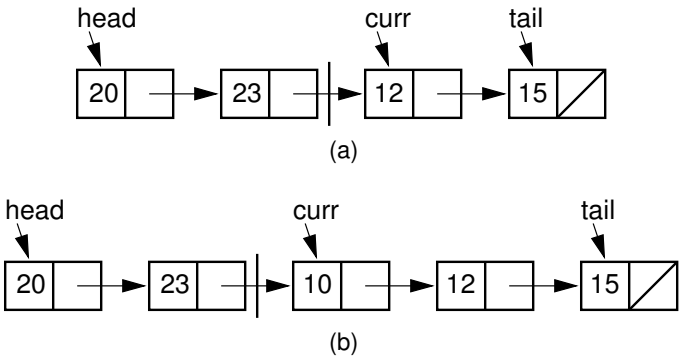


Figure 4.5 Illustration of a faulty linked-list implementation where **curr** points directly to the current node. (a) Linked list prior to inserting element with value 10. (b) Desired effect of inserting element with value 10.

by a diagonal slash through a pointer variable's box. The vertical line between the nodes labeled 23 and 12 in Figure 4.5(a) indicates the current position (immediately to the right of this line).

The list's first node is accessed from a pointer named **head**. To speed access to the end of the list, and to allow the **append** method to be performed in constant time, a pointer named **tail** is also kept to the last link of the list. The position of the current element is indicated by another pointer, named **curr**. Finally, because there is no simple way to compute the length of the list simply from these three pointers, the list length must be stored explicitly, and updated by every operation that modifies the list size. The value **cnt** stores the length of the list.

Class **LList** also includes private helper methods **init** and **removeall**. They are used by **LList**'s constructor, destructor, and **clear** methods.

Note that **LList**'s constructor maintains the optional parameter for minimum list size introduced for Class **AList**. This is done simply to keep the calls to the constructor the same for both variants. Because the linked list class does not need to declare a fixed-size array when the list is created, this parameter is unnecessary for linked lists. It is ignored by the implementation.

A key design decision for the linked list implementation is how to represent the current position. The most reasonable choices appear to be a pointer to the current element. But there is a big advantage to making **curr** point to the element preceding the current element.

Figure 4.5(a) shows the list's **curr** pointer pointing to the current element. The vertical line between the nodes containing 23 and 12 indicates the logical position of the current element. Consider what happens if we wish to insert a new node with value 10 into the list. The result should be as shown in Figure 4.5(b). However, there is a problem. To "splice" the list node containing the new element into the list, the list node storing 23 must have its **next** pointer changed to point to the new

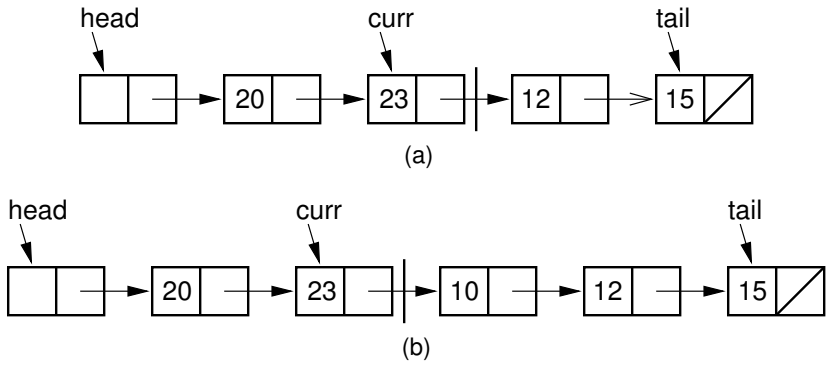


Figure 4.6 Insertion using a header node, with **curr** pointing one node head of the current element. (a) Linked list before insertion. The current node contains 12. (b) Linked list after inserting the node containing 10.

node. Unfortunately, there is no convenient access to the node preceding the one pointed to by **curr**.

There is an easy solution to this problem. If we set **curr** to point directly to the preceding element, there is no difficulty in adding a new element after **curr**. Figure 4.6 shows how the list looks when pointer variable **curr** is set to point to the node preceding the physical current node. See Exercise 4.5 for further discussion of why making **curr** point directly to the current element fails.

We encounter a number of potential special cases when the list is empty, or when the current position is at an end of the list. In particular, when the list is empty we have no element for **head**, **tail**, and **curr** to point to. Implementing special cases for **insert** and **remove** increases code complexity, making it harder to understand, and thus increases the chance of introducing a programming bug.

These special cases can be eliminated by implementing linked lists with an additional **header node** as the first node of the list. This header node is a link node like any other, but its value is ignored and it is not considered to be an actual element of the list. The header node saves coding effort because we no longer need to consider special cases for empty lists or when the current position is at one end of the list. The cost of this simplification is the space for the header node. However, there are space savings due to smaller code size, because statements to handle the special cases are omitted. In practice, this reduction in code size typically saves more space than that required for the header node, depending on the number of lists created. Figure 4.7 shows the state of an initialized or empty list when using a header node.

Figure 4.8 shows the definition for the linked list class, named **LList**. Class **LList** inherits from the abstract list class and thus must implement all of Class **List**'s member functions.

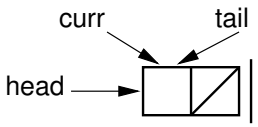


Figure 4.7 Initial state of a linked list when using a header node.

Implementations for most member functions of the **list** class are straightforward. However, **insert** and **remove** should be studied carefully.

Inserting a new element is a three-step process. First, the new list node is created and the new element is stored into it. Second, the **next** field of the new list node is assigned to point to the current node (the one *after* the node that **curr** points to). Third, the **next** field of node pointed to by **curr** is assigned to point to the newly inserted node. The following line in the **insert** method of Figure 4.8 does all three of these steps.

```
curr->next = new Link<E>(it, curr->next);
```

Operator **new** creates the new link node and calls the **Link** class constructor, which takes two parameters. The first is the element. The second is the value to be placed in the list node's **next** field, in this case"**curr->next**." Figure 4.9 illustrates this three-step process. Once the new node is added, **tail** is pushed forward if the new element was added to the end of the list. Insertion requires $\Theta(1)$ time.

Removing a node from the linked list requires only that the appropriate pointer be redirected around the node to be deleted. The following lines from the **remove** method of Figure 4.8 do precisely this.

```
Link<E>* ltemp = curr->next;   // Remember link node
curr->next = curr->next->next; // Remove from list
```

We must be careful not to "lose" the memory for the deleted link node. So, temporary pointer **ltemp** is first assigned to point to the node being removed. A call to **delete** is later used to return the old node to free storage. Figure 4.10 illustrates the **remove** method.Assuming that the free-store **delete** operator requires constant time, removing an element requires $\Theta(1)$ time.

Method **next** simply moves **curr** one position toward the tail of the list, which takes $\Theta(1)$ time. Method **prev** moves **curr** one position toward the head of the list, but its implementation is more difficult. In a singly linked list, there is no pointer to the previous node. Thus, the only alternative is to march down the list from the beginning until we reach the current node (being sure always to remember the node before it, because that is what we really want). This takes $\Theta(n)$ time in the average and worst cases. Implementation of method **moveToPos** is similar in that finding the ith position requires marching down i positions from the head of the list, taking $\Theta(i)$ time.

Implementations for the remaining operations each require $\Theta(1)$ time.

```
// Linked list implementation
template <typename E> class LList: public List<E> {
private:
  Link<E>* head;       // Pointer to list header
  Link<E>* tail;       // Pointer to last element
  Link<E>* curr;       // Access to current element
  int cnt;             // Size of list

  void init() {        // Intialization helper method
    curr = tail = head = new Link<E>;
    cnt = 0;
  }

  void removeall() {   // Return link nodes to free store
    while(head != NULL) {
      curr = head;
      head = head->next;
      delete curr;
    }
  }

public:
  LList(int size=defaultSize) { init(); }    // Constructor
  ~LList() { removeall(); }                   // Destructor
  void print() const;                 // Print list contents
  void clear() { removeall(); init(); }       // Clear list

  // Insert "it" at current position
  void insert(const E& it) {
    curr->next = new Link<E>(it, curr->next);
    if (tail == curr) tail = curr->next;  // New tail
    cnt++;
  }

  void append(const E& it) { // Append "it" to list
    tail = tail->next = new Link<E>(it, NULL);
    cnt++;
  }

  // Remove and return current element
  E remove() {
    Assert(curr->next != NULL, "No element");
    E it = curr->next->element;       // Remember value
    Link<E>* ltemp = curr->next;      // Remember link node
    if (tail == curr->next) tail = curr; // Reset tail
    curr->next = curr->next->next;    // Remove from list
    delete ltemp;                     // Reclaim space
    cnt--;                            // Decrement the count
    return it;
  }
```

Figure 4.8 A linked list implementation.

```
  void moveToStart() // Place curr at list start
    { curr = head; }

  void moveToEnd()   // Place curr at list end
    { curr = tail; }

  // Move curr one step left; no change if already at front
  void prev() {
    if (curr == head) return;       // No previous element
    Link<E>* temp = head;
    // March down list until we find the previous element
    while (temp->next!=curr) temp=temp->next;
    curr = temp;
  }

  // Move curr one step right; no change if already at end
  void next()
    { if (curr != tail) curr = curr->next; }

  int length() const  { return cnt; } // Return length

  // Return the position of the current element
  int currPos() const {
    Link<E>* temp = head;
    int i;
    for (i=0; curr != temp; i++)
      temp = temp->next;
    return i;
  }

  // Move down list to "pos" position
  void moveToPos(int pos) {
    Assert ((pos>=0)&&(pos<=cnt), "Position out of range");
    curr = head;
    for(int i=0; i<pos; i++) curr = curr->next;
  }

  const E& getValue() const { // Return current element
    Assert(curr->next != NULL, "No value");
    return curr->next->element;
  }
};
```

Figure 4.8 (continued)

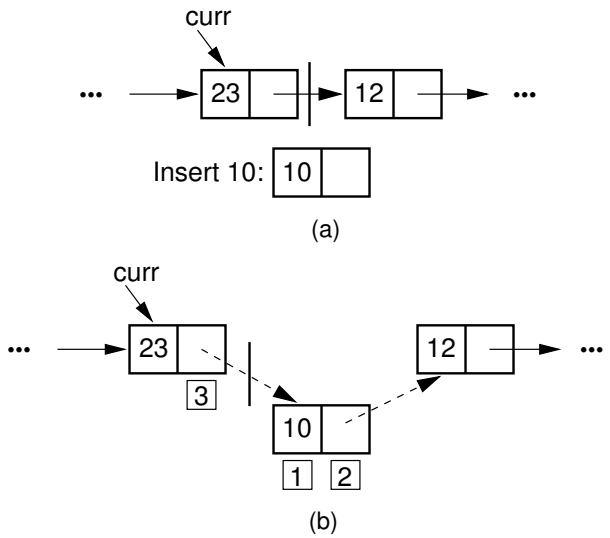


Figure 4.9 The linked list insertion process. (a) The linked list before insertion. (b) The linked list after insertion. 1 marks the **`element`** field of the new link node. 2 marks the **`next`** field of the new link node, which is set to point to what used to be the current node (the node with value 12). 3 marks the **`next`** field of the node preceding the current position. It used to point to the node containing 12; now it points to the new node containing 10.

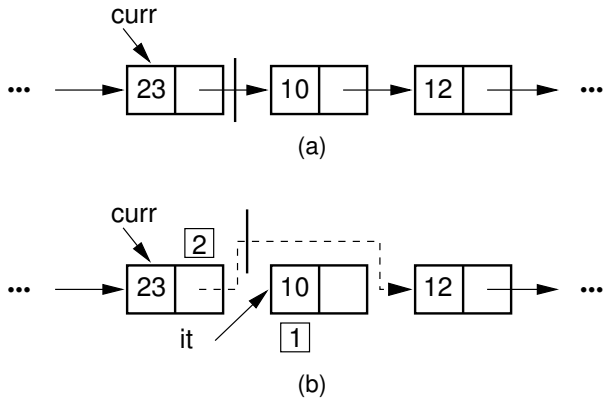


Figure 4.10 The linked list removal process. (a) The linked list before removing the node with value 10. (b) The linked list after removal. 1 marks the list node being removed. **`it`** is set to point to the element. 2 marks the **`next`** field of the preceding list node, which is set to point to the node following the one being deleted.

Freelists

The C++ free-store management operators **new** and **delete** are relatively expensive to use. Section 12.3 discusses how general-purpose memory managers are implemented. The expense comes from the fact that free-store routines must be capable of handling requests to and from free store with no particular pattern, as well as memory requests of vastly different sizes. Thismakes them inefficient compared to what might be implemented for more controlled patterns of memory access.

List nodes are created and deleted in a linked list implementation in a way that allows the **Link** class programmer to provide simple but efficient memory management routines. Instead of making repeated calls to **new** and **delete**, the **Link** class can handle its own **freelist**. A freelist holds those list nodes that are not currently being used. When a node is deleted from a linked list, it is placed at the head of the freelist. When a new element is to be added to a linked list, the freelist is checked to see if a list node is available. If so, the node is taken from the freelist. If the freelist is empty, the standard **new** operator must then be called.

Freelists are particularly useful for linked lists that periodically grow and then shrink. The freelist will never grow larger than the largest size yet reached by the linked list. Requests for new nodes (after the list has shrunk) can be handled by the freelist. Another good opportunity to use a freelist occurs when a program uses multiple lists. So long as they do not all grow and shrink together, the free list can let link nodes move between the lists.

One approach to implementing freelists would be to create two new operators to use instead of the standard free-store routines **new** and **delete**. This requires that the user's code, such as the linked list class implementation of Figure 4.8, be modified to call these freelist operators. A second approach is to use **C++ operator overloading** to replace the meaning of **new** and **delete** when operating on **Link** class objects. In this way, programs that use the **LList** class need not be modified at all to take advantage of a freelist. Whether the **Link** class is implemented with freelists, or relies on the regular free-store mechanism, is entirely hidden from the list class user. Figure 4.11 shows the reimplementation for the **Link** classwith freelist methods overloading the standard free-store operators. Note how simple they are, because they need only remove and add an element to the front of the freelist, respectively. The freelist versions of **new** and **delete** both run in $\Theta(1)$ time, except in the case where the freelist is exhausted and the **new** operation must be called. On my computer, a call to the overloaded **new** and **delete** operators requires about one tenth of the time required by the system free-store operators.

There is an additional efficiency gain to be had from a freelist implementation. The implementation of Figure 4.11 makes a separate call to the system **new** operator for each link node requested whenever the freelist is empty. These link nodes tend to be small — only a few bytes more than the size of the **element** field. If at some point in time the program requires thousands of active link nodes, these will

```
// Singly linked list node with freelist support
template <typename E> class Link {
private:
  static Link<E>* freelist; // Reference to freelist head
public:
  E element;                // Value for this node
  Link* next;                  // Point to next node in list

  // Constructors
  Link(const E& elemval, Link* nextval =NULL)
    { element = elemval;  next = nextval; }
  Link(Link* nextval =NULL) { next = nextval; }

  void* operator new(size_t) {  // Overloaded new operator
    if (freelist == NULL) return ::new Link; // Create space
    Link<E>* temp = freelist; // Can take from freelist
    freelist = freelist->next;
    return temp;                  // Return the link
  }

  // Overloaded delete operator
  void operator delete(void* ptr) {
    ((Link<E>*)ptr)->next = freelist; // Put on freelist
    freelist = (Link<E>*)ptr;
  }
};

// The freelist head pointer is actually created here
template <typename E>
Link<E>* Link<E>::freelist = NULL;
```

Figure 4.11 Implementation for the **Link** class with a freelist. Note that the redefinition for **new** refers to **::new** on the third line. This indicates that the standard C++ **new** operator is used, rather than the redefined **new** operator. If the colons had not been used, then the **Link** class **new** operator would be called, setting up an infinite recursion. The **static** declaration for member **freelist** means that all **Link** class objects share the same freelist pointer variable instead of each object storing its own copy.

have been created by many calls to the system version of **new**. An alternative is to allocate many link nodes in a single call to the system version of **new**, anticipating that if the freelist is exhausted now, more nodes will be needed soon. It is faster to make one call to **new** to get space for 100 **link** nodes, and then load all 100 onto the freelist at once, rather than to make 100 separate calls to **new**. The following statement will assign **ptr** to point to an array of 100 link nodes.

```
ptr = ::new Link[100];
```

The implementation for the **new** operator in the **link** class could then place each of these 100 nodes onto the freelist.

The **freelist** variable declaration uses the keyword **static**. This creates a single variable shared among all instances of the **Link** nodes. We want only a single freelist for all **Link** nodes of a given type. A program might create multiple lists. If they are all of the same type (that is, their element types are the same), then they can and should share the same freelist. This will happen with the implementation of Figure 4.11. If lists are created that have different element types, because this code is implemented with a template, the need for different list implementations will be discovered by the compiler at compile time. Separate versions of the list class will be generated for each element type. Thus, each element type will also get its own separate copy of the **Link** class. And each distinct **Link** class implementation will get a separate freelist.

4.1.3 Comparison of List Implementations

Now that you have seen two substantially different implementations for lists, it is natural to ask which is better. In particular, if you must implement a list for some task, which implementation should you choose?

Array-based lists have the disadvantage that their size must be predetermined before the array can be allocated. Array-based lists cannot grow beyond their predetermined size. Whenever the list contains only a few elements, a substantial amount of space might be tied up in a largely empty array. Linked lists have the advantage that they only need space for the objects actually on the list. There is no limit to the number of elements on a linked list, as long as there is free-store memory available. The amount of space required by a linked list is $\Theta(n)$, while the space required by the array-based list implementation is $\Omega(n)$, but can be greater.

Array-based lists have the advantage that there is no wasted space for an individual element. Linked lists require that an extra pointer be added to every list node. If the element size is small, then the overhead for links can be a significant fraction of the total storage. When the array for the array-based list is completely filled, there is no storage overhead. The array-based list will then be more space efficient, by a constant factor, than the linked implementation.

A simple formula can be used to determine whether the array-based list or linked list implementation will be more space efficient in a particular situation. Call n the number of elements currently in the list, P the size of a pointer in storage units (typically four bytes), E the size of a data element in storage units (this could be anything, from one bit for a Boolean variable on up to thousands of bytes or more for complex records), and D the maximum number of list elements that can be stored in the array. The amount of space required for the array-based list is DE, regardless of the number of elements actually stored in the list at any given time. The amount of space required for the linked list is $n(P + E)$. The smaller of these expressions for a given value n determines the more space-efficient implementation for n elements. In general, the linked implementation requires less space

than the array-based implementation when relatively few elements are in the list. Conversely, the array-based implementation becomes more space efficient when the array is close to full. Using the equation, we can solve for n to determine the break-even point beyond which the array-based implementation is more space efficient in any particular situation. This occurs when

$$n > DE/(P + E).$$

If $P = E$, then the break-even point is at $D/2$. This would happen if the element field is either a four-byte **int** value or a pointer, and the next field is a typical four-byte pointer. That is, the array-based implementation would be more efficient (if the link field and the element field are the same size) whenever the array is more than half full.

As a rule of thumb, linked lists are more space efficient when implementing lists whose number of elements varies widely or is unknown. Array-based lists are generally more space efficient when the user knows in advance approximately how large the list will become.

Array-based lists are faster for random access by position. Positions can easily be adjusted forwards or backwards by the **next** and **prev** methods. These operations always take $\Theta(1)$ time. In contrast, singly linked lists have no explicit access to the previous element, and access by position requires that we march down the list from the front (or the current position) to the specified position. Both of these operations require $\Theta(n)$ time in the average and worst cases, if we assume that each position on the list is equally likely to be accessed on any call to **prev** or **moveToPos**.

Given a pointer to a suitable location in the list, the **insert** and **remove** methods for linked lists require only $\Theta(1)$ time. Array-based lists must shift the remainder of the list up or down within the array. This requires $\Theta(n)$ time in the average and worst cases. For many applications, the time to insert and delete elements dominates all other operations. For this reason, linked lists are often preferred to array-based lists.

When implementing the array-based list, an implementor could allow the size of the array to grow and shrink depending on the number of elements that are actually stored. This data structure is known as a **dynamic array**. Both the Java and C++/STL **Vector** classes implement a dynamic array. Dynamic arrays allow the programmer to get around the limitation on the standard array that its size cannot be changed once the array has been created. This also means that space need not be allocated to the dynamic array until it is to be used. The disadvantage of this approach is that it takes time to deal with space adjustments on the array. Each time the array grows in size, its contents must be copied. A good implementation of the dynamic array will grow and shrink the array in such a way as to keep the overall cost for a series of insert/delete operations relatively inexpensive, even though an

occasional insert/delete operation might be expensive. A simple rule of thumb is to double the size of the array when it becomes full, and to cut the array size in half when it becomes one quarter full. To analyze the overall cost of dynamic array operations over time, we need to use a technique known as **amortized analysis**, which is discussed in Section 14.3.

4.1.4 Element Implementations

List users must decide whether they wish to store a copy of any given element on each list that contains it. For small elements such as an integer, this makes sense. If the elements are payroll records, it might be desirable for the list node to store a pointer to the record rather than store a copy of the record itself. This change would allow multiple list nodes (or other data structures) to point to the same record, rather than make repeated copies of the record. Not only might this save space, but it also means that a modification to an element's value is automatically reflected at all locations where it is referenced. The disadvantage of storing a pointer to each element is that the pointer requires space of its own. If elements are never duplicated, then this additional space adds unnecessary overhead.

The C++ implementations for lists presented in this section give the user of the list the choice of whether to store copies of elements or pointers to elements. The user can declare `E` to be, for example, a pointer to a payroll record. In this case, multiple lists can point to the same copy of the record. On the other hand, if the user declares `E` to be the record itself, then a new copy of the record will be made when it is inserted into the list.

Whether it is more advantageous to use pointers to shared elements or separate copies depends on the intended application. In general, the larger the elements and the more they are duplicated, the more likely that pointers to shared elements is the better approach.

A second issue faced by implementors of a list class (or any other data structure that stores a collection of user-defined data elements) is whether the elements stored are all required to be of the same type. This is known as **homogeneity** in a data structure. In some applications, the user would like to define the class of the data element that is stored on a given list, and then never permit objects of a different class to be stored on that same list. In other applications, the user would like to permit the objects stored on a single list to be of differing types.

For the list implementations presented in this section, the compiler requires that all objects stored on the list be of the same type. In fact, because the lists are implemented using templates, a new class is created by the compiler for each data type. For implementors who wish to minimize the number of classes created by the compiler, the lists can all store a `void*` pointer, with the user performing the necessary casting to and from the actual object type for each element. However, this

approach requires that the user do his or her own type checking, either to enforce homogeneity or to differentiate between the various object types.

Besides **C**++ templates, there are other techniques that implementors of a list class can use to ensure that the element type for a given list remains fixed, while still permitting different lists to store different element types. One approach is to store an object of the appropriate type in the header node of the list (perhaps an object of the appropriate type is supplied as a parameter to the list constructor), and then check that all insert operations on that list use the same element type.

The third issue that users of the list implementations must face is primarily of concern when programming in languages that do not support automatic garbage collection. That is how to deal with the memory of the objects stored on the list when the list is deleted or the **`clear`** method is called. The list destructor and the **`clear`** method are problematic in that there is a potential that they will bemisused, thus causing a memory leak. The type of the element stored determines whether there is a potential for trouble here. If the elements are of a simple type such as an **`int`**, then there is no need to delete the elements explicitly. If the elements are of a user-defined class, then their own destructor will be called. However, what if the list elements are pointers to objects? Then deleting **`listArray`** in the array-based implementation, or deleting a link node in the linked list implementation, might remove the only reference to an object, leaving its memory space inaccessible. Unfortunately, there is no way for the list implementation to know whether a given object is pointed to in another part of the program or not. Thus, the user of the list must be responsible for deleting these objects when that is appropriate.

4.1.5 Doubly Linked Lists

The singly linked list presented in Section 4.1.2 allows for direct access from a list node only to the next node in the list. A **doubly linked list** allows convenient access from a list node to the next node and also to the preceding node on the list. The doubly linked list node accomplishes this in the obvious way by storing two pointers: one to the node following it (as in the singly linked list), and a second pointer to the node preceding it. The most common reason to use a doubly linked list is because it is easier to implement than a singly linked list. While the code for the doubly linked implementation is a little longer than for the singly linked version, it tends to be a bit more "obvious" in its intention, and so easier to implement and debug. Figure 4.12 illustrates the doubly linked list concept. Whether a list implementation is doubly or singly linked should be hidden from the **`List`** class user.

Like our singly linked list implementation, the doubly linked list implementation makes use of a header node. We also add a tailer node to the end of the list. The tailer is similar to the header, in that it is a node that contains no value, and it always exists. When the doubly linked list is initialized, the header and tailer nodes

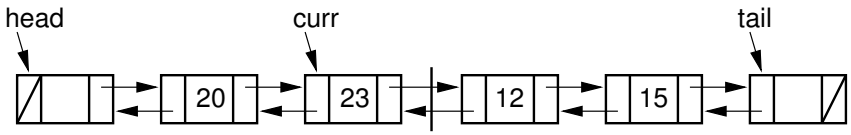


Figure 4.12 A doubly linked list.

are created. Data member **head** points to the header node, and **tail** points to the tailer node. The purpose of these nodes is to simplify the **insert**, **append**, and **remove** methods by eliminating all need for special-case code when the list is empty, or when we insert at the head or tail of the list.

For singly linked lists we set **curr** to point to the node preceding the node that contained the actual current element, due to lack of access to the previous node during insertion and deletion. Since we do have access to the previous node in a doubly linked list, this is no longer necessary. We could set **curr** to point directly to the node containing the current element. However, I have chosen to keep the same convention for the **curr** pointer as we set up for singly linked lists, purely for the sake of consistency.

Figure 4.13 shows the complete implementation for a **Link** class to be used with doubly linked lists. This code is a little longer than that for the singly linked list node implementation since the doubly linked list nodes have an extra data member.

Figure 4.14 shows the implementation for the **insert**, **append**, **remove**, and **prev** doubly linked list methods. The class declaration and the remaining member functions for the doubly linked list class are nearly identical to the singly linked list version.

The **insert** method is especially simple for our doubly linked list implementation, because most of the work is done by the node's constructor. Figure 4.15 shows the list before and after insertion of a node with value 10.

The three parameters to the **new** operator allow the list node class constructor to set the **element**, **prev**, and **next** fields, respectively, for the new link node. The **new** operator returns a pointer to the newly created node. The nodes to either side have their pointers updated to point to the newly created node. The existence of the header and tailer nodes mean that there are no special cases to worry about when inserting into an empty list.

The **append** method is also simple. Again, the **Link** class constructor sets the **element**, **prev**, and **next** fields of the node when the **new** operator is executed.

Method **remove** (illustrated by Figure 4.16) is straightforward, though the code is somewhat longer. First, the variable **it** is assigned the value being removed. Note that we must separate the element, which is returned to the caller, from the link object. The following lines then adjust the list.

```
// Doubly linked list link node with freelist support
template <typename E> class Link {
private:
  static Link<E>* freelist; // Reference to freelist head

public:
  E element;       // Value for this node
  Link* next;         // Pointer to next node in list
  Link* prev;         // Pointer to previous node

  // Constructors
  Link(const E& it, Link* prevp, Link* nextp) {
    element = it;
    prev = prevp;
    next = nextp;
  }
  Link(Link* prevp =NULL, Link* nextp =NULL) {
    prev = prevp;
    next = nextp;
  }

  void* operator new(size_t) {  // Overloaded new operator
    if (freelist == NULL) return ::new Link; // Create space
    Link<E>* temp = freelist; // Can take from freelist
    freelist = freelist->next;
    return temp;                 // Return the link
  }

  // Overloaded delete operator
  void operator delete(void* ptr) {
    ((Link<E>*)ptr)->next = freelist; // Put on freelist
    freelist = (Link<E>*)ptr;
  }
};

// The freelist head pointer is actually created here
template <typename E>
Link<E>* Link<E>::freelist = NULL;
```

Figure 4.13 Doubly linked list node implementation with a freelist.

```
// Insert "it" at current position
void insert(const E& it) {
  curr->next = curr->next->prev =
    new Link<E>(it, curr, curr->next);
  cnt++;
}

// Append "it" to the end of the list.
void append(const E& it) {
  tail->prev = tail->prev->next =
    new Link<E>(it, tail->prev, tail);
  cnt++;
}

// Remove and return current element
E remove() {
  if (curr->next == tail)        // Nothing to remove
    return NULL;
  E it = curr->next->element;    // Remember value
  Link<E>* ltemp = curr->next;   // Remember link node
  curr->next->next->prev = curr;
  curr->next = curr->next->next; // Remove from list
  delete ltemp;                  // Reclaim space
  cnt--;                         // Decrement cnt
  return it;
}

// Move fence one step left; no change if left is empty
void prev() {
  if (curr != head)  // Can't back up from list head
    curr = curr->prev;
}
```

Figure 4.14 Implementations for doubly linked list **insert**, **append**, **remove**, and **prev** methods.

```
Link<E>* ltemp = curr->next;   // Remember link node
curr->next->next->prev = curr;
curr->next = curr->next->next; // Remove from list
delete ltemp;                  // Reclaim space
```

The first line sets a temporary pointer to the node being removed. The second line makes the next node's **prev** pointer point to the left of the node being removed. Finally, the **next** field of the node preceding the one being deleted is adjusted. The final steps of method **remove** are to update the listlength, return the deleted node to free store, and return the value of the deleted element.

The only disadvantage of the doubly linked list as compared to the singly linked list is the additional space used. The doubly linked list requires two pointers per node, and so in the implementation presented it requires twice as much overhead as the singly linked list.

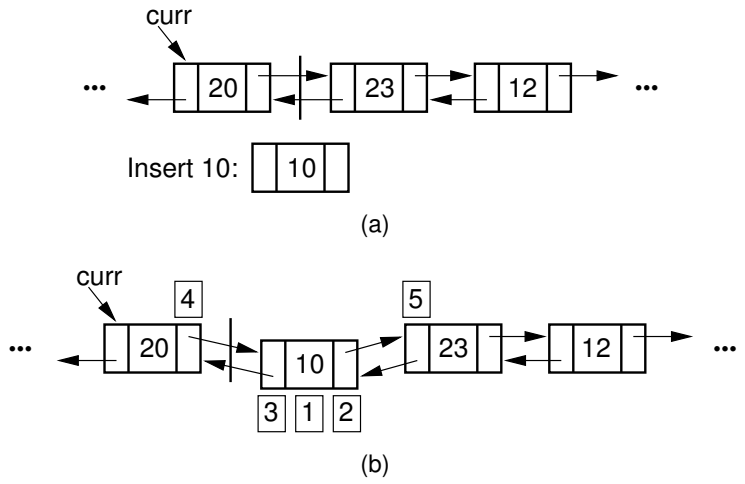


Figure 4.15 Insertion for doubly linked lists. The labels 1, 2, and 3 correspond to assignments done by the linked list node constructor. 4 marks the assignment to **`curr->next`**. 5 marks the assignment to the **`prev`** pointer of the node following the newly inserted node.

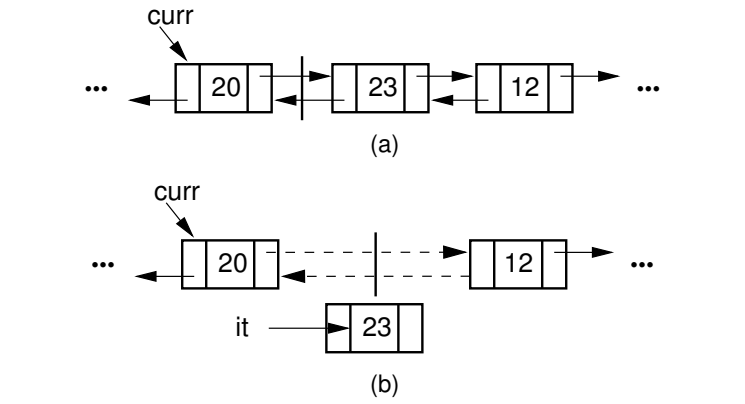


Figure 4.16 Doubly linked list removal. Element **`it`** stores the element of the node being removed. Then the nodes to either side have their pointers adjusted.

Example 4.1 There is a space-saving technique that can be employed to eliminate the additional space requirement, though it will complicate the implementation and be somewhat slower. Thus, this is an example of a space/time tradeoff. It is based on observing that, if we store the sum of two values, then we can get either value back by subtracting the other. That is, if we store $a + b$ in variable c, then $b = c - a$ and $a = c - b$. Of course, to recover one of the values out of the stored summation, the other value must be supplied. A pointer to the first node in the list, along with the value of one of its two link fields, will allow access to all of the remaining nodes of the list in order. This is because the pointer to the node must be the same as the value of the following node's **`prev`** pointer, as well as the previous node's **`next`** pointer. It is possible to move down the list breaking apart the summed link fields as though you were opening a zipper. Details for implementing this variation are left as an exercise.

The principle behind this technique is worth remembering, as it has many applications. The following code fragment will swap the contents of two variables without using a temporary variable (at the cost of three arithmetic operations).

```
a = a + b;
b = a - b; // Now b contains original value of a
a = a - b; // Now a contains original value of b
```

A similar effect can be had by using the exclusive-or operator. This fact is widely used in computer graphics. A region of the computer screen can be highlighted by XORing the outline of a box around it. XORing the box outline a second time restores the original contents of the screen.

4.2 Stacks

The **stack** is a list-like structure in which elements may be inserted or removed from only one end. While this restriction makes stacks less flexible than lists, it also makes stacks both efficient (for those operations they can do) and easy to implement. Many applications require only the limited form of insert and remove operations that stacks provide. In such cases, it is more efficient to use the simpler stack data structure rather than the generic list. For example, the freelist of Section 4.1.2 is really a stack.

Despite their restrictions, stacks have many uses. Thus, a special vocabulary for stacks has developed. Accountants used stacks long before the invention of the computer. They called the stack a "LIFO" list, which stands for "Last-In, First-

```
// Stack abtract class
template <typename E> class Stack {
private:
  void operator =(const Stack&) {}     // Protect assignment
  Stack(const Stack&) {}          // Protect copy constructor

public:
  Stack() {}                        // Default constructor
  virtual ~Stack() {}               // Base destructor

  // Reinitialize the stack.  The user is responsible for
  // reclaiming the storage used by the stack elements.
  virtual void clear() = 0;

  // Push an element onto the top of the stack.
  // it: The element being pushed onto the stack.
  virtual void push(const E& it) = 0;

  // Remove the element at the top of the stack.
  // Return: The element at the top of the stack.
  virtual E pop() = 0;

  // Return: A copy of the top element.
  virtual const E& topValue() const = 0;

  // Return: The number of elements in the stack.
  virtual int length() const = 0;
};
```

Figure 4.17 The stack ADT.

Out." Note that one implication of the LIFO policy is that stacks remove elements in reverse order of their arrival.

The accessible element of the stack is called the **top** element. Elements are not said to be inserted, they are **pushed** onto the stack. When removed, an element is said to be **popped** from the stack. Figure 4.17 shows a sample stack ADT.

As with lists, there are many variations on stack implementation. The two approaches presented here are **array-based** and **linked stacks**, which are analogous to array-based and linked lists, respectively.

4.2.1 Array-Based Stacks

Figure 4.18 shows a complete implementation for the array-based stack class. As with the array-based list implementation, **`listArray`** must be declared of fixed size when the stack is created. In the stack constructor, **`size`** serves to indicate this size. Method **`top`** acts somewhat like a current position value (because the "current" position is always at the top of the stack), as well as indicating the number of elements currently in the stack.

```
// Array-based stack implementation
template <typename E> class AStack: public Stack<E> {
private:
  int maxSize;              // Maximum size of stack
  int top;                  // Index for top element
  E *listArray;          // Array holding stack elements

public:
  AStack(int size =defaultSize)   // Constructor
    { maxSize = size; top = 0; listArray = new E[size]; }

  ~AStack() { delete [] listArray; }  // Destructor

  void clear() { top = 0; }           // Reinitialize

  void push(const E& it) {         // Put "it" on stack
    Assert(top != maxSize, "Stack is full");
    listArray[top++] = it;
  }

  E pop() {                 // Pop top element
    Assert(top != 0, "Stack is empty");
    return listArray[--top];
  }

  const E& topValue() const {     // Return top element
    Assert(top != 0, "Stack is empty");
    return listArray[top-1];
  }

  int length() const { return top; }  // Return length
};
```

Figure 4.18 Array-based stack class implementation.

The array-based stack implementation is essentially a simplified version of the array-based list. The only important design decision to be made is which end of the array should represent the top of the stack. One choice is to make the top be at position 0 in the array. In terms of list functions, all **insert** and **remove** operations would then be on the element in position 0. This implementation is inefficient, because now every **push** or **pop** operation will require that all elements currently in the stack be shifted one position in the array, for a cost of $\Theta(n)$ if there are n elements. The other choice is have the top element be at position $n-1$ when there are n elements in the stack. In other words, as elements are pushed onto the stack, they are appended to the tail of the list. Method **pop** removes the tail element. In this case, the cost for each **push** or **pop** operation is only $\Theta(1)$.

For the implementation of Figure 4.18, **top** is defined to be the array index of the first free position in the stack. Thus, an empty stack has **top** set to 0, the first available free position in the array. (Alternatively, **top** could have been defined to

be the index for the top element in the stack, rather than the first free position. If this had been done, the empty list would initialize **top** as -1.) Methods **push** and **pop** simply place an element into, or remove an element from, the array position indicated by **top**. Because **top** is assumed to be at the first free position, **push** first inserts its value into the top position and then increments **top**, while **pop** first decrements **top** and then removes the top element.

4.2.2 Linked Stacks

The linked stack implementation is quite simple. The freelist of Section 4.1.2 is an example of a linked stack. Elements are inserted and removed only from the head of the list. A header node is not used because no special-case code is required for lists of zero or one elements. Figure 4.19 shows the complete linked stack implementation. The only data member is **top**, a pointer to the first (top) link node of the stack. Method **push** first modifies the **next** field of the newly created link node to point to the top of the stack and then sets **top** to point to the new link node. Method **pop** is also quite simple. Variable **temp** stores the top nodes' value, while **ltemp** links to the top node as it is removed from the stack. The stack is updated by setting **top** to point to the next link in the stack. The old top node is then returned to free store (or the freelist), and the element value is returned.

4.2.3 Comparison of Array-Based and Linked Stacks

All operations for the array-based and linked stack implementations take constant time, so from a time efficiency perspective, neither has a significant advantage. Another basis for comparison is the total space required. The analysis is similar to that done for list implementations. The array-based stack must declare a fixed-size array initially, and some of that space is wasted whenever the stack is not full. The linked stack can shrink and grow but requires the overhead of a link field for every element.

When multiple stacks are to be implemented, it is possible to take advantage of the one-way growth of the array-based stack. This can be done by using a single array to store two stacks. One stack grows inward from each end as illustrated by Figure 4.20, hopefully leading to less wasted space. However, this only works well when the space requirements of the two stacks are inversely correlated. In other words, ideally when one stack grows, the other will shrink. This is particularly effective when elements are taken from one stack and given to the other. If instead both stacks grow at the same time, then the free space in the middle of the array will be exhausted quickly.

```
// Linked stack implementation
template <typename E> class LStack: public Stack<E> {
private:
  Link<E>* top;             // Pointer to first element
  int size;                    // Number of elements

public:
  LStack(int sz =defaultSize) // Constructor
    { top = NULL; size = 0; }

  ~LStack() { clear(); }           // Destructor

  void clear() {                   // Reinitialize
    while (top != NULL) {          // Delete link nodes
      Link<E>* temp = top;
      top = top->next;
      delete temp;
    }
    size = 0;
  }

  void push(const E& it) { // Put "it" on stack
    top = new Link<E>(it, top);
    size++;
  }

  E pop() {                 // Remove "it" from stack
    Assert(top != NULL, "Stack is empty");
    E it = top->element;
    Link<E>* ltemp = top->next;
    delete top;
    top = ltemp;
    size--;
    return it;
  }

  const E& topValue() const { // Return top value
    Assert(top != 0, "Stack is empty");
    return top->element;
  }

  int length() const { return size; } // Return length
};
```

Figure 4.19 Linked stack class implementation.

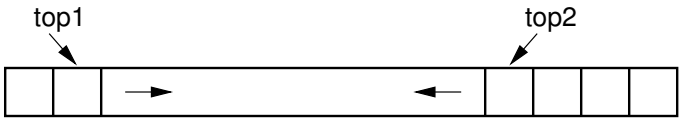


Figure 4.20 Two stacks implemented within in a single array, both growing toward the middle.

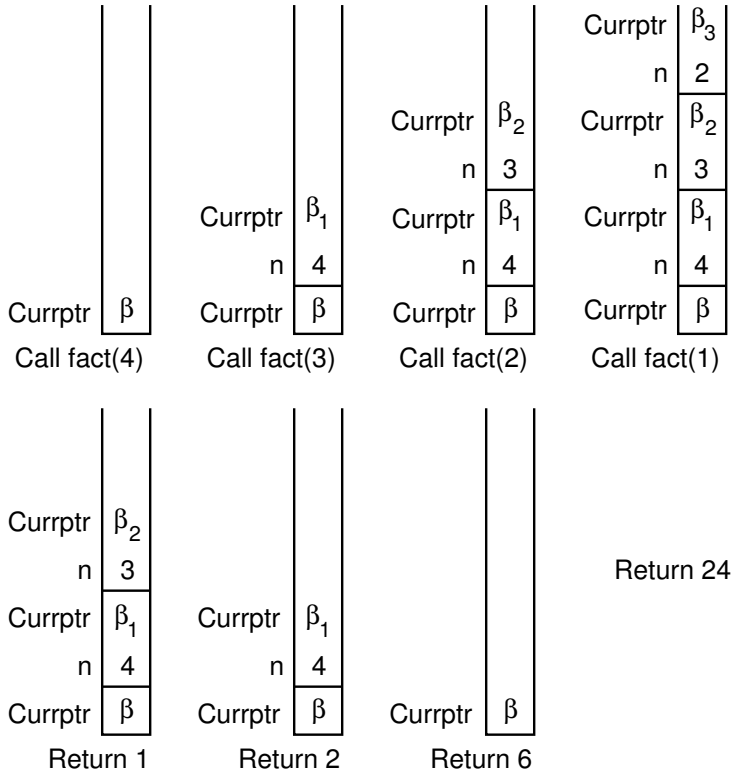


Figure 4.21 Implementing recursion with a stack. β values indicate the address of the program instruction to return to after completing the current function call. On each recursive function call to **`fact`** (as implemented in Section 2.5), both the return address and the current value of **`n`** must be saved. Each return from **`fact`** pops the top activation record off the stack.

4.2.4 Implementing Recursion

Perhaps the most common computer application that uses stacks is not even visible to its users. This is the implementation of subroutine calls in most programming language runtime environments. A subroutine call is normally implemented by placing necessary information about the subroutine (including the return address, parameters, and local variables) onto a stack. This information is called an **activation record**. Further subroutine calls add to the stack. Each return from a subroutine pops the top activation record off the stack. Figure 4.21 illustrates the implementation of the recursive factorial function of Section 2.5 from the runtime environment's point of view.

Consider what happens when we call **`fact`** with the value 4. We use β to indicate the address of the program instruction where the call to **`fact`** is made. Thus, the stack must first store the address β, and the value 4 is passed to **`fact`**.

Next, a recursive call to **fact** is made, this time with value 3. We will name the program address from which the call is made β_1. The address β_1, along with the current value for n (which is 4), is saved on the stack. Function **fact** is invoked with input parameter 3.

In similar manner, another recursive call is made with input parameter 2, requiring that the address from which the call is made (say β_2) and the current value for n (which is 3) are stored on the stack. A final recursive call with input parameter 1 is made, requiring that the stack store the calling address (say β_3) and current value (which is 2).

At this point, we have reached the base case for **fact**, and so the recursion begins to unwind. Each return from **fact** involves popping the stored value for n from the stack, along with the return address from the function call. The return value for **fact** is multiplied by the restored value for n, and the result is returned.

Because an activation record must be created and placed onto the stack for each subroutine call, making subroutine calls is a relatively expensive operation. While recursion is often used to make implementation easy and clear, sometimes you might want to eliminate the overhead imposed by the recursive function calls. In some cases, such as the factorial function of Section 2.5, recursion can easily be replaced by iteration.

Example 4.2 As a simple example of replacing recursion with a stack, consider the following non-recursive version of the factorial function.

```
long fact(int n, Stack<int>& S) { // Compute n!
  // To fit n! in a long variable, require n <= 12
  Assert((n >= 0) && (n <= 12), "Input out of range");
  while (n > 1) S.push(n--);  // Load up the stack
  long result = 1;            // Holds final result
  while(S.length() > 0)
     result = result * S.pop();   // Compute
  return result;
}
```

Here, we simply push successively smaller values of n onto the stack until the base case is reached, then repeatedly pop off the stored values and multiply them into the result.

An iterative form of the factorial function is both simpler and faster than the version shown in Example 4.2. But it is not always possible to replace recursion with iteration. Recursion, or some imitation of it, is necessary when implementing algorithms that require multiple branching such as in the Towers of Hanoi algorithm, or when traversing a binary tree. The Mergesort and Quicksort algorithms of Chapter 7 are also examples in which recursion is required. Fortunately, it is always possible to imitate recursion with a stack. Let us now turn to a non-recursive version of the Towers of Hanoi function, which cannot be done iteratively.

Example 4.3 The **TOH** function shown in Figure 2.2 makes two recursive calls: one to move $n-1$ rings off the bottom ring, and another to move these $n-1$ rings back to the goal pole. We can eliminate the recursion by using a stack to store a representation of the three operations that **TOH** must perform: two recursive calls and a move operation. To do so, we must first come up with a representation of the various operations, implemented as a class whose objects will be stored on the stack.

Figure 4.22 shows such a class. We first define an enumerated type called **TOHop**, with two values MOVE and TOH, to indicate calls to the **move** function and recursive calls to **TOH**, respectively. Class **TOHobj** stores five values: an operation field (indicating either a move or a new TOH operation), the number of rings, and the three poles. Note that the move operation actually needs only to store information about two poles. Thus, there are two constructors: one to store the state when imitating a recursive call, and one to store the state for a move operation.

An array-based stack is used because we know that the stack will need to store exactly $2n+1$ elements. The new version of **TOH** begins by placing on the stack a description of the initial problem for n rings. The rest of the function is simply a **while** loop that pops the stack and executes the appropriate operation. In the case of a **TOH** operation (for $n > 0$), we store on the stack representations for the three operations executed by the recursive version. However, these operations must be placed on the stack in reverse order, so that they will be popped off in the correct order.

Recursive algorithms lend themselves to efficient implementation with a stack when the amount of information needed to describe a sub-problem is small. For example, Section 7.5 discusses a stack-based implementation for Quicksort.

4.3 Queues

Like the stack, the **queue** is a list-like structure that provides restricted access to its elements. Queue elements may only be inserted at the back (called an **enqueue** operation) and removed from the front (called a **dequeue** operation). Queues operate like standing in line at a movie theater ticket counter.[1] If nobody cheats, then newcomers go to the back of the line. The person at the front of the line is the next to be served. Thus, queues release their elements in order of arrival. Accountants have used queues since long before the existence of computers. They call a queue a "FIFO" list, which stands for "First-In, First-Out." Figure 4.23 shows a sample

[1] In Britain, a line of people is called a "queue," and getting into line to wait for service is called "queuing up."

```
// Operation choices: DOMOVE will move a disk
// DOTOH corresponds to a recursive call
enum TOHop { DOMOVE, DOTOH };
class TOHobj { // An operation object
public:
  TOHop op;              // This operation type
  int num;               // How many disks
  Pole start, goal, tmp; // Define pole order

  // DOTOH operation constructor
  TOHobj(int n, Pole s, Pole g, Pole t) {
    op = DOTOH; num = n;
    start = s; goal = g; tmp = t;
  }

  // DOMOVE operation constructor
  TOHobj(Pole s, Pole g)
    { op = DOMOVE; start = s; goal = g; }
};

void TOH(int n, Pole start, Pole goal, Pole tmp,
         Stack<TOHobj*>& S) {
  S.push(new TOHobj(n, start, goal, tmp)); // Initial
  TOHobj* t;
  while (S.length() > 0) {       // Grab next task
    t = S.pop();
    if (t->op == DOMOVE)   // Do a move
      move(t->start, t->goal);
    else if (t->num > 0) {
      // Store (in reverse) 3 recursive statements
      int num = t->num;
      Pole tmp = t->tmp;  Pole goal = t->goal;
      Pole start = t->start;
      S.push(new TOHobj(num-1, tmp, goal, start));
      S.push(new TOHobj(start, goal));
      S.push(new TOHobj(num-1, start, tmp, goal));
    }
    delete t; // Must delete the TOHobj we made
  }
}
```

Figure 4.22 Stack-based implementation for Towers of Hanoi.

queue ADT. This section presents two implementations for queues: the array-based queue and the linked queue.

4.3.1 Array-Based Queues

The array-based queue is somewhat tricky to implement effectively. A simple conversion of the array-based list implementation is not efficient.

Assume that there are n elements in the queue. By analogy to the array-based list implementation, we could require that all elements of the queue be stored in the first n positions of the array. If we choose the rear element of the queue to be in

```
// Abstract queue class
template <typename E> class Queue {
private:
  void operator =(const Queue&) {}     // Protect assignment
  Queue(const Queue&) {}          // Protect copy constructor

public:
  Queue() {}           // Default
  virtual ~Queue() {} // Base destructor

  // Reinitialize the queue.  The user is responsible for
  // reclaiming the storage used by the queue elements.
  virtual void clear() = 0;

  // Place an element at the rear of the queue.
  // it: The element being enqueued.
  virtual void enqueue(const E&) = 0;

  // Remove and return element at the front of the queue.
  // Return: The element at the front of the queue.
  virtual E dequeue() = 0;

  // Return: A copy of the front element.
  virtual const E& frontValue() const = 0;

  // Return: The number of elements in the queue.
  virtual int length() const = 0;
};
```

Figure 4.23 The C++ ADT for a queue.

position 0, then **dequeue** operations require only $\Theta(1)$ time because the front element of the queue (the one being removed) is the last element in the array. However, **enqueue** operations will require $\Theta(n)$ time, because the n elements currently in the queue must each be shifted one position in the array. If instead we chose the rear element of the queue to be in position $n-1$, then an **enqueue** operation is equivalent to an **append** operation on a list. This requires only $\Theta(1)$ time. But now, a **dequeue** operation requires $\Theta(n)$ time, because all of the elements must be shifted down by one position to retain the property that the remaining $n-1$ queue elements reside in the first $n-1$ positions of the array.

A far more efficient implementation can be obtained by relaxing the requirement that all elements of the queue must be in the first n positions of the array. We will still require that the queue be stored be in contiguous array positions, but the contents of the queue will be permitted to drift within the array, as illustrated by Figure 4.24. Now, both the **enqueue** and the **dequeue** operations can be performed in $\Theta(1)$ time because no other elements in the queue need be moved.

This implementation raises a new problem. Assume that the front element of the queue is initially at position 0, and that elements are added to successively

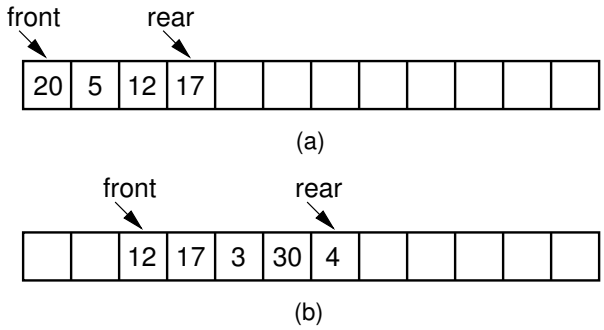


Figure 4.24 After repeated use, elements in the array-based queue will drift to the back of the array. (a) The queue after the initial four numbers 20, 5, 12, and 17 have been inserted. (b) The queue after elements 20 and 5 are deleted, following which 3, 30, and 4 are inserted.

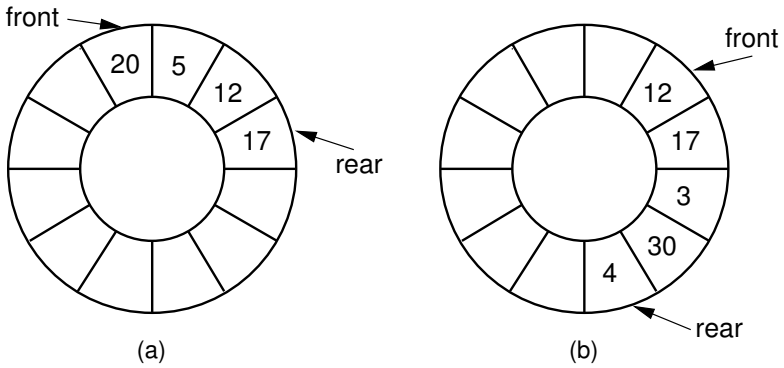


Figure 4.25 The circular queue with array positions increasing in the clockwise direction. (a) The queue after the initial four numbers 20, 5, 12, and 17 have been inserted. (b) The queue after elements 20 and 5 are deleted, following which 3, 30, and 4 are inserted.

higher-numbered positions in the array. When elements are removed from the queue, the front index increases. Over time, the entire queue will drift toward the higher-numbered positions in the array. Once an element is inserted into the highest-numbered position in the array, the queue has run out of space. This happens despite the fact that there might be free positions at the low end of the array where elements have previously been removed from the queue.

The "drifting queue" problem can be solved by pretending that the array is circular and so allow the queue to continue directly from the highest-numbered position in the array to the lowest-numbered position. This is easily implemented through use of the modulus operator (denoted by `%` in C++). In this way, positions in the array are numbered from 0 through `size`-1, and position `size`-1 is defined to immediately precede position 0 (which is equivalent to position `size % size`). Figure 4.25 illustrates this solution.

There remains one more serious, though subtle, problem to the array-based queue implementation. How can we recognize when the queue is empty or full? Assume that **front** stores the array index for the front element in the queue, and **rear** stores the array index for the rear element. If both **front** and **rear** have the same position, then with this scheme there must be one element in the queue. Thus, an empty queue would be recognized by having **rear** be *one less* than **front** (taking into account the fact that the queue is circular, so position **size**-1 is actually considered to be one less than position 0). But what if the queue is completely full? In other words, what is the situation when a queue with n array positions available contains n elements? In this case, if the front element is in position 0, then the rear element is in position **size**-1. But this means that the value for **rear** is one less than the value for **front** when the circular nature of the queue is taken into account. In other words, the full queue is indistinguishable from the empty queue!

You might think that the problem is in the assumption about **front** and **rear** being defined to store the array indices of the front and rear elements, respectively, and that some modification in this definition will allow a solution. Unfortunately, the problem cannot be remedied by a simple change to the definition for **front** and **rear**, because of the number of conditions or **states** that the queue can be in. Ignoring the actual position of the first element, and ignoring the actual values of the elements stored in the queue, how many different states are there? There can be no elements in the queue, one element, two, and so on. At most there can be n elements in the queue if there are n array positions. This means that there are $n + 1$ different states for the queue (0 through n elements are possible).

If the value of **front** is fixed, then $n+1$ different values for **rear** are needed to distinguish among the $n+1$ states. However, there are only n possible values for **rear** unless we invent a special case for, say, empty queues. This is an example of the Pigeonhole Principle defined in Exercise 2.30. The Pigeonhole Principle states that, given n pigeonholes and $n + 1$ pigeons, when all of the pigeons go into the holes we can be sure that at least one hole contains more than one pigeon. In similar manner, we can be sure that two of the $n + 1$ states are indistinguishable by the n relative values of **front** and **rear**. We must seek some other way to distinguish full from empty queues.

One obvious solution is to keep an explicit count of the number of elements in the queue, or at least a Boolean variable that indicates whether the queue is empty or not. Another solution is to make the array be of size $n + 1$, and only allow n elements to be stored. Which of these solutions to adopt is purely a matter of the implementor's taste in such affairs. My choice is to use an array of size $n + 1$.

Figure 4.26 shows an array-based queue implementation. **listArray** holds the queue elements, and as usual, the queue constructor allows an optional parameter to set the maximum size of the queue. The array as created is actually large enough to hold one element more than the queue will allow, so that empty queues

```
// Array-based queue implementation
template <typename E> class AQueue: public Queue<E> {
private:
  int maxSize;               // Maximum size of queue
  int front;                 // Index of front element
  int rear;                  // Index of rear element
  E *listArray;           // Array holding queue elements

public:
  AQueue(int size =defaultSize) {  // Constructor
    // Make list array one position larger for empty slot
    maxSize = size+1;
    rear = 0;  front = 1;
    listArray = new E[maxSize];
  }

  ~AQueue() { delete [] listArray; } // Destructor

  void clear() { rear = 0; front = 1; } // Reinitialize

  void enqueue(const E& it) {     // Put "it" in queue
    Assert(((rear+2) % maxSize) != front, "Queue is full");
    rear = (rear+1) % maxSize;       // Circular increment
    listArray[rear] = it;
  }

  E dequeue() {           // Take element out
    Assert(length() != 0, "Queue is empty");
    E it = listArray[front];
    front = (front+1) % maxSize;    // Circular increment
    return it;
  }

  const E& frontValue() const {  // Get front value
    Assert(length() != 0, "Queue is empty");
    return listArray[front];
  }

  virtual int length() const         // Return length
   { return ((rear+maxSize) - front + 1) % maxSize; }
};
```

Figure 4.26 An array-based queue implementation.

can be distinguished from full queues. Member **maxSize** is used to control the circular motion of the queue (it is the base for the modulus operator). Member **rear** is set to the position of the current rear element, while **front** is the position of the current front element.

In this implementation, the front of the queue is defined to be toward the lower numbered positions in the array (in the counter-clockwise direction in Figure 4.25), and the rear is defined to be toward the higher-numbered positions. Thus, **enqueue** increments the rear pointer (modulus **size**), and **dequeue** increments the front pointer. Implementation of all member functions is straightforward.

4.3.2 Linked Queues

The linked queue implementation is a straightforward adaptation of the linked list. Figure 4.27 shows the linked queue class declaration. Methods **front** and **rear** are pointers to the front and rear queue elements, respectively. We will use a header link node, which allows for a simpler implementation of the enqueue operation by avoiding any special cases when the queue is empty. On initialization, the **front** and **rear** pointers will point to the header node, and front will always point to the header node while rear points to the true last link node in the queue. Method **enqueue** places the new element in a link node at the end of the linked list (i.e., the node that **rear** points to) and then advances **rear** to point to the new link node. Method **dequeue** removes and returns the first element of the list.

4.3.3 Comparison of Array-Based and Linked Queues

All member functions for both the array-based and linked queue implementations require constant time. The space comparison issues are the same as for the equivalent stack implementations. Unlike the array-based stack implementation, there is no convenient way to store two queues in the same array, unless items are always transferred directly from one queue to the other.

4.4 Dictionaries

The most common objective of computer programs is to store and retrieve data. Much of this book is about efficient ways to organize collections of data records so that they can be stored and retrieved quickly. In this section we describe a simple interface for such a collection, called a **dictionary**. The dictionary ADT provides operations for storing records, finding records, and removing records from the collection. This ADT gives us a standard basis for comparing various data structures.

Before we can discuss the interface for a dictionary, we must first define the concepts of a **key** and **comparable** objects. If we want to search for a given record

```
// Linked queue implementation
template <typename E> class LQueue: public Queue<E> {
private:
  Link<E>* front;       // Pointer to front queue node
  Link<E>* rear;        // Pointer to rear queue node
  int size;                // Number of elements in queue

public:
  LQueue(int sz =defaultSize) // Constructor
    { front = rear = new Link<E>(); size = 0; }

  ~LQueue() { clear(); delete front; }      // Destructor

  void clear() {           // Clear queue
    while(front->next != NULL) { // Delete each link node
      rear = front;
      delete rear;
    }
    rear = front;
    size = 0;
  }

  void enqueue(const E& it) { // Put element on rear
    rear->next = new Link<E>(it, NULL);
    rear = rear->next;
    size++;
  }

  E dequeue() {              // Remove element from front
    Assert(size != 0, "Queue is empty");
    E it = front->next->element;  // Store dequeued value
    Link<E>* ltemp = front->next; // Hold dequeued link
    front->next = ltemp->next;       // Advance front
    if (rear == ltemp) rear = front; // Dequeue last element
    delete ltemp;                    // Delete link
    size --;
    return it;                       // Return element value
  }

  const E& frontValue() const { // Get front element
    Assert(size != 0, "Queue is empty");
    return front->next->element;
  }

  virtual int length() const { return size; }
};
```

Figure 4.27 Linked queue class implementation.

in a database, how should we describe what we are looking for? A database record could simply be a number, or it could be quite complicated, such as a payroll record with many fields of varying types. We do not want to describe what we are looking for by detailing and matching the entire contents of the record. If we knew everything about the record already, we probably would not need to look for it. Instead, we typically define what record we want in terms of a key value. For example, if searching for payroll records, we might wish to search for the record that matches a particular ID number. In this example the ID number is the **search key**.

To implement the search function, we require that keys be comparable. At a minimum, we must be able to take two keys and reliably determine whether they are equal or not. That is enough to enable a sequential search through a database of records and find one that matches a given key. However, we typically would like for the keys to define a total order (see Section 2.1), which means that we can tell which of two keys is greater than the other. Using key types with total orderings gives the database implementor the opportunity to organize a collection of records in a way that makes searching more efficient. An example is storing the records in sorted order in an array, which permits a binary search. Fortunately, in practice most fields of most records consist of simple data types with natural total orders. For example, integers, floats, doubles, and character strings all are totally ordered. Ordering fields that are naturally multi-dimensional, such as a point in two or three dimensions, present special opportunities if we wish to take advantage of their multidimensional nature. This problem is addressed in Section 13.3.

Figure 4.28 shows the definition for a simple abstract dictionary class. The methods **`insert`** and **`find`** are the heart of the class. Method **`insert`** takes a record and inserts it into the dictionary. Method **`find`** takes a key value and returns some record from the dictionary whose key matches the one provided. If there are multiple records in the dictionary with that key value, there is no requirement as to which one is returned.

Method **`clear`** simply re-initializes the dictionary. The **`remove`** method is similar to **`find`**, except that it also deletes the record returned from the dictionary. Once again, if there are multiple records in the dictionary that match the desired key, there is no requirement as to which one actually is removed and returned. Method **`size`** returns the number of elements in the dictionary.

The remaining Method is **`removeAny`**. This is similar to **`remove`**, except that it does not take a key value. Instead, it removes an arbitrary record from the dictionary, if one exists. The purpose of this method is to allow a user the ability to iterate over all elements in the dictionary (of course, the dictionary will become empty in the process). Without the **`removeAny`** method, a dictionary user could not get at a record of the dictionary that he didn't already know the key value for. With the **`removeAny`** method, the user can process all records in the dictionary as shown in the following code fragment.

```
// The Dictionary abstract class.
template <typename Key, typename E>
class  Dictionary {
private:
  void operator =(const Dictionary&) {}
  Dictionary(const Dictionary&) {}

public:
  Dictionary() {}          // Default constructor
  virtual ~Dictionary() {} // Base destructor

  // Reinitialize dictionary
  virtual void clear() = 0;

  // Insert a record
  // k: The key for the record being inserted.
  // e: The record being inserted.
  virtual void insert(const Key& k, const E& e) = 0;

  // Remove and return a record.
  // k: The key of the record to be removed.
  // Return: A maching record. If multiple records match
  // "k", remove an arbitrary one. Return NULL if no record
  // with key "k" exists.
  virtual E remove(const Key& k) = 0;

  // Remove and return an arbitrary record from dictionary.
  // Return: The record removed, or NULL if none exists.
  virtual E removeAny() = 0;

  // Return: A record matching "k" (NULL if none exists).
  // If multiple records match, return an arbitrary one.
  // k: The key of the record to find
  virtual E find(const Key& k) const = 0;

  // Return the number of records in the dictionary.
  virtual int size() = 0;
};
```

Figure 4.28 The ADT for a simple dictionary.

```
// A simple payroll entry with ID, name, address fields
class Payroll {
private:
  int ID;
  string name;
  string address;

public:
  // Constructor
  Payroll(int inID, string inname, string inaddr) {
    ID = inID;
    name = inname;
    address = inaddr;
  }

  ~Payroll() {}  // Destructor

  // Local data member access functions
  int getID() { return ID; }
  string getname() { return name; }
  string getaddr() { return address; }
};
```

Figure 4.29 A payroll record implementation.

```
while (dict.size() > 0) {
  it = dict.removeAny();
  doSomething(it);
}
```

There are other approaches that might seem more natural for iterating though a dictionary, such as using a "first" and a "next" function. But not all data structures that we want to use to implement a dictionary are able to do "first" efficiently. For example, a hash table implementation cannot efficiently locate the record in the table with the smallest key value. By using **`RemoveAny`**, we have a mechanism that provides generic access.

Given a database storing records of a particular type, we might want to search for records in multiple ways. For example, we might want to store payroll records in one dictionary that allows us to search by ID, and also store those same records in a second dictionary that allows us to search by name.

Figure 4.29 shows an implementation for a payroll record. Class **`Payroll`** has multiple fields, each of which might be used as a search key. Simply by varying the type for the key, and using the appropriate field in each record as the key value, we can define a dictionary whose search key is the ID field, another whose search key is the name field, and a third whose search key is the address field. Figure 4.30 shows an example where **`Payroll`** objects are stored in two separate dictionaries, one using the ID field as the key and the other using the name field as the key.

```
int main() {
  // IDdict organizes Payroll records by ID
  UALdict<int, Payroll*> IDdict;
  // namedict organizes Payroll records by name
  UALdict<string, Payroll*> namedict;
  Payroll *foo1, *foo2, *findfoo1, *findfoo2;

  foo1 = new Payroll(5, "Joe", "Anytown");
  foo2 = new Payroll(10, "John", "Mytown");

  IDdict.insert(foo1->getID(), foo1);
  IDdict.insert(foo2->getID(), foo2);
  namedict.insert(foo1->getname(), foo1);
  namedict.insert(foo2->getname(), foo2);

  findfoo1 = IDdict.find(5);
  if (findfoo1 != NULL) cout << findfoo1;
  else cout << "NULL ";
  findfoo2 = namedict.find("John");
  if (findfoo2 != NULL) cout << findfoo2;
  else cout << "NULL ";
}
```

Figure 4.30 A dictionary search example. Here, payroll records are stored in two dictionaries, one organized by ID and the other organized by name. Both dictionaries are implemented with an unsorted array-based list.

The fundamental operation for a dictionary is finding a record that matches a given key. This raises the issue of how to extract the key from a record. We would like any given dictionary implementation to support arbitrary record types, so we need some mechanism for extracting keys that is sufficiently general. One approach is to require all record types to support some particular method that returns the key value. For example, in Java the **`Comparable`** interface can be used to provide this effect. Unfortunately, this approach does not work when the same record type is meant to be stored in multiple dictionaries, each keyed by a different field of the record. This is typical in database applications. Another, more general approach is to supply a class whose job is to extract the key from the record. Unfortunately, this solution also does not work in all situations, because there are record types for which it is not possible to write a key extraction method.[2]

[2]One example of such a situation occurs when we have a collection of records that describe books in a library. One of the fields for such a record might be a list of subject keywords, where the typical record stores a few keywords. Our dictionary might be implemented as a list of records sorted by keyword. If a book contains three keywords, it would appear three times on the list, once for each associated keyword. However, given the record, there is no simple way to determine which keyword on the keyword list triggered this appearance of the record. Thus, we cannot write a function that extracts the key from such a record.

```
// Container for a key-value pair
template <typename Key, typename E>
class KVpair {
private:
  Key k;
  E e;
public:
  // Constructors
  KVpair() {}
  KVpair(Key kval, E eval)
    { k = kval; e = eval; }
  KVpair(const KVpair& o)  // Copy constructor
    { k = o.k; e = o.e; }

  void operator =(const KVpair& o) // Assignment operator
    { k = o.k; e = o.e; }

  // Data member access functions
  Key key() { return k; }
  void setKey(Key ink) { k = ink; }
  E value() { return e; }
};
```

Figure 4.31 Implementation for a class representing a key-value pair.

The fundamental issue is that the key value for a record is not an intrinsic property of the record's class, or of any field within the class. The key for a record is actually a property of the context in which the record is used.

A truly general alternative is to explicitly store the key associated with a given record, as a separate field in the dictionary. That is, each entry in the dictionary will contain both a record and its associated key. Such entries are known as key-value pairs. It is typical that storing the key explicitly duplicates some field in the record. However, keys tend to be much smaller than records, so this additional space overhead will not be great. A simple class for representing key-value pairs is shown in Figure 4.31. The **`insert`** method of the dictionary class supports the key-value pair implementation because it takes two parameters, a record and its associated key for that dictionary.

Now that we have defined the dictionary ADT and settled on the design approach of storing key-value pairs for our dictionary entries, we are ready to consider ways to implement it. Two possibilities would be to use an array-based or linked list. Figure 4.32 shows an implementation for the dictionary using an (unsorted) array-based list.

Examining class **`UALdict`** (UAL stands for "unsorted array-based list), we can easily see that **`insert`** is a constant-time operation, because it simply inserts the new record at the end of the list. However, **`find`**, and **`remove`** both require $\Theta(n)$ time in the average and worst cases, because we need to do a sequential search. Method **`remove`** in particular must touch every record in the list, because once the

```
// Dictionary implemented with an unsorted array-based list
template <typename Key, typename E>
class UALdict : public Dictionary<Key, E> {
private:
  AList<KVpair<Key,E> >* list;
public:
  UALdict(int size=defaultSize)   // Constructor
    { list = new AList<KVpair<Key,E> >(size); }
  ~UALdict() { delete list; }          // Destructor
  void clear() { list->clear(); }     // Reinitialize

  // Insert an element: append to list
  void insert(const Key&k, const E& e) {
    KVpair<Key,E> temp(k, e);
    list->append(temp);
  }

  // Use sequential search to find the element to remove
  E remove(const Key& k) {
    E temp = find(k); // "find" will set list position
    if(temp != NULL) list->remove();
    return temp;
  }

  E removeAny() { // Remove the last element
    Assert(size() != 0, "Dictionary is empty");
    list->moveToEnd();
    list->prev();
    KVpair<Key,E> e = list->remove();
    return e.value();
  }

  // Find "k" using sequential search
  E find(const Key& k) const {
    for(list->moveToStart();
        list->currPos() < list->length(); list->next()) {
      KVpair<Key,E> temp = list->getValue();
      if (k == temp.key())
        return temp.value();
    }
    return NULL; // "k" does not appear in dictionary
  }
```

Figure 4.32 A dictionary implemented with an unsorted array-based list.

```
  int size() // Return list size
    { return list->length(); }
};
```

Figure 4.32 (continued)

```
// Sorted array-based list
// Inherit from AList as a protected base class
template <typename Key, typename E>
class SAList: protected AList<KVpair<Key,E> > {
public:
  SAList(int size=defaultSize) :
    AList<KVpair<Key,E> >(size) {}

  ~SAList() {}                    // Destructor

  // Redefine insert function to keep values sorted
  void insert(KVpair<Key,E>& it) { // Insert at right
    KVpair<Key,E> curr;
    for (moveToStart(); currPos() < length(); next()) {
      curr = getValue();
      if(curr.key() > it.key())
        break;
    }
    AList<KVpair<Key,E> >::insert(it); // Do AList insert
  }

  // With the exception of append, all remaining methods are
  // exposed from AList. Append is not available to SAlist
  // class users since it has not been explicitly exposed.
  AList<KVpair<Key,E> >::clear;
  AList<KVpair<Key,E> >::remove;
  AList<KVpair<Key,E> >::moveToStart;
  AList<KVpair<Key,E> >::moveToEnd;
  AList<KVpair<Key,E> >::prev;
  AList<KVpair<Key,E> >::next;
  AList<KVpair<Key,E> >::length;
  AList<KVpair<Key,E> >::currPos;
  AList<KVpair<Key,E> >::moveToPos;
  AList<KVpair<Key,E> >::getValue;
};
```

Figure 4.33 An implementation for a sorted array-based list.

desired record is found, the remaining records must be shifted down in the list to fill the gap. Method **removeAny** removes the last record from the list, so this is a constant-time operation.

As an alternative, we could implement the dictionary using a linked list. The implementation would be quite similar to that shown in Figure 4.32, and the cost of the functions should be the same asymptotically.

Another alternative would be to implement the dictionary with a sorted list. The advantage of this approach would be that we might be able to speed up the **find** operation by using a binary search. To do so, first we must define a variation on the **List** ADT to support sorted lists.An implementation for the array-based sorted list is shown in Figure 4.33. A sorted list is somewhat different from an unsorted list in that it cannot permit the user to control where elements get inserted. Thus,

the **insert** method must be quite different in a sorted list than in an unsorted list. Likewise, the user cannot be permitted to append elements onto the list. For these reasons, a sorted list cannot be implemented with straightforward inheritance from the **List** ADT.

Class **SAList** (SAL stands for "sorted array-based list") does inherit from class **AList**; however it does so using class **AList** as a protected base class. This means that **SAList** has available for its use any member functions of **AList**, but those member functions are not necessarily available to the user of **SAList**. However, many of the **AList** member functions are useful to the **SALlist** user. Thus, most of the **AList** member functions are passed along directly to the **SAList** user without change. For example, the line

```
AList<KVpair<Key,E> >::remove;
```

provides **SAList**'s clients with access to the **remove** method of **AList**. However, the original **insert** method from class **AList** is replaced, and the **append** method of **AList** is kept hidden.

The dictionary ADT can easily be implemented from class **SAList**, as shown in Figure 4.34. Method **insert** for the dictionary simply calls the **insert** method of the sorted list. Method **find** uses a generalization of the binary search function originally shown in Section 3.5. The cost for **find** in a sorted list is $\Theta(\log n)$ for a list of length n. This is a great improvement over the cost of **find** in an unsorted list. Unfortunately, the cost of **insert** changes from constant time in the unsorted list to $\Theta(n)$ time in the sorted list. Whether the sorted list implementation for the dictionary ADT is more or less efficient than the unsorted list implementation depends on the relative number of **insert** and **find** operations to be performed. If many more **find** operations than **insert** operations are used, then it might be worth using a sorted list to implement the dictionary. In both cases, **remove** requires $\Theta(n)$ time in the worst and average cases. Even if we used binary search to cut down on the time to find the record prior to removal, we would still need to shift down the remaining records in the list to fill the gap left by the **remove** operation.

Given two keys, we have not properly addressed the issue of how to compare them. One possibility would be to simply use the basic **==**, **<=**, and **>=** operators built into C++. This is the approach taken by our implementations for dictionaries shown inFigures 4.32 and 4.34. If the key type is **int**, for example, this will work fine. However, if the key is a pointer to a string or any other type of object, then this will not give the desired result. When we compare two strings we probably want to know which comes first in alphabetical order, but what we will get from the standard comparison operators is simply which object appears first in memory. Unfortunately, the code will compile fine, but the answers probably will not be fine.

In a language like C++ that supports operator overloading, we could require that the user of the dictionary overload the **==**, **<=**, and **>=** operators for the given

```
// Dictionary implemented with a sorted array-based list
template <typename Key, typename E>
class SALdict : public Dictionary<Key, E> {
private:
  SAList<Key,E>* list;
public:
  SALdict(int size=defaultSize)   // Constructor
    { list = new SAList<Key,E>(size); }
  ~SALdict() { delete list; }         // Destructor
  void clear() { list->clear(); }     // Reinitialize

  // Insert an element: Keep elements sorted
  void insert(const Key&k, const E& e) {
    KVpair<Key,E> temp(k, e);
    list->insert(temp);
  }

  // Use sequential search to find the element to remove
  E remove(const Key& k) {
    E temp = find(k);
    if (temp != NULL) list->remove();
    return temp;
  }

  E removeAny() { // Remove the last element
    Assert(size() != 0, "Dictionary is empty");
    list->moveToEnd();
    list->prev();
    KVpair<Key,E> e = list->remove();
    return e.value();
  }

  // Find "K" using binary search
  E find(const Key& k) const {
    int l = -1;
    int r = list->length();
    while (l+1 != r) { // Stop when l and r meet
      int i = (l+r)/2; // Check middle of remaining subarray
      list->moveToPos(i);
      KVpair<Key,E> temp = list->getValue();
      if (k < temp.key()) r = i;        // In left
      if (k == temp.key()) return temp.value(); // Found it
      if (k > temp.key()) l = i;        // In right
    }
    return NULL; // "k" does not appear in dictionary
  }
```

Figure 4.34 Dictionary implementation using a sorted array-based list.

```
  int size() // Return list size
    { return list->length(); }
};
```

Figure 4.34 (continued)

key type. This requirement then becomes an obligation on the user of the dictionary class. Unfortunately, this obligation is hidden within the code of the dictionary (and possibly in the user's manual) rather than exposed in the dictionary's interface. As a result, some users of the dictionary might neglect to implement the overloading, with unexpected results. Again, the compiler will not catch this problem.

The most general solution is to have users supply their own definition for comparing keys. The concept of a class that does comparison (called a **comparator**) is quite important. By making these operations be template parameters, the requirement to supply the comparator class becomes part of the interface. This design is an example of the Strategy design pattern, because the "strategies" for comparing and getting keys from records are provided by the client. In some cases, it makes sense for the comparator class to extract the key from the record type, as an alternative to storing key-value pairs.

Here is an example of the required class for comparing two integers.

```
class intintCompare { // Comparator class for integer keys
public:
  static bool lt(int x, int y) { return x < y; }
  static bool eq(int x, int y) { return x == y; }
  static bool gt(int x, int y) { return x > y; }
};
```

Class **intintCompare** provides methods for determining if two **int** variables are equal (**eq**), or if the first is less than the second (**lt**), or greater than the second (**gt**).

Here is a class for comparing two C-style character strings. It makes use of the standard library function **strcmp** to do the actual comparison.

```
class CCCompare { // Compare two character strings
public:
  static bool lt(char* x, char* y)
    { return strcmp(x, y) < 0; }
  static bool eq(char* x, char* y)
    { return strcmp(x, y) == 0; }
  static bool gt(char* x, char* y)
    { return strcmp(x, y) > 0; }
};
```

We will usea comparator in Section 5.5 to implement comparison in heaps, and in Chapter 7 to implement comparison in sorting algorithms.

4.5 Further Reading

For more discussion on choice of functions used to define the **`List`** ADT, see the work of the Reusable Software Research Group from Ohio State. Their definition for the **`List`** ADT can be found in [SWH93]. More information about designing such classes can be found in [SW94].

4.6 Exercises

4.1 Assume a list has the following configuration:

$$\langle \mid 2,\ 23,\ 15,\ 5,\ 9 \rangle.$$

Write a series of C++ statements using the **`List`** ADT of Figure 4.1 to delete the element with value 15.

4.2 Show the list configuration resulting from each series of list operations using the **`List`** ADT of Figure 4.1. Assume that lists **`L1`** and **`L2`** are empty at the beginning of each series. Show where the current position is in the list.

(a)
```
L1.append(10);
L1.append(20);
L1.append(15);
```

(b)
```
L2.append(10);
L2.append(20);
L2.append(15);
L2.moveToStart();
L2.insert(39);
L2.next();
L2.insert(12);
```

4.3 Write a series of C++ statements that uses the **`List`** ADT of Figure 4.1 to create a list capable of holding twenty elements and which actually stores the list with the following configuration:

$$\langle\ 2,\ 23 \mid 15,\ 5,\ 9\ \rangle.$$

4.4 Using the list ADT of Figure 4.1, write a function to interchange the current element and the one following it.

4.5 In the linked list implementation presented in Section 4.1.2, the current position is implemented using a pointer to the element ahead of the logical current node. The more "natural" approach might seem to be to have **`curr`** point directly to the node containing the current element. However, if this was done, then the pointer of the node preceding the current one cannot be

updated properly because there is no access to this node from **curr**. An alternative is to add a new node *after* the current element, copy the value of the current element to this new node, and then insert the new value into the old current node.

(a) What happens if **curr** is at the end of the list already? Is there still a way to make this work? Is the resulting code simpler or more complex than the implementation of Section 4.1.2?

(b) Will deletion always work in constant time if **curr** points directly to the current node? In particular, can you make several deletions in a row?

4.6 Add to the **LList** class implementation a member function to reverse the order of the elements on the list. Your algorithm should run in $\Theta(n)$ time for a list of n elements.

4.7 Write a function to merge two linked lists. The input lists have their elements in sorted order, from lowest to highest. The output list should also be sorted from lowest to highest. Your algorithm should run in linear time on the length of the output list.

4.8 A **circular linked list** is one in which the **next** field for the last link node of the list points to the first link node of the list. This can be useful when you wish to have a relative positioning for elements, but no concept of an absolute first or last position.

(a) Modify the code of Figure 4.8 to implement circular singly linked lists.

(b) Modify the code of Figure 4.14 to implement circular doubly linked lists.

4.9 Section 4.1.3 states "the space required by the array-based list implementation is $\Omega(n)$, but can be greater." Explain why this is so.

4.10 Section 4.1.3 presents an equation for determining the break-even point for the space requirements of two implementations of lists. The variables are D, E, P, and n. What are the dimensional units for each variable? Show that both sides of the equation balance in terms of their dimensional units.

4.11 Use the space equation of Section 4.1.3 to determine the break-even point for an array-based list and linked list implementation for lists when the sizes for the data field, a pointer, and the array-based list's array are as specified. State when the linked list needs less space than the array.

(a) The data field is eight bytes, a pointer is four bytes, and the array holds twenty elements.

(b) The data field is two bytes, a pointer is four bytes, and the array holds thirty elements.

(c) The data field is one byte, a pointer is four bytes, and the array holds thirty elements.

(d) The data field is 32 bytes, a pointer is four bytes, and the array holds forty elements.

4.12 Determine the size of an **`int`** variable, a **`double`** variable, and a pointer on your computer. (The **C**++ operator **`sizeof`** might be useful here if you do not already know the answer.)

(a) Calculate the break-even point, as a function of n, beyond which the array-based list is more space efficient than the linked list for lists whose elements are of type **`int`**.

(b) Calculate the break-even point, as a function of n, beyond which the array-based list is more space efficient than the linked list for lists whose elements are of type **`double`**.

4.13 Modify the code of Figure 4.18 to implement two stacks sharing the same array, as shown in Figure 4.20.

4.14 Modify the array-based queue definition of Figure 4.26 to use a separate Boolean member to keep track of whether the queue is empty, rather than require that one array position remain empty.

4.15 A **palindrome** is a string that reads the same forwards as backwards. Using only a fixed number of stacks and queues, the stack and queue ADT functions, and a fixed number of **`int`** and **`char`** variables, write an algorithm to determine if a string is a palindrome. Assume that the string is read from standard input one character at a time. The algorithm should output **`true`** or **`false`** as appropriate.

4.16 Re-implement function **`fibr`** from Exercise 2.11, using a stack to replace the recursive call as described in Section 4.2.4.

4.17 Write a recursive algorithm to compute the value of the recurrence relation

$$\mathbf{T}(n) = \mathbf{T}(\lceil n/2 \rceil) + \mathbf{T}(\lfloor n/2 \rfloor) + n; \quad \mathbf{T}(1) = 1.$$

Then, rewrite your algorithm to simulate the recursive calls with a stack.

4.18 Let Q be a non-empty queue, and let S be an empty stack. Using only the stack and queue ADT functions and a single element variable X, write an algorithm to reverse the order of the elements in Q.

4.19 A common problem for compilers and text editors is to determine if the parentheses (or other brackets) in a string are balanced and properly nested. For example, the string "((())())()" contains properly nested pairs of parentheses, but the string ")()(" does not, and the string "())" does not contain properly matching parentheses.

(a) Give an algorithm that returns **`true`** if a string contains properly nested and balanced parentheses, and **`false`** otherwise. Use a stack to keep track of the number of left parentheses seen so far. *Hint*: At no time while scanning a legal string from left to right will you have encountered more right parentheses than left parentheses.

(b) Give an algorithm that returns the position in the string of the first offending parenthesis if the string is not properly nested and balanced. That is, if an excess right parenthesis is found, return its position; if there are too many left parentheses, return the position of the first excess left parenthesis. Return -1 if the string is properly balanced and nested. Use a stack to keep track of the number and positions of left parentheses seen so far.

4.20 Imagine that you are designing an application where you need to perform the operations **`Insert`**, **`Delete_Maximum`**, and **`Delete_Minimum`**. For this application, the cost of inserting is not important, because it can be done off-line prior to startup of the time-critical section, but the performance of the two deletion operations are critical. Repeated deletions of either kind must work as fast as possible. Suggest a data structure that can support this application, and justify your suggestion. What is the time complexity for each of the three key operations?

4.21 Write a function that reverses the order of an array of n items.

4.7 Projects

4.1 A **deque** (pronounced "deck") is like a queue, except that items may be added and removed from both the front and the rear. Write either an array-based or linked implementation for the deque.

4.2 One solution to the problem of running out of space for an array-based list implementation is to replace the array with a larger array whenever the original array overflows. A good rule that leads to an implementation that is both space and time efficient is to double the current size of the array when there is an overflow. Re-implement the array-based **`List`** class of Figure 4.2 to support this array-doubling rule.

4.3 Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers. What is the asymptotic running time for each of your operations, expressed in terms of the number of digits for the two operands of each function?

4.4 Implement doubly linked lists by storing the sum of the **`next`** and **`prev`** pointers in a single pointer variable as described in Example 4.1.

4.5 Implement a city database using unordered lists. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x and y coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. Another operation that should be supported is to

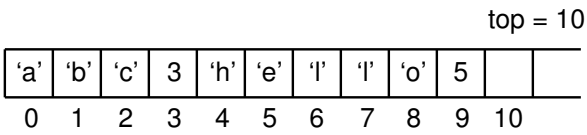


Figure 4.35 An array-based stack storing variable-length strings. Each position stores either one character or the length of the string immediately to the left of it in the stack.

print all records within a given distance of a specified point. Implement the database using an array-based list implementation, and then a linked list implementation. Collect running time statistics for each operation in both implementations. What are your conclusions about the relative advantages and disadvantages of the two implementations? Would storing records on the list in alphabetical order by city name speed any of the operations? Would keeping the list in alphabetical order slow any of the operations?

4.6 Modify the code of Figure 4.18 to support storing variable-length strings of at most 255 characters. The stack array should have type **`char`**. A string is represented by a series of characters (one character per stack element), with the length of the string stored in the stack element immediately above the string itself, as illustrated by Figure 4.35. The **`push`** operation would store an element requiring i storage units in the i positions beginning with the current value of **`top`** and store the size in the position i storage units above **`top`**. The value of **`top`** would then be reset above the newly inserted element. The **`pop`** operation need only look at the size value stored in position `top` -1 and then pop off the appropriate number of units. You may store the string on the stack in reverse order if you prefer, provided that when it is popped from the stack, it is returned in its proper order.

4.7 Define an ADT for a bag (see Section 2.1) and create an array-based implementation for bags. Be sure that your bag ADT does not rely in any way on knowing or controlling the position of an element. Then, implement the dictionary ADT of Figure 4.28 using your bag implementation.

4.8 Implement the dictionary ADT of Figure 4.28 using an unsorted linked list as defined by class **`LList`** in Figure 4.8. Make the implementation as efficient as you can, given the restriction that your implementation must use the unsorted linked list and its access operations to implement the dictionary. State the asymptotic time requirements for each function member of the dictionary ADT under your implementation.

4.9 Implement the dictionary ADT of Figure 4.28 based on stacks. Your implementation should declare and use two stacks.

4.10 Implement the dictionary ADT of Figure 4.28 based on queues. Your implementation should declare and use two queues.

5

Binary Trees

The list representations of Chapter 4 have a fundamental limitation: Either search or insert can be made efficient, but not both at the same time. Tree structures permit both efficient access and update to large collections of data. Binary trees in particular are widely used and relatively easy to implement. But binary trees are useful for many things besides searching. Just a few examples of applications that trees can speed up include prioritizing jobs, describing mathematical expressions and the syntactic elements of computer programs, or organizing the information needed to drive data compression algorithms.

This chapter begins by presenting definitions and some key properties of binary trees. Section 5.2 discusses how to process all nodes of the binary tree in an organized manner. Section 5.3 presents various methods for implementing binary trees and their nodes. Sections 5.4 through 5.6 present three examples of binary trees used in specific applications: the Binary Search Tree (BST) for implementing dictionaries, heaps for implementing priority queues, and Huffman coding trees for text compression. The BST, heap, and Huffman coding tree each have distinctive structural features that affect their implementation and use.

5.1 Definitions and Properties

A **binary tree** is made up of a finite set of elements called **nodes**. This set either is empty or consists of a node called the **root** together with two binary trees, called the left and right **subtrees**, which are disjoint from each other and from the root. (Disjoint means that they have no nodes in common.) The roots of these subtrees are **children** of the root. There is an **edge** from a node to each of its children, and a node is said to be the **parent** of its children.

If n_1, n_2, ..., n_k is a sequence of nodes in the tree such that n_i is the parent of n_{i+1} for $1 \leq i < k$, then this sequence is called a **path** from n_1 to n_k. The **length** of the path is $k - 1$. If there is a path from node *R* to node *M*, then *R* is an **ancestor** of *M*, and *M* is a **descendant** of *R*. Thus, all nodes in the tree are descendants of the

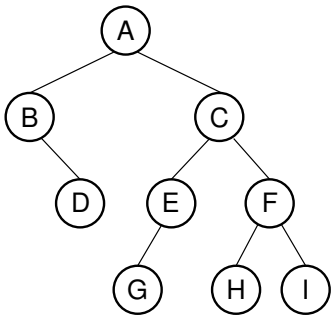


Figure 5.1 A binary tree. Node *A* is the root. Nodes *B* and *C* are *A*'s children. Nodes *B* and *D* together form a subtree. Node *B* has two children: Its left child is the empty tree and its right child is *D*. Nodes *A*, *C*, and *E* are ancestors of *G*. Nodes *D*, *E*, and *F* make up level 2 of the tree; node *A* is at level 0. The edges from *A* to *C* to *E* to *G* form a path of length 3. Nodes *D*, *G*, *H*, and *I* are leaves. Nodes *A*, *B*, *C*, *E*, and *F* are internal nodes. The depth of *I* is 3. The height of this tree is 4.

root of the tree, while the root is the ancestor of all nodes. The **depth** of a node *M* in the tree is the length of the path from the root of the tree to *M*. The **height** of a tree is one more than the depth of the deepest node in the tree. All nodes of depth d are at **level** d in the tree. The root is the only node at level 0, and its depth is 0. A **leaf** node is any node that has two empty children. An **internal** node is any node that has at least one non-empty child.

Figure 5.1 illustrates the various terms used to identify parts of a binary tree. Figure 5.2 illustrates an important point regarding the structure of binary trees. Because *all* binary tree nodes have two children (one or both of which might be empty), the two binary trees of Figure 5.2 are *not* the same.

Two restricted forms of binary tree are sufficiently important to warrant special names. Each node in a **full** binary tree is either (1) an internal node with exactly two non-empty children or (2) a leaf. A **complete** binary tree has a restricted shape obtained by starting at the root and filling the tree by levels from left to right. In the complete binary tree of height d, all levels except possibly level $d-1$ are completely full. The bottom level has its nodes filled in from the left side.

Figure 5.3 illustrates the differences between full and complete binary trees.[1] There is no particular relationship between these two tree shapes; that is, the tree of Figure 5.3(a) is full but not complete while the tree of Figure 5.3(b) is complete but

[1] While these definitions for full and complete binary tree are the ones most commonly used, they are not universal. Because the common meaning of the words "full" and "complete" are quite similar, there is little that you can do to distinguish between them other than to memorize the definitions. Here is a memory aid that you might find useful: "Complete" is a wider word than "full," and complete binary trees tend to be wider than full binary trees because each level of a complete binary tree is as wide as possible.

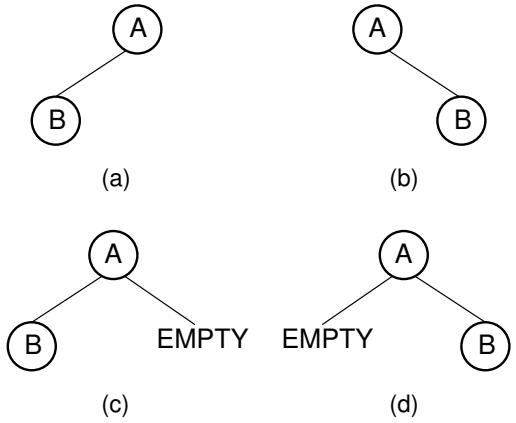


Figure 5.2 Two different binary trees. (a) A binary tree whose root has a non-empty left child. (b) A binary tree whose root has a non-empty right child. (c) The binary tree of (a) with the missing right child made explicit. (d) The binary tree of (b) with the missing left child made explicit.

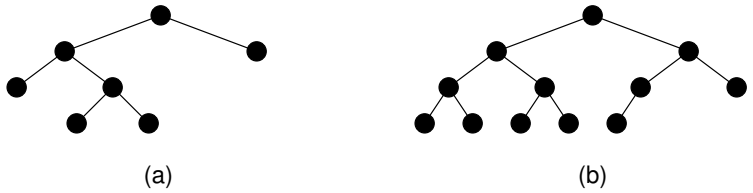


Figure 5.3 Examples of full and complete binary trees. (a) This tree is full (but not complete). (b) This tree is complete (but not full).

not full. The heap data structure (Section 5.5) is an example of a complete binary tree. The Huffman coding tree (Section 5.6) is an example of a full binary tree.

5.1.1 The Full Binary Tree Theorem

Some binary tree implementations store data only at the leaf nodes, using the internal nodes to provide structure to the tree. More generally, binary tree implementations might require some amount of space for internal nodes, and a different amount for leaf nodes. Thus, to analyze the space required by such implementations, it is useful to know the minimum and maximum fraction of the nodes that are leaves in a tree containing n internal nodes.

Unfortunately, this fraction is not fixed. A binary tree of n internal nodes might have only one leaf. This occurs when the internal nodes are arranged in a chain ending in a single leaf as shown in Figure 5.4. In this case, the number of leaves is low because each internal node has only one non-empty child. To find an upper bound on the number of leaves for a tree of n internal nodes, first note that the upper

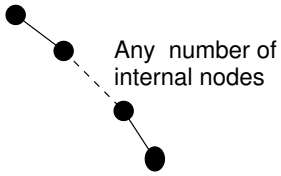


Figure 5.4 A tree containing many internal nodes and a single leaf.

bound will occur when each internal node has two non-empty children, that is, when the tree is full. However, this observation does not tell what shape of tree will yield the highest percentage of non-empty leaves. It turns out not to matter, because all full binary trees with n internal nodes have the same number of leaves. This fact allows us to compute the space requirements for a full binary tree implementation whose leaves require a different amount of space from its internal nodes.

Theorem 5.1 Full Binary Tree Theorem: *The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.*

Proof: The proof is by mathematical induction on n, the number of internal nodes. This is an example of an induction proof where we reduce from an arbitrary instance of size n to an instance of size $n-1$ that meets the induction hypothesis.

- **Base Cases**: The non-empty tree with zero internal nodes has one leaf node. A full binary tree with one internal node has two leaf nodes. Thus, the base cases for $n = 0$ and $n = 1$ conform to the theorem.
- **Induction Hypothesis**: Assume that any full binary tree **T** containing $n-1$ internal nodes has n leaves.
- **Induction Step**: Given tree **T** with n internal nodes, select an internal node *I* whose children are both leaf nodes. Remove both of *I*'s children, making *I* a leaf node. Call the new tree **T**′. **T**′ has $n-1$ internal nodes. From the induction hypothesis, **T**′ has n leaves. Now, restore *I*'s two children. We once again have tree **T** with n internal nodes. How many leaves does **T** have? Because **T**′ has n leaves, adding the two children yields $n+2$. However, node *I* counted as one of the leaves in **T**′ and has now become an internal node. Thus, tree **T** has $n+1$ leaf nodes and n internal nodes.

By mathematical induction the theorem holds for all values of $n \geq 0$. □

When analyzing the space requirements for a binary tree implementation, it is useful to know how many empty subtrees a tree contains. A simple extension of the Full Binary Tree Theorem tells us exactly how many empty subtrees there are in *any* binary tree, whether full or not. Here are two approaches to proving the following theorem, and each suggests a useful way of thinking about binary trees.

Theorem 5.2 *The number of empty subtrees in a non-empty binary tree is one more than the number of nodes in the tree.*

Proof 1: Take an arbitrary binary tree **T** and replace every empty subtree with a leaf node. Call the new tree **T′**. All nodes originally in **T** will be internal nodes in **T′** (because even the leaf nodes of **T** have children in **T′**). **T′** is a full binary tree, because every internal node of **T** now must have two children in **T′**, and each leaf node in **T** must have two children in **T′** (the leaves just added). The Full Binary Tree Theorem tells us that the number of leaves in a full binary tree is one more than the number of internal nodes. Thus, the number of new leaves that were added to create **T′** is one more than the number of nodes in **T**. Each leaf node in **T′** corresponds to an empty subtree in **T**. Thus, the number of empty subtrees in **T** is one more than the number of nodes in **T**. □

Proof 2: By definition, every node in binary tree **T** has two children, for a total of $2n$ children in a tree of n nodes. Every node except the root node has one parent, for a total of $n - 1$ nodes with parents. In other words, there are $n - 1$ non-empty children. Because the total number of children is $2n$, the remaining $n + 1$ children must be empty. □

5.1.2 A Binary Tree Node ADT

Just as a linked list is comprised of a collection of link objects, a tree is comprised of a collection of node objects. Figure 5.5 shows an ADT for binary tree nodes, called **`BinNode`**. This class will be used by some of the binary tree structures presented later. Class **`BinNode`** is a template with parameter **`E`**, which is the type for the data record stored in the node. Member functions are provided that set or return the element value, set or return a pointer to the left child, set or return a pointer to the right child, or indicate whether the node is a leaf.

5.2 Binary Tree Traversals

Often we wish to process a binary tree by "visiting" each of its nodes, each time performing a specific action such as printing the contents of the node. Any process for visiting all of the nodes in some order is called a **traversal**. Any traversal that lists every node in the tree exactly once is called an **enumeration** of the tree's nodes. Some applications do not require that the nodes be visited in any particular order as long as each node is visited precisely once. For other applications, nodes must be visited in an order that preserves some relationship. For example, we might wish to make sure that we visit any given node *before* we visit its children. This is called a **preorder traversal**.

```
// Binary tree node abstract class
template <typename E> class BinNode {
public:
  virtual ~BinNode() {} // Base destructor

  // Return the node's value
  virtual E& element() = 0;

  // Set the node's value
  virtual void setElement(const E&) = 0;

  // Return the node's left child
  virtual BinNode* left() const = 0;

  // Set the node's left child
  virtual void setLeft(BinNode*) = 0;

  // Return the node's right child
  virtual BinNode* right() const = 0;

  // Set the node's right child
  virtual void setRight(BinNode*) = 0;

  // Return true if the node is a leaf, false otherwise
  virtual bool isLeaf() = 0;
};
```

Figure 5.5 A binary tree node ADT.

Example 5.1 The preorder enumeration for the tree of Figure 5.1 is

ABDCEGFHI.

The first node printed is the root. Then all nodes of the left subtree are printed (in preorder) before any node of the right subtree.

Alternatively, we might wish to visit each node only *after* we visit its children (and their subtrees). For example, this would be necessary if we wish to return all nodes in the tree to free store. We would like to delete the children of a node before deleting the node itself. But to do that requires that the children's children be deleted first, and so on. This is called a **postorder traversal**.

Example 5.2 The postorder enumeration for the tree of Figure 5.1 is

DBGEHIFCA.

An **inorder traversal** first visits the left child (including its entire subtree), then visits the node, and finally visits the right child (including its entire subtree). The

binary search tree of Section 5.4 makes use of this traversal to print all nodes in ascending order of value.

Example 5.3 The inorder enumeration for the tree of Figure 5.1 is

BDAGECHFI.

A traversal routine is naturally written as a recursive function. Its input parameter is a pointer to a node which we will call **root** because each node can be viewed as the root of a some subtree. The initial call to the traversal function passes in a pointer to the root node of the tree. The traversal function visits **root** and its children (if any) in the desired order. For example, a preorder traversal specifies that **root** be visited before its children. This can easily be implemented as follows.

```
template <typename E>
void preorder(BinNode<E>* root) {
  if (root == NULL) return; // Empty subtree, do nothing
  visit(root);              // Perform desired action
  preorder(root->left());
  preorder(root->right());
}
```

Function **preorder** first checks that the tree is not empty (if it is, then the traversal is done and **preorder** simply returns). Otherwise, **preorder** makes a call to **visit**, which processes the root node (i.e., prints the value or performs whatever computation as required by the application). Function **preorder** is then called recursively on the left subtree, which will visit all nodes in that subtree. Finally, **preorder** is called on the right subtree, visiting all nodes in the right subtree. Postorder and inorder traversals are similar. They simply change the order in which the node and its children are visited, as appropriate.

An important decision in the implementation of any recursive function on trees is when to check for an empty subtree. Function **preorder** first checks to see if the value for **root** is **NULL**. If not, it will recursively call itself on the left and right children of **root**. In other words, **preorder** makes no attempt to avoid calling itself on an empty child. Some programmers use an alternate design in which the left and right pointers of the current node are checked so that the recursive call is made only on non-empty children. Such a design typically looks as follows:

```
template <typename E>
void preorder2(BinNode<E>* root) {
  visit(root);  // Perform whatever action is desired
  if (root->left() != NULL) preorder2(root->left());
  if (root->right() != NULL) preorder2(root->right());
}
```

At first it might appear that **preorder2** is more efficient than **preorder**, because it makes only half as many recursive calls. (Why?) On the other hand, **preorder2** must access the left and right child pointers twice as often. The net result is little or no performance improvement.

In reality, the design of **preorder2** is inferior to that of **preorder** for two reasons. First, while it is not apparent in this simple example, for more complex traversals it can become awkward to place the check for the **NULL** pointer in the calling code. Even here we had to write two tests for **NULL**, rather than the one needed by **preorder**. The more important concern with **preorder2** is that it tends to be error prone. While **preorder2** insures that no recursive calls will be made on empty subtrees, it will fail if the initial call passes in a **NULL** pointer. This would occur if the original tree is empty. To avoid the bug, either **preorder2** needs an additional test for a **NULL** pointer at the beginning (making the subsequent tests redundant after all), or the caller of **preorder2** has a hidden obligation to pass in a non-empty tree, which is unreliable design. The net result is that many programmers forget to test for the possibility that the empty tree is being traversed. By using the first design, which explicitly supports processing of empty subtrees, the problem is avoided.

Another issue to consider when designing a traversal is how to define the visitor function that is to be executed on every node. One approach is simply to write a new version of the traversal for each such visitor function as needed. The disadvantage to this is that whatever function does the traversal must have access to the **BinNode** class. It is probably better design to permit only the tree class to have access to the **BinNode** class.

Another approach is for the tree class to supply a generic traversal function which takes the visitor either as a template parameter or as a function parameter. This is known as the **visitor design pattern**. A major constraint on this approach is that the **signature** for all visitor functions, that is, their return type and parameters, must be fixed in advance. Thus, the designer of the generic traversal function must be able to adequately judge what parameters and return type will likely be needed by potential visitor functions.

Handling information flow between parts of a program can be a significant design challenge, especially when dealing with recursive functions such as tree traversals. In general, we can run into trouble either with passing in the correct information needed by the function to do its work, or with returning information to the recursive function's caller. We will see many examples throughout the book that illustrate methods for passing information in and out of recursive functions as they traverse a tree structure. Here are a few simple examples.

First we consider the simple case where a computation requires that we communicate information back up the tree to the end user.

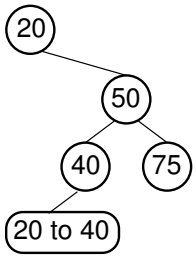


Figure 5.6 To be a binary search tree, the left child of the node with value 40 must have a value between 20 and 40.

Example 5.4 We wish to count the number of nodes in a binary tree. The key insight is that the total count for any (non-empty) subtree is one for the root plus the counts for the left and right subtrees. Where do left and right subtree counts come from? Calls to function **count** on the subtrees will compute this for us. Thus, we can implement **count** as follows.

```
template <typename E>
int count(BinNode<E>* root) {
  if (root == NULL) return 0;  // Nothing to count
  return 1 + count(root->left())
           + count(root->right());
}
```

Another problem that occurs when recursively processing data collections is controlling which members of the collection will be visited. For example, some tree "traversals" might in fact visit only some tree nodes, while avoiding processing of others. Exercise 5.20 must solve exactly this problem in the context of a binary search tree. It must visit only those children of a given node that might possibly fall within a given range of values. Fortunately, it requires only a simple local calculation to determine which child(ren) to visit.

A more difficult situation is illustrated by the following problem. Given an arbitrary binary tree we wish to determine if, for every node A, are all nodes in A's left subtree less than the value of A, and are all nodes in A's right subtree greater than the value of A? (This happens to be the definition for a binary search tree, described in Section 5.4.) Unfortunately, to make this decision we need to know some context that is not available just by looking at the node's parent or children. As shown by Figure 5.6, it is not enough to verify that A's left child has a value less than that of A, and that A's right child has a greater value. Nor is it enough to verify that A has a value consistent with that of its parent. In fact, we need to know information about what range of values is legal for a given node. That information might come from any of the node's ancestors. Thus, relevant range information must be passed down the tree. We can implement this function as follows.

```
template <typename Key, typename E>
bool checkBST(BSTNode<Key,E>* root, Key low, Key high) {
  if (root == NULL) return true; // Empty subtree
  Key rootkey = root->key();
  if ((rootkey < low) || (rootkey > high))
    return false; // Out of range
  if (!checkBST<Key,E>(root->left(), low, rootkey))
    return false; // Left side failed
  return checkBST<Key,E>(root->right(), rootkey, high);
}
```

5.3 Binary Tree Node Implementations

In this section we examine ways to implement binary tree nodes. We begin with some options for pointer-based binary tree node implementations. Then comes a discussion on techniques for determining the space requirements for a given implementation. The section concludes with an introduction to the array-based implementation for complete binary trees.

5.3.1 Pointer-Based Node Implementations

By definition, all binary tree nodes have two children, though one or both children can be empty. Binary tree nodes typically contain a value field, with the type of the field depending on the application. The most common node implementation includes a value field and pointers to the two children.

Figure 5.7 shows a simple implementation for the **BinNode** abstract class, which we will name **BSTNode**. Class **BSTNode** includes a data member of type **E**, (which is the second template parameter) for the element type. To support search structures such as the Binary Search Tree, an additional field is included, with corresponding access methods, to store a key value (whose purpose is explained in Section 4.4). Its type is determined by the first template parameter, named **Key**. Every **BSTNode** object also has two pointers, one to its left child and another to its right child. Overloaded **new** and **delete** operators could be added to support a freelist, as described in Section 4.1.2.Figure 5.8 illustrates the **BSTNode** implementation.

Some programmers find it convenient to add a pointer to the node's parent, allowing easy upward movement in the tree. Using a parent pointer is somewhat analogous to adding a link to the previous node in a doubly linked list. In practice, the parent pointer is almost always unnecessary and adds to the space overhead for the tree implementation. It is not just a problem that parent pointers take space. More importantly, many uses of the parent pointer are driven by improper understanding of recursion and so indicate poor programming. If you are inclined toward using a parent pointer, consider if there is a more efficient implementation possible.

```
// Simple binary tree node implementation
template <typename Key, typename E>
class BSTNode : public BinNode<E> {
private:
  Key k;                    // The node's key
  E it;                     // The node's value
  BSTNode* lc;              // Pointer to left child
  BSTNode* rc;              // Pointer to right child

public:
  // Two constructors -- with and without initial values
  BSTNode() { lc = rc = NULL; }
  BSTNode(Key K, E e, BSTNode* l =NULL, BSTNode* r =NULL)
    { k = K; it = e; lc = l; rc = r; }
  ~BSTNode() {}             // Destructor

  // Functions to set and return the value and key
  E& element() { return it; }
  void setElement(const E& e) { it = e; }
  Key& key() { return k; }
  void setKey(const Key& K) { k = K; }

  // Functions to set and return the children
  inline BSTNode* left() const { return lc; }
  void setLeft(BinNode<E>* b) { lc = (BSTNode*)b; }
  inline BSTNode* right() const { return rc; }
  void setRight(BinNode<E>* b) { rc = (BSTNode*)b; }

  // Return true if it is a leaf, false otherwise
  bool isLeaf() { return (lc == NULL) && (rc == NULL); }
};
```

Figure 5.7 A binary tree node class implementation.

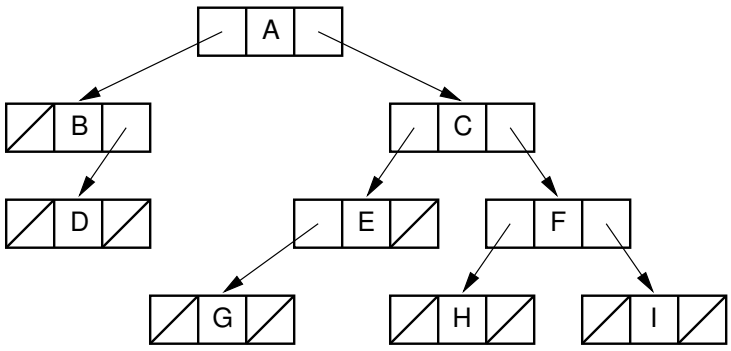


Figure 5.8 Illustration of a typical pointer-based binary tree implementation, where each node stores two child pointers and a value.

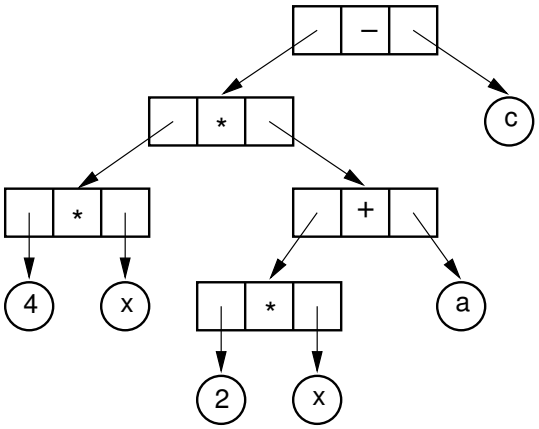


Figure 5.9 An expression tree for $4x(2x + a) - c$.

An important decision in the design of a pointer-based node implementation is whether the same class definition will be used for leaves and internal nodes. Using the same class for both will simplify the implementation, but might be an inefficient use of space. Some applications require data values only for the leaves. Other applications require one type of value for the leaves and another for the internal nodes. Examples include the binary trie of Section 13.1, the PR quadtree of Section 13.3, the Huffman coding tree of Section 5.6, and the expression tree illustrated by Figure 5.9. By definition, only internal nodes have non-empty children. If we use the same node implementation for both internal and leaf nodes, then both must store the child pointers. But it seems wasteful to store child pointers in the leaf nodes. Thus, there are many reasons why it can save space to have separate implementations for internal and leaf nodes.

As an example of a tree that stores different information at the leaf and internal nodes, consider the expression tree illustrated by Figure 5.9. The expression tree represents an algebraic expression composed of binary operators such as addition, subtraction, multiplication, and division. Internal nodes store operators, while the leaves store operands. The tree of Figure 5.9 represents the expression $4x(2x + a) - c$. The storage requirements for a leaf in an expression tree are quite different from those of an internal node. Internal nodes store one of a small set of operators, so internal nodes could store a small code identifying the operator such as a single byte for the operator's character symbol. In contrast, leaves store variable names or numbers, which is considerably larger in order to handle the wider range of possible values. At the same time, leaf nodes need not store child pointers.

C++ allows us to differentiate leaf from internal nodes through the use of class inheritance. A **base class** provides a general definition for an object, and a **subclass** modifies a base class to add more detail. A base class can be declared for binary tree nodes in general, with subclasses defined for the internal and leaf nodes. The base class of Figure 5.10 is named **`VarBinNode`**. It includes a virtual member function

named **isLeaf**, which indicates the node type. Subclasses for the internal and leaf node types each implement **isLeaf**. Internal nodes store child pointers of the base class type; they do not distinguish their children's actual subclass. Whenever a node is examined, its version of **isLeaf** indicates the node's subclass.

Figure 5.10 includes two subclasses derived from class **VarBinNode**, named **LeafNode** and **IntlNode**. Class **IntlNode** can access its children through pointers of type **VarBinNode**. Function **traverse** illustrates the use of these classes. When **traverse** calls method **isLeaf**, C++'s runtime environment determines which subclass this particular instance of **rt** happens to be and calls that subclass's version of **isLeaf**. Method **isLeaf** then provides the actual node type to its caller. The other member functions for the derived subclasses are accessed by type-casting the base class pointer as appropriate, as shown in function **traverse**.

There is another approach that we can take to represent separate leaf and internal nodes, also using a virtual base class and separate node classes for the two types. This is to implement nodes using the **composite design pattern**. This approach is noticeably different from the one of Figure 5.10 in that the node classes themselves implement the functionality of **traverse**. Figure 5.11 shows the implementation. Here, base class **VarBinNode** declares a member function **traverse** that each subclass must implement. Each subclass then implements its own appropriate behavior for its role in a traversal. The whole traversal process is called by invoking **traverse** on the root node, which in turn invokes **traverse** on its children.

When comparing the implementations of Figures 5.10 and 5.11, each has advantages and disadvantages. The first does not require that the node classes know about the **traverse** function. With this approach, it is easy to add new methods to the tree class that do other traversals or other operations on nodes of the tree. However, we see that **traverse** in Figure 5.10 does need to be familiar with each node subclass. Adding a new node subclass would therefore require modifications to the **traverse** function. In contrast, the approach of Figure 5.11 requires that any new operation on the tree that requires a traversal also be implemented in the node subclasses. On the other hand, the approach of Figure 5.11 avoids the need for the **traverse** function to know anything about the distinct abilities of the node subclasses. Those subclasses handle the responsibility of performing a traversal on themselves. A secondary benefit is that there is no need for **traverse** to explicitly enumerate all of the different node subclasses, directing appropriate action for each. With only two node classes this is a minor point. But if there were many such subclasses, this could become a bigger problem. A disadvantage is that the traversal operation must not be called on a **NULL** pointer, because there is no object to catch the call. This problem could be avoided by using a flyweight (see Section 1.3.1) to implement empty nodes.

Typically, the version of Figure 5.10 would be preferred in this example if **traverse** is a member function of the tree class, and if the node subclasses are

```
// Node implementation with simple inheritance
class VarBinNode {   // Node abstract base class
public:
  virtual ~VarBinNode() {}
  virtual bool isLeaf() = 0;    // Subclasses must implement
};


class LeafNode : public VarBinNode { // Leaf node
private:
  Operand var;                       // Operand value

public:
  LeafNode(const Operand& val) { var = val; } // Constructor
  bool isLeaf() { return true; }     // Version for LeafNode
  Operand value() { return var; }    // Return node value
};


class IntlNode : public VarBinNode { // Internal node
private:
  VarBinNode* left;                  // Left child
  VarBinNode* right;                 // Right child
  Operator opx;                      // Operator value

public:
  IntlNode(const Operator& op, VarBinNode* l, VarBinNode* r)
    { opx = op; left = l; right = r; } // Constructor
  bool isLeaf() { return false; }    // Version for IntlNode
  VarBinNode* leftchild() { return left; }   // Left child
  VarBinNode* rightchild() { return right; } // Right child
  Operator value() { return opx; }           // Value
};


void traverse(VarBinNode *root) {    // Preorder traversal
  if (root == NULL) return;          // Nothing to visit
  if (root->isLeaf())                // Do leaf node
    cout << "Leaf: " << ((LeafNode *)root)->value() << endl;
  else {                             // Do internal node
    cout << "Internal: "
         << ((IntlNode *)root)->value() << endl;
    traverse(((IntlNode *)root)->leftchild());
    traverse(((IntlNode *)root)->rightchild());
  }
}
```

Figure 5.10 An implementation for separate internal and leaf node representations using C++ class inheritance and virtual functions.

```
// Node implementation with the composite design pattern
class VarBinNode {   // Node abstract base class
public:
  virtual ~VarBinNode() {}     // Generic destructor
  virtual bool isLeaf() = 0;
  virtual void traverse() = 0;
};


class LeafNode : public VarBinNode { // Leaf node
private:
  Operand var;                       // Operand value

public:
  LeafNode(const Operand& val) { var = val; } // Constructor
  bool isLeaf() { return true; }     // isLeaf for Leafnode
  Operand value() { return var; }    // Return node value
  void traverse() { cout << "Leaf: " << value() << endl; }
};


class IntlNode : public VarBinNode { // Internal node
private:
  VarBinNode* lc;                    // Left child
  VarBinNode* rc;                    // Right child
  Operator opx;                      // Operator value

public:
  IntlNode(const Operator& op, VarBinNode* l, VarBinNode* r)
    { opx = op; lc = l; rc = r; }    // Constructor

  bool isLeaf() { return false; }    // isLeaf for IntlNode
  VarBinNode* left() { return lc; }  // Left child
  VarBinNode* right() { return rc; } // Right child
  Operator value() { return opx; }   // Value

  void traverse() { // Traversal behavior for internal nodes
    cout << "Internal: " << value() << endl;
    if (left() != NULL) left()->traverse();
    if (right() != NULL) right()->traverse();
  }
};


// Do a preorder traversal
void traverse(VarBinNode *root) {
  if (root != NULL) root->traverse();
}
```

Figure 5.11 A second implementation for separate internal and leaf node representations using C++ class inheritance and virtual functions using the composite design pattern. Here, the functionality of **`traverse`** is embedded into the node subclasses.

hidden from users of that tree class. On the other hand, if the nodes are objects that have meaning to users of the tree separate from their existence as nodes in the tree, then the version of Figure 5.11 might be preferred because hiding the internal behavior of the nodes becomes more important.

Another advantage of the composite design is that implementing each node type's functionality might be easier. This is because you can focus solely on the information passing and other behavior needed by this node type to do its job. This breaks down the complexity that many programmers feel overwhelmed by when dealing with complex information flows related to recursive processing.

5.3.2 Space Requirements

This section presents techniques for calculating the amount of overhead required by a binary tree implementation. Recall that overhead is the amount of space necessary to maintain the data structure. In other words, it is any space not used to store data records. The amount of overhead depends on several factors including which nodes store data values (all nodes, or just the leaves), whether the leaves store child pointers, and whether the tree is a full binary tree.

In a simple pointer-based implementation for the binary tree such as that of Figure 5.7, every node has two pointers to its children (even when the children are **NULL**). This implementation requires total space amounting to $n(2P + D)$ for a tree of n nodes. Here, P stands for the amount of space required by a pointer, and D stands for the amount of space required by a data value. The total overhead space will be $2Pn$ for the entire tree. Thus, the overhead fraction will be $2P/(2P + D)$. The actual value for this expression depends on the relative size of pointers versus data fields. If we arbitrarily assume that $P = D$, then a full tree has about two thirds of its total space taken up in overhead. Worse yet, Theorem 5.2 tells us that about half of the pointers are "wasted" **NULL** values that serve only to indicate tree structure, but which do not provide access to new data.

A common implementation is not to store any actual data in a node, but rather a pointer to the data record. In this case, each node will typically store three pointers, all of which are overhead, resulting in an overhead fraction of $3P/(3P + D)$.

If only leaves store data values, then the fraction of total space devoted to overhead depends on whether the tree is full. If the tree is not full, then conceivably there might only be one leaf node at the end of a series of internal nodes. Thus, the overhead can be an arbitrarily high percentage for non-full binary trees. The overhead fraction drops as the tree becomes closer to full, being lowest when the tree is truly full. In this case, about one half of the nodes are internal.

Great savings can be had by eliminating the pointers from leaf nodes in full binary trees. Again assume the tree stores a pointer to the data field. Because about half of the nodes are leaves and half internal nodes, and because only internal nodes

now have child pointers, the overhead fraction in this case will be approximately

$$\frac{\frac{n}{2}(2P)}{\frac{n}{2}(2P) + Dn} = \frac{P}{P + D}.$$

If $P = D$, the overhead drops to about one half of the total space. However, if only leaf nodes store useful information, the overhead fraction for this implementation is actually three quarters of the total space, because half of the "data" space is unused.

If a full binary tree needs to store data only at the leaf nodes, a better implementation would have the internal nodes store two pointers and no data field while the leaf nodes store only a pointer to the data field. This implementation requires $\frac{n}{2}2P+\frac{n}{2}(p+d)$ units of space. If $P = D$, then the overhead is $3P/(3P+D) = 3/4$. It might seem counter-intuitive that the overhead ratio has gone up while the total amount of space has gone down. The reason is because we have changed our definition of "data" to refer only to what is stored in the leaf nodes, so while the overhead fraction is higher, it is from a total storage requirement that is lower.

There is one serious flaw with this analysis. When using separate implementations for internal and leaf nodes, there must be a way to distinguish between the node types. When separate node types are implemented via **C**++ subclasses, the runtime environment stores information with each object allowing it to determine, for example, the correct subclass to use when the **`isLeaf`** virtual function is called. Thus, each node requires additional space. Only one bit is truly necessary to distinguish the two possibilities. In rare applications where space is a critical resource, implementors can often find a spare bit within the node's value field in which to store the node type indicator. An alternative is to use a spare bit within a node pointer to indicate node type. For example, this is often possible when the compiler requires that structures and objects start on word boundaries, leaving the last bit of a pointer value always zero. Thus, this bit can be used to store the node-type flag and is reset to zero before the pointer is dereferenced. Another alternative when the leaf value field is smaller than a pointer is to replace the pointer to a leaf with that leaf's value. When space is limited, such techniques can make the difference between success and failure. In any other situation, such "bit packing" tricks should be avoided because they are difficult to debug and understand at best, and are often machine dependent at worst.[2]

[2]In the early to mid 1980s, I worked on a Geographic Information System that stored spatial data in quadtrees (see Section 13.3). At the time space was a critical resource, so we used a bit-packing approach where we stored the nodetype flag as the last bit in the parent node's pointer. This worked perfectly on various 32-bit workstations. Unfortunately, in those days IBM PC-compatibles used 16-bit pointers. We never did figure out how to port our code to the 16-bit machine.

5.3.3 Array Implementation for Complete Binary Trees

The previous section points out that a large fraction of the space in a typical binary tree node implementation is devoted to structural overhead, not to storing data. This section presents a simple, compact implementation for complete binary trees. Recall that complete binary trees have all levels except the bottom filled out completely, and the bottom level has all of its nodes filled in from left to right. Thus, a complete binary tree of n nodes has only one possible shape. You might think that a complete binary tree is such an unusual occurrence that there is no reason to develop a special implementation for it. However, the complete binary tree has practical uses, the most important being the heap data structure discussed in Section 5.5. Heaps are often used to implement priority queues (Section 5.5) and for external sorting algorithms (Section 8.5.2).

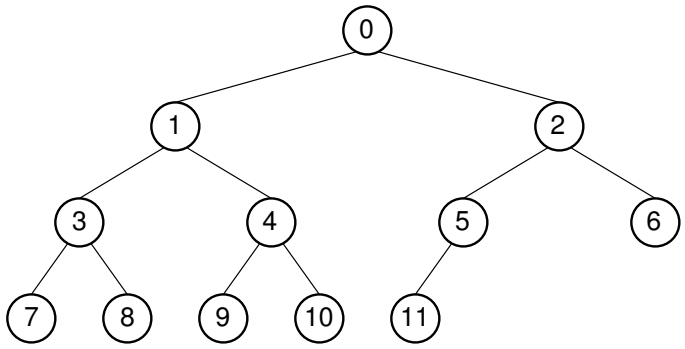
We begin by assigning numbers to the node positions in the complete binary tree, level by level, from left to right as shown in Figure 5.12(a). An array can store the tree's data values efficiently, placing each data value in the array position corresponding to that node's position within the tree. Figure 5.12(b) lists the array indices for the children, parent, and siblings of each node in Figure 5.12(a). From Figure 5.12(b), you should see a pattern regarding the positions of a node's relatives within the array. Simple formulas can be derived for calculating the array index for each relative of a node r from r's index. No explicit pointers are necessary to reach a node's left or right child. This means there is no overhead to the array implementation if the array is selected to be of size n for a tree of n nodes.

The formulae for calculating the array indices of the various relatives of a node are as follows. The total number of nodes in the tree is n. The index of the node in question is r, which must fall in the range 0 to $n - 1$.

- Parent$(r) = \lfloor (r-1)/2 \rfloor$ if $r \neq 0$.
- Left child$(r) = 2r + 1$ if $2r + 1 < n$.
- Right child$(r) = 2r + 2$ if $2r + 2 < n$.
- Left sibling$(r) = r - 1$ if r is even.
- Right sibling$(r) = r + 1$ if r is odd and $r + 1 < n$.

5.4 Binary Search Trees

Section 4.4 presented the dictionary ADT, along with dictionary implementations based on sorted and unsorted lists. When implementing the dictionary with an unsorted list, inserting a new record into the dictionary can be performed quickly by putting it at the end of the list. However, searching an unsorted list for a particular record requires $\Theta(n)$ time in the average case. For a large database, this is probably much too slow. Alternatively, the records can be stored in a sorted list. If the list is implemented using a linked list, then no speedup to the search operation will



(a)

Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	–	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	–	–	–	–	–	–
Right Child	2	4	6	8	10	–	–	–	–	–	–	–
Left Sibling	–	–	1	–	3	–	5	–	7	–	9	–
Right Sibling	–	2	–	4	–	6	–	8	–	10	–	–

(b)

Figure 5.12 A complete binary tree and its array implementation. (a) The complete binary tree with twelve nodes. Each node has been labeled with its position in the tree. (b) The positions for the relatives of each node. A dash indicates that the relative does not exist.

result from storing the records in sorted order. On the other hand, if we use a sorted array-based list to implement the dictionary, then binary search can be used to find a record in only $\Theta(\log n)$ time. However, insertion will now require $\Theta(n)$ time on average because, once the proper location for the new record in the sorted list has been found, many records might be shifted to make room for the new record.

Is there some way to organize a collection of records so that inserting records and searching for records can both be done quickly? This section presents the binary search tree (BST), which allows an improved solution to this problem.

A BST is a binary tree that conforms to the following condition, known as the **Binary Search Tree Property**: All nodes stored in the left subtree of a node whose key value is K have key values less than K. All nodes stored in the right subtree of a node whose key value is K have key values greater than or equal to K. Figure 5.13 shows two BSTs for a collection of values. One consequence of the Binary Search Tree Property is that if the BST nodes are printed using an inorder traversal (see Section 5.2), the resulting enumeration will be in sorted order from lowest to highest.

Figure 5.14 shows a class declaration for the BST that implements the dictionary ADT. The public member functions include those required by the dictionary

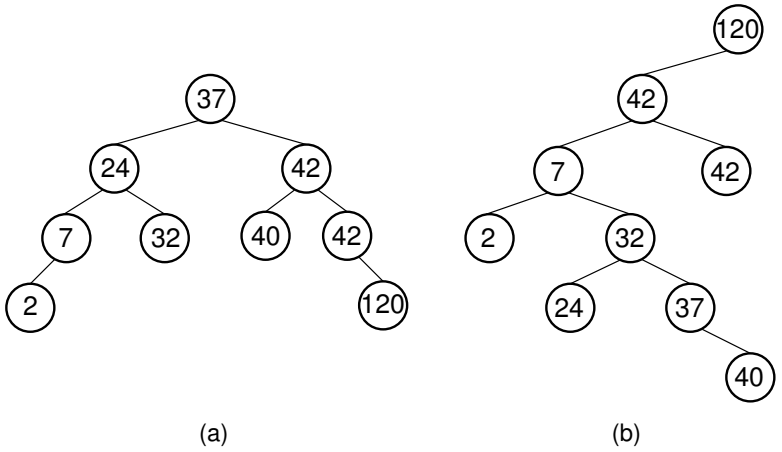


Figure 5.13 Two Binary Search Trees for a collection of values. Tree (a) results if values are inserted in the order 37, 24, 42, 7, 2, 40, 42, 32, 120. Tree (b) results if the same values are inserted in the order 120, 42, 42, 7, 2, 32, 37, 24, 40.

ADT, along with a constructor and destructor. Recall from the discussion in Section 4.4 that there are various ways to deal with keys and comparing records (three approaches being key/value pairs, a special comparison method, and passing in a comparator function). Our BST implementation will handle comparison by explicitly storing a key separate from the data value at each node of the tree.

To find a record with key value K in a BST, begin at the root. If the root stores a record with key value K, then the search is over. If not, then we must search deeper in the tree. What makes the BST efficient during search is that we need search only one of the node's two subtrees. If K is less than the root node's key value, we search only the left subtree. If K is greater than the root node's key value, we search only the right subtree. This process continues until a record with key value K is found, or we reach a leaf node. If we reach a leaf node without encountering K, then no record exists in the BST whose key value is K.

Example 5.5 Consider searching for the node with key value 32 in the tree of Figure 5.13(a). Because 32 is less than the root value of 37, the search proceeds to the left subtree. Because 32 is greater than 24, we search in 24's right subtree. At this point the node containing 32 is found. If the search value were 35, the same path would be followed to the node containing 32. Because this node has no children, we know that 35 is not in the BST.

Notice that in Figure 5.14, public member function **`find`** calls private member function **`findhelp`**. Method **`find`** takes the search key as an explicit parameter and its BST as an implicit parameter, and returns the record that matches the key.

```
// Binary Search Tree implementation for the Dictionary ADT
template <typename Key, typename E>
class BST : public Dictionary<Key,E> {
private:
  BSTNode<Key,E>* root;   // Root of the BST
  int nodecount;          // Number of nodes in the BST

  // Private "helper" functions
  void clearhelp(BSTNode<Key, E>*);
  BSTNode<Key,E>* inserthelp(BSTNode<Key, E>*,
                             const Key&, const E&);
  BSTNode<Key,E>* deletemin(BSTNode<Key, E>*);
  BSTNode<Key,E>* getmin(BSTNode<Key, E>*);
  BSTNode<Key,E>* removehelp(BSTNode<Key, E>*, const Key&);
  E findhelp(BSTNode<Key, E>*, const Key&) const;
  void printhelp(BSTNode<Key, E>*, int) const;

public:
  BST() { root = NULL; nodecount = 0; }  // Constructor
  ~BST() { clearhelp(root); }            // Destructor

  void clear()   // Reinitialize tree
    { clearhelp(root); root = NULL; nodecount = 0; }

  // Insert a record into the tree.
  // k Key value of the record.
  // e The record to insert.
  void insert(const Key& k, const E& e) {
    root = inserthelp(root, k, e);
    nodecount++;
  }

  // Remove a record from the tree.
  // k Key value of record to remove.
  // Return: The record removed, or NULL if there is none.
  E remove(const Key& k) {
    E temp = findhelp(root, k);   // First find it
    if (temp != NULL) {
      root = removehelp(root, k);
      nodecount--;
    }
    return temp;
  }
```

Figure 5.14 The binary search tree implementation.

```
  // Remove and return the root node from the dictionary.
  // Return: The record removed, null if tree is empty.
  E removeAny() {  // Delete min value
    if (root != NULL) {
      E temp = root->element();
      root = removehelp(root, root->key());
      nodecount--;
      return temp;
    }
    else return NULL;
  }

  // Return Record with key value k, NULL if none exist.
  // k: The key value to find. */
  // Return some record matching "k".
  // Return true if such exists, false otherwise. If
  // multiple records match "k", return an arbitrary one.
  E find(const Key& k) const { return findhelp(root, k); }

  // Return the number of records in the dictionary.
  int size() { return nodecount; }

  void print() const { // Print the contents of the BST
    if (root == NULL) cout << "The BST is empty.\n";
    else printhelp(root, 0);
  }
};
```

Figure 5.14 (continued)

However, the find operation is most easily implemented as a recursive function whose parameters are the root of a subtree and the search key. Member **findhelp** has the desired form for this recursive subroutine and is implemented as follows.

```
template <typename Key, typename E>
E BST<Key, E>::findhelp(BSTNode<Key, E>* root,
                        const Key& k) const {
  if (root == NULL) return NULL;          // Empty tree
  if (k < root->key())
    return findhelp(root->left(), k);   // Check left
  else if (k > root->key())
    return findhelp(root->right(), k);  // Check right
  else return root->element();  // Found it
}
```

Once the desired record is found, it is passed through return values up the chain of recursive calls. If a suitable record is not found, **null** is returned.

Inserting a record with key value k requires that we first find where that record would have been if it were in the tree. This takes us to either a leaf node, or to an

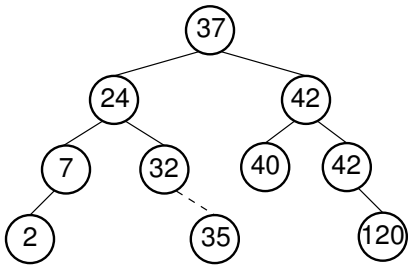


Figure 5.15 An example of BST insertion. A record with value 35 is inserted into the BST of Figure 5.13(a). The node with value 32 becomes the parent of the new node containing 35.

internal node with no child in the appropriate direction.[3] Call this node R'. We then add a new node containing the new record as a child of R'. Figure 5.15 illustrates this operation. The value 35 is added as the right child of the node with value 32. Here is the implementation for **inserthelp**:

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::inserthelp(
    BSTNode<Key, E>* root, const Key& k, const E& it) {
  if (root == NULL)  // Empty tree: create node
    return new BSTNode<Key, E>(k, it, NULL, NULL);
  if (k < root->key())
    root->setLeft(inserthelp(root->left(), k, it));
  else root->setRight(inserthelp(root->right(), k, it));
  return root;       // Return tree with node inserted
}
```

You should pay careful attention to the implementation for **inserthelp**. Note that **inserthelp** returns a pointer to a **BSTNode**. What is being returned is a subtree identical to the old subtree, except that it has been modified to contain the new record being inserted. Each node along a path from the root to the parent of the new node added to the tree will have its appropriate child pointer assigned to it. Except for the last node in the path, none of these nodes will actually change their child's pointer value. In that sense, many of the assignments seem redundant. However, the cost of these additional assignments is worth paying to keep the insertion process simple. The alternative is to check if a given assignment is necessary, which is probably more expensive than the assignment!

The shape of a BST depends on the order in which elements are inserted. A new element is added to the BST as a new leaf node, potentially increasing the depth of the tree. Figure 5.13 illustrates two BSTs for a collection of values. It is possible

[3]This assumes that no node has a key value equal to the one being inserted. If we find a node that duplicates the key value to be inserted, we have two options. If the application does not allow nodes with equal keys, then this insertion should be treated as an error (or ignored). If duplicate keys are allowed, our convention will be to insert the duplicate in the right subtree.

for the BST containing n nodes to be a chain of nodes with height n. This would happen if, for example, all elements were inserted in sorted order. In general, it is preferable for a BST to be as shallow as possible. This keeps the average cost of a BST operation low.

Removing a node from a BST is a bit trickier than inserting a node, but it is not complicated if all of the possible cases are considered individually. Before tackling the general node removal process, let us first discuss how to remove from a given subtree the node with the smallest key value. This routine will be used later by the general node removal function. To remove the node with the minimum key value from a subtree, first find that node by continuously moving down the left link until there is no further left link to follow. Call this node *S*. To remove *S*, simply have the parent of *S* change its pointer to point to the right child of *S*. We know that *S* has no left child (because if *S* did have a left child, *S* would not be the node with minimum key value). Thus, changing the pointer as described will maintain a BST, with *S* removed. The code for this method, named **deletemin**, is as follows:

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
deletemin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL) // Found min
    return rt->right();
  else {                        // Continue left
    rt->setLeft(deletemin(rt->left()));
    return rt;
  }
}
```

Example 5.6 Figure 5.16 illustrates the **deletemin** process. Beginning at the root node with value 10, **deletemin** follows the left link until there is no further left link, in this case reaching the node with value 5. The node with value 10 is changed to point to the right child of the node containing the minimum value. This is indicated in Figure 5.16 by a dashed line.

A pointer to the node containing the minimum-valued element is stored in parameter **S**. The return value of the **deletemin** method is the subtree of the current node with the minimum-valued node in the subtree removed. As with method **inserthelp**, each node on the path back to the root has its left child pointer reassigned to the subtree resulting from its call to the **deletemin** method.

A useful companion method is **getmin** which returns a pointer to the node containing the minimum value in the subtree.

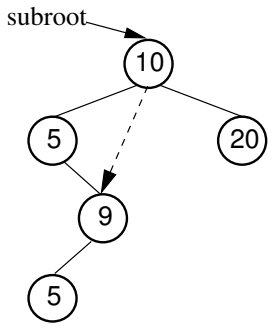


Figure 5.16 An example of deleting the node with minimum value. In this tree, the node with minimum value, 5, is the left child of the root. Thus, the root's **left** pointer is changed to point to 5's right child.

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
getmin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL)
    return rt;
  else return getmin(rt->left());
}
```

Removing a node with given key value R from the BST requires that we first find R and then remove it from the tree. So, the first part of the remove operation is a search to find R. Once R is found, there are several possibilities. If R has no children, then R's parent has its pointer set to **NULL**. If R has one child, then R's parent has its pointer set to R's child (similar to **deletemin**). The problem comes if R has two children. One simple approach, though expensive, is to set R's parent to point to one of R's subtrees, and then reinsert the remaining subtree's nodes one at a time. A better alternative is to find a value in one of the subtrees that can replace the value in R.

Thus, the question becomes: Which value can substitute for the one being removed? It cannot be any arbitrary value, because we must preserve the BST property without making major changes to the structure of the tree. Which value is most like the one being removed? The answer is the least key value greater than (or equal to) the one being removed, or else the greatest key value less than the one being removed. If either of these values replace the one being removed, then the BST property is maintained.

Example 5.7 Assume that we wish to remove the value 37 from the BST of Figure 5.13(a). Instead of removing the root node, we remove the node with the least value in the right subtree (using the **deletemin** operation). This value can then replace the value in the root. In this example we first remove the node with value 40, because it contains the least value in the

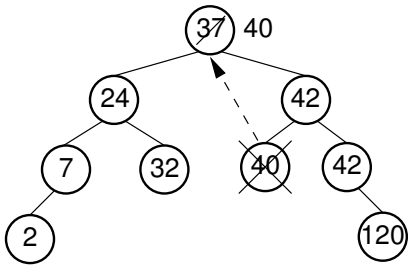


Figure 5.17 An example of removing the value 37 from the BST. The node containing this value has two children. We replace value 37 with the least value from the node's right subtree, in this case 40.

right subtree. We then substitute 40 as the new value for the root node. Figure 5.17 illustrates this process.

When duplicate node values do not appear in the tree, it makes no difference whether the replacement is the greatest value from the left subtree or the least value from the right subtree. If duplicates are stored, then we must select the replacement from the *right* subtree. To see why, call the greatest value in the left subtree *G*. If multiple nodes in the left subtree have value *G*, selecting *G* as the replacement value for the root of the subtree will result in a tree with equal values to the left of the node now containing *G*. Precisely this situation occurs if we replace value 120 with the greatest value in the left subtree of Figure 5.13(b). Selecting the least value from the right subtree does not have a similar problem, because it does not violate the Binary Search Tree Property if equal values appear in the right subtree.

From the above, we see that if we want to remove the record stored in a node with two children, then we simply call **`deletemin`** on the node's right subtree and substitute the record returned for the record being removed. Figure 5.18 shows an implementation for **`removehelp`**.

The cost for **`findhelp`** and **`inserthelp`** is the depth of the node found or inserted. The cost for **`removehelp`** is the depth of the node being removed, or in the case when this node has two children, the depth of the node with smallest value in its right subtree. Thus, in the worst case, the cost for any one of these operations is the depth of the deepest node in the tree. This is why it is desirable to keep BSTs **balanced**, that is, with least possible height. If a binary tree is balanced, then the height for a tree of n nodes is approximately $\log n$. However, if the tree is completely unbalanced, for example in the shape of a linked list, then the height for a tree with n nodes can be as great as n. Thus, a balanced BST will in the average case have operations costing $\Theta(\log n)$, while a badly unbalanced BST can have operations in the worst case costing $\Theta(n)$. Consider the situation where we construct a BST of n nodes by inserting records one at a time. If we are fortunate to have them arrive in an order that results in a balanced tree (a "random" order is

```
// Remove a node with key value k
// Return: The tree with the node removed
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
removehelp(BSTNode<Key, E>* rt, const Key& k) {
  if (rt == NULL) return NULL;    // k is not in tree
  else if (k < rt->key())
    rt->setLeft(removehelp(rt->left(), k));
  else if (k > rt->key())
    rt->setRight(removehelp(rt->right(), k));
  else {                            // Found: remove it
    BSTNode<Key, E>* temp = rt;
    if (rt->left() == NULL) {     // Only a right child
      rt = rt->right();          //  so point to right
      delete temp;
    }
    else if (rt->right() == NULL) { // Only a left child
      rt = rt->left();           //  so point to left
      delete temp;
    }
    else {                      // Both children are non-empty
      BSTNode<Key, E>* temp = getmin(rt->right());
      rt->setElement(temp->element());
      rt->setKey(temp->key());
      rt->setRight(deletemin(rt->right()));
      delete temp;
    }
  }
  return rt;
}
```

Figure 5.18 Implementation for the BST **removehelp** method.

likely to be good enough for this purpose), then each insertion will cost on average $\Theta(\log n)$, for a total cost of $\Theta(n \log n)$. However, if the records are inserted in order of increasing value, then the resulting tree will be a chain of height n. The cost of insertion in this case will be $\sum_{i=1}^{n} i = \Theta(n^2)$.

Traversing a BST costs $\Theta(n)$ regardless of the shape of the tree. Each node is visited exactly once, and each child pointer is followed exactly once.

Below are two example traversals. The first is member **clearhelp**, which returns the nodes of the BST to the freelist. Because the children of a node must be freed before the node itself, this is a postorder traversal.

```
template <typename Key, typename E>
void BST<Key, E>::
clearhelp(BSTNode<Key, E>* root) {
  if (root == NULL) return;
  clearhelp(root->left());
  clearhelp(root->right());
  delete root;
}
```

The next example is **printhelp**, which performs an inorder traversal on the BST to print the node values in ascending order. Note that **printhelp** indents each line to indicate the depth of the corresponding node in the tree. Thus we pass in the current level of the tree in **level**, and increment this value each time that we make a recursive call.

```
template <typename Key, typename E>
void BST<Key, E>::
printhelp(BSTNode<Key, E>* root, int level) const {
  if (root == NULL) return;           // Empty tree
  printhelp(root->left(), level+1);   // Do left subtree
  for (int i=0; i<level; i++)         // Indent to level
    cout << "  ";
  cout << root->key() << "\n";        // Print node value
  printhelp(root->right(), level+1);  // Do right subtree
}
```

While the BST is simple to implement and efficient when the tree is balanced, the possibility of its being unbalanced is a serious liability. There are techniques for organizing a BST to guarantee good performance. Two examples are the AVL tree and the splay tree of Section 13.2. Other search trees are guaranteed to remain balanced, such as the 2-3 tree of Section 10.4.

5.5 Heaps and Priority Queues

There are many situations, both in real life and in computing applications, where we wish to choose the next "most important" from a collection of people, tasks, or objects. For example, doctors in a hospital emergency room often choose to see next the "most critical" patient rather than the one who arrived first. When scheduling programs for execution in a multitasking operating system, at any given moment there might be several programs (usually called **jobs**) ready to run. The next job selected is the one with the highest **priority**. Priority is indicated by a particular value associated with the job (and might change while the job remains in the wait list).

When a collection of objects is organized by importance or priority, we call this a **priority queue**. A normal queue data structure will not implement a priority queue efficiently because search for the element with highest priority will take $\Theta(n)$ time. A list, whether sorted or not, will also require $\Theta(n)$ time for either insertion or removal. A BST that organizes records by priority could be used, with the total of n inserts and n remove operations requiring $\Theta(n \log n)$ time in the average case. However, there is always the possibility that the BST will become unbalanced, leading to bad performance. Instead, we would like to find a data structure that is guaranteed to have good performance for this special application.

This section presents the **heap**[4] data structure. A heap is defined by two properties. First, it is a complete binary tree, so heaps are nearly always implemented using the array representation for complete binary trees presented in Section 5.3.3. Second, the values stored in a heap are **partially ordered**. This means that there is a relationship between the value stored at any node and the values of its children. There are two variants of the heap, depending on the definition of this relationship.

A **max-heap** has the property that every node stores a value that is *greater* than or equal to the value of either of its children. Because the root has a value greater than or equal to its children, which in turn have values greater than or equal to their children, the root stores the maximum of all values in the tree.

A **min-heap** has the property that every node stores a value that is *less* than or equal to that of its children. Because the root has a value less than or equal to its children, which in turn have values less than or equal to their children, the root stores the minimum of all values in the tree.

Note that there is no necessary relationship between the value of a node and that of its sibling in either the min-heap or the max-heap. For example, it is possible that the values for all nodes in the left subtree of the root are greater than the values for every node of the right subtree. We can contrast BSTs and heaps by the strength of their ordering relationships. A BST defines a total order on its nodes in that, given the positions for any two nodes in the tree, the one to the "left" (equivalently, the one appearing earlier in an inorder traversal) has a smaller key value than the one to the "right." In contrast, a heap implements a partial order. Given their positions, we can determine the relative order for the key values of two nodes in the heap *only* if one is a descendant of the other.

Min-heaps and max-heaps both have their uses. For example, the Heapsort of Section 7.6 uses the max-heap, while the Replacement Selection algorithm of Section 8.5.2 uses a min-heap. The examples in the rest of this section will use a max-heap.

Be careful not to confuse the logical representation of a heap with its physical implementation by means of the array-based complete binary tree. The two are not synonymous because the logical view of the heap is actually a tree structure, while the typical physical implementation uses an array.

Figure 5.19 shows an implementation for heaps.The class is a template with two parameters. `E` defines the type for the data elements stored in the heap, while `Comp` is the comparison class for comparing two elements. This class can implement either a min-heap or a max-heap by changing the definition for `Comp`. `Comp` defines method `prior`, a binary Boolean function that returns true if the first parameter should come before the second in the heap.

This class definition makes two concessions to the fact that an array-based implementation is used. First, heap nodes are indicated by their logical position within

[4]The term "heap" is also sometimes used to refer to a memory pool. See Section 12.3.

```
// Heap class
template <typename E, typename Comp> class heap {
private:
  E* Heap;          // Pointer to the heap array
  int maxsize;         // Maximum size of the heap
  int n;               // Number of elements now in the heap

  // Helper function to put element in its correct place
  void siftdown(int pos) {
    while (!isLeaf(pos)) { // Stop if pos is a leaf
      int j = leftchild(pos);  int rc = rightchild(pos);
      if ((rc < n) && Comp::prior(Heap[rc], Heap[j]))
        j = rc;            // Set j to greater child's value
      if (Comp::prior(Heap[pos], Heap[j])) return; // Done
      swap(Heap, pos, j);
      pos = j;             // Move down
    }
  }

public:
  heap(E* h, int num, int max)     // Constructor
    { Heap = h;  n = num;  maxsize = max;  buildHeap(); }
  int size() const       // Return current heap size
    { return n; }
  bool isLeaf(int pos) const // True if pos is a leaf
    { return (pos >= n/2) && (pos < n); }
  int leftchild(int pos) const
    { return 2*pos + 1; }    // Return leftchild position
  int rightchild(int pos) const
    { return 2*pos + 2; }    // Return rightchild position
  int parent(int pos) const  // Return parent position
    { return (pos-1)/2; }
  void buildHeap()           // Heapify contents of Heap
    { for (int i=n/2-1; i>=0; i--) siftdown(i); }

  // Insert "it" into the heap
  void insert(const E& it) {
    Assert(n < maxsize, "Heap is full");
    int curr = n++;
    Heap[curr] = it;            // Start at end of heap
    // Now sift up until curr's parent > curr
    while ((curr!=0) &&
           (Comp::prior(Heap[curr], Heap[parent(curr)]))) {
      swap(Heap, curr, parent(curr));
      curr = parent(curr);
    }
  }
```

Figure 5.19 An implementation for the heap.

```
  // Remove first value
  E removefirst() {
    Assert (n > 0, "Heap is empty");
    swap(Heap, 0, --n);       // Swap first with last value
    if (n != 0) siftdown(0);  // Siftdown new root val
    return Heap[n];             // Return deleted value
  }

  // Remove and return element at specified position
  E remove(int pos) {
    Assert((pos >= 0) && (pos < n), "Bad position");
    if (pos == (n-1)) n--; // Last element, no work to do
    else
    {
      swap(Heap, pos, --n);          // Swap with last value
      while ((pos != 0) &&
             (Comp::prior(Heap[pos], Heap[parent(pos)]))) {
        swap(Heap, pos, parent(pos)); // Push up large key
        pos = parent(pos);
      }
      if (n != 0) siftdown(pos);     // Push down small key
    }
    return Heap[n];
  }
};
```

Figure 5.19 (continued)

the heap rather than by a pointer to the node. In practice, the logical heap position corresponds to the identically numbered physical position in the array. Second, the constructor takes as input a pointer to the array to be used. This approach provides the greatest flexibility for using the heap because all data values can be loaded into the array directly by the client. The advantage of this comes during the heap construction phase, as explained below. The constructor also takes an integer parameter indicating the initial size of the heap (based on the number of elements initially loaded into the array) and a second integer parameter indicating the maximum size allowed for the heap (the size of the array).

Method **heapsize** returns the current size of the heap. **H.isLeaf(pos)** returns **true** if position **pos** is a leaf in heap **H**, and **false** otherwise. Members **leftchild**, **rightchild**, and **parent** return the position (actually, the array index) for the left child, right child, and parent of the position passed, respectively.

One way to build a heap is to insert the elements one at a time. Method **insert** will insert a new element V into the heap. You might expect the heap insertion process to be similar to the insert function for a BST, starting at the root and working down through the heap. However, this approach is not likely to work because the heap must maintain the shape of a complete binary tree. Equivalently, if the heap takes up the first n positions of its array prior to the call to **insert**, it must take

up the first $n + 1$ positions after. To accomplish this, **`insert`** first places *V* at position n of the array. Of course, *V* is unlikely to be in the correct position. To move *V* to the right place, it is compared to its parent's value. If the value of *V* is less than or equal to the value of its parent, then it is in the correct place and the insert routine is finished. If the value of *V* is greater than that of its parent, then the two elements swap positions. From here, the process of comparing *V* to its (current) parent continues until *V* reaches its correct position.

Since the heap is a complete binary tree, its height is guaranteed to be the minimum possible. In particular, a heap containing n nodes will have a height of $\Theta(\log n)$. Intuitively, we can see that this must be true because each level that we add will slightly more than double the number of nodes in the tree (the ith level has 2^i nodes, and the sum of the first i levels is $2^{i+1} - 1$). Starting at 1, we can double only $\log n$ times to reach a value of n. To be precise, the height of a heap with n nodes is $\lceil \log(n + 1) \rceil$.

Each call to **`insert`** takes $\Theta(\log n)$ time in the worst case, because the value being inserted can move at most the distance from the bottom of the tree to the top of the tree. Thus, to insert n values into the heap, if we insert them one at a time, will take $\Theta(n \log n)$ time in the worst case.

If all n values are available at the beginning of the building process, we can build the heap faster than just inserting the values into the heap one by one. Consider Figure 5.20(a), which shows one series of exchanges that could be used to build the heap. All exchanges are between a node and one of its children. The heap is formed as a result of this exchange process. The array for the right-hand tree of Figure 5.20(a) would appear as follows:

7	4	6	1	2	3	5

Figure 5.20(b) shows an alternate series of exchanges that also forms a heap, but much more efficiently. The equivalent array representation would be

7	5	6	4	2	1	3

From this example, it is clear that the heap for any given set of numbers is not unique, and we see that some rearrangements of the input values require fewer exchanges than others to build the heap. So, how do we pick the best rearrangement?

One good algorithm stems from induction. Suppose that the left and right subtrees of the root are already heaps, and *R* is the name of the element at the root. This situation is illustrated by Figure 5.21. In this case there are two possibilities. (1) *R* has a value greater than or equal to its two children. In this case, construction is complete. (2) *R* has a value less than one or both of its children. In this case, *R* should be exchanged with the child that has greater value. The result will be a heap, except that *R* might still be less than one or both of its (new) children. In this case, we simply continue the process of "pushing down" *R* until it reaches a

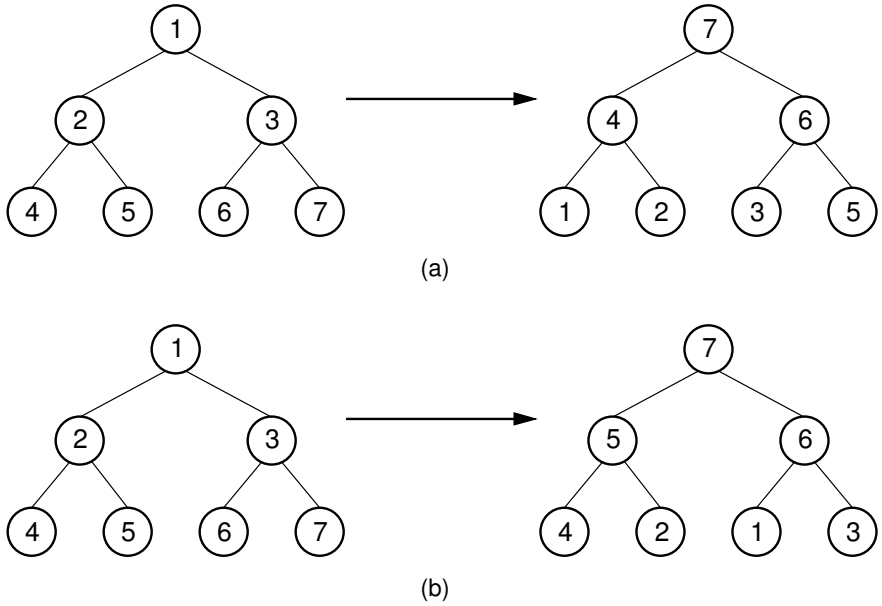


Figure 5.20 Two series of exchanges to build a max-heap. (a) This heap is built by a series of nine exchanges in the order (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6). (b) This heap is built by a series of four exchanges in the order (5-2), (7-3), (7-1), (6-1).

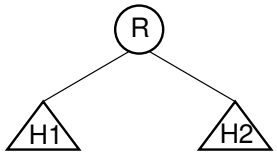


Figure 5.21 Final stage in the heap-building algorithm. Both subtrees of node *R* are heaps. All that remains is to push *R* down to its proper level in the heap.

level where it is greater than its children, or is a leaf node. This process is implemented by the private method **`siftdown`**. The siftdown operation is illustrated by Figure 5.22.

This approach assumes that the subtrees are already heaps, suggesting that a complete algorithm can be obtained by visiting the nodes in some order such that the children of a node are visited *before* the node itself. One simple way to do this is simply to work from the high index of the array to the low index. Actually, the build process need not visit the leaf nodes (they can never move down because they are already at the bottom), so the building algorithm can start in the middle of the array, with the first internal node. The exchanges shown in Figure 5.20(b) result from this process. Method **`buildHeap`** implements the building algorithm.

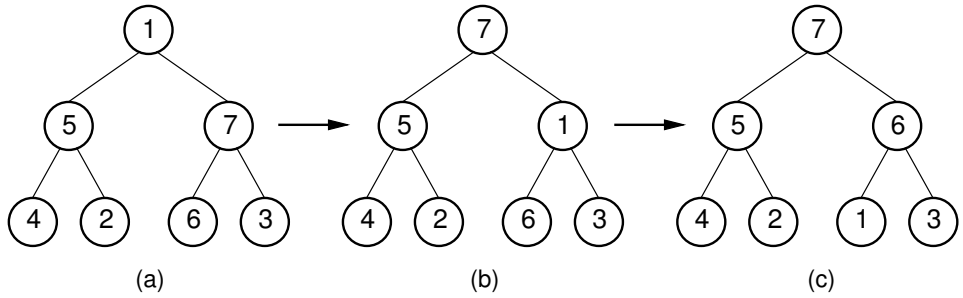


Figure 5.22 The siftdown operation. The subtrees of the root are assumed to be heaps. (a) The partially completed heap. (b) Values 1 and 7 are swapped. (c) Values 1 and 6 are swapped to form the final heap.

What is the cost of **`buildHeap`**? Clearly it is the sum of the costs for the calls to **`siftdown`**. Each **`siftdown`** operation can cost at most the number of levels it takes for the node being sifted to reach the bottom of the tree. In any complete tree, approximately half of the nodes are leaves and so cannot be moved downward at all. One quarter of the nodes are one level above the leaves, and so their elements can move down at most one level. At each step up the tree we get half the number of nodes as were at the previous level, and an additional height of one. The maximum sum of total distances that elements can go is therefore

$$\sum_{i=1}^{\log n}(i-1)\frac{n}{2^i} = \frac{n}{2}\sum_{i=1}^{\log n}\frac{i-1}{2^{i-1}}.$$

From Equation 2.9 we know that this summation has a closed-form solution of approximately 2, so this algorithm takes $\Theta(n)$ time in the worst case. This is far better than building the heap one element at a time, which would cost $\Theta(n \log n)$ in the worst case. It is also faster than the $\Theta(n \log n)$ average-case time and $\Theta(n^2)$ worst-case time required to build the BST.

Removing the maximum (root) value from a heap containing n elements requires that we maintain the complete binary tree shape, and that the remaining $n-1$ node values conform to the heap property. We can maintain the proper shape by moving the element in the last position in the heap (the current last element in the array) to the root position. We now consider the heap to be one element smaller. Unfortunately, the new root value is probably *not* the maximum value in the new heap. This problem is easily solved by using **`siftdown`** to reorder the heap. Because the heap is $\log n$ levels deep, the cost of deleting the maximum element is $\Theta(\log n)$ in the average and worst cases.

The heap is a natural implementation for the priority queue discussed at the beginning of this section. Jobs can be added to the heap (using their priority value as the ordering key) when needed. Method **`removemax`** can be called whenever a new job is to be executed.

Some applications of priority queues require the ability to change the priority of an object already stored in the queue. This might require that the object's position in the heap representation be updated. Unfortunately, a max-heap is not efficient when searching for an arbitrary value; it is only good for finding the maximum value. However, if we already know the index for an object within the heap, it is a simple matter to update its priority (including changing its position to maintain the heap property) or remove it. The **`remove`** method takes as input the position of the node to be removed from the heap. A typical implementation for priority queues requiring updating of priorities will need to use an auxiliary data structure that supports efficient search for objects (such as a BST). Records in the auxiliary data structure will store the object's heap index, so that the object can be deleted from the heap and reinserted with its new priority (see Project 5.5). Sections 11.4.1 and 11.5.1 present applications for a priority queue with priority updating.

5.6 Huffman Coding Trees

The space/time tradeoff principle from Section 3.9 states that one can often gain an improvement in space requirements in exchange for a penalty in running time. There are many situations where this is a desirable tradeoff. A typical example is storing files on disk. If the files are not actively used, the owner might wish to compress them to save space. Later, they can be uncompressed for use, which costs some time, but only once.

We often represent a set of items in a computer program by assigning a unique code to each item. For example, the standard ASCII coding scheme assigns a unique eight-bit value to each character. It takes a certain minimum number of bits to provide unique codes for each character. For example, it takes $\lceil \log 128 \rceil$ or seven bits to provide the 128 unique codes needed to represent the 128 symbols of the ASCII character set.[5]

The requirement for $\lceil \log n \rceil$ bits to represent n unique code values assumes that all codes will be the same length, as are ASCII codes. This is called a **fixed-length** coding scheme. If all characters were used equally often, then a fixed-length coding scheme is the most space efficient method. However, you are probably aware that not all characters are used equally often in many applications. For example, the various letters in an English language document have greatly different frequencies of use.

Figure 5.23 shows the relative frequencies of the letters of the alphabet. From this table we can see that the letter 'E' appears about 60 times more often than the letter 'Z.' In normal ASCII, the words "DEED" and "MUCK" require the same

[5]The ASCII standard is eight bits, not seven, even though there are only 128 characters represented. The eighth bit is used either to check for transmission errors, or to support "extended" ASCII codes with an additional 128 characters.

Letter	Frequency	Letter	Frequency
A	77	N	67
B	17	O	67
C	32	P	20
D	42	Q	5
E	120	R	59
F	24	S	67
G	17	T	85
H	50	U	37
I	76	V	12
J	4	W	22
K	7	X	4
L	42	Y	22
M	24	Z	2

Figure 5.23 Relative frequencies for the 26 letters of the alphabet as they appear in a selected set of English documents. "Frequency" represents the expected frequency of occurrence per 1000 letters, ignoring case.

amount of space (four bytes). It would seem that words such as "DEED," which are composed of relatively common letters, should be storable in less space than words such as "MUCK," which are composed of relatively uncommon letters.

If some characters are used more frequently than others, is it possible to take advantage of this fact and somehow assign them shorter codes? The price could be that other characters require longer codes, but this might be worthwhile if such characters appear rarely enough. This concept is at the heart of file compression techniques in common use today. The next section presents one such approach to assigning **variable-length** codes, called Huffman coding. While it is not commonly used in its simplest form for file compression (there are better methods), Huffman coding gives the flavor of such coding schemes. One motivation for studying Huffman coding is because it provides our first opportunity to see a type of tree structure referred to as a **search trie**.

5.6.1 Building Huffman Coding Trees

Huffman coding assigns codes to characters such that the length of the code depends on the relative frequency or **weight** of the corresponding character. Thus, it is a variable-length code. If the estimated frequencies for letters match the actual frequency found in an encoded message, then the length of that message will typically be less than if a fixed-length code had been used. The Huffman code for each letter is derived from a full binary tree called the **Huffman coding tree**, or simply the **Huffman tree**. Each leaf of the Huffman tree corresponds to a letter, and we define the weight of the leaf node to be the weight (frequency) of its associated

Letter	C	D	E	K	L	M	U	Z
Frequency	32	42	120	7	42	24	37	2

Figure 5.24 The relative frequencies for eight selected letters.

letter. The goal is to build a tree with the **minimum external path weight**. Define the **weighted path length** of a leaf to be its weight times its depth. The binary tree with minimum external path weight is the one with the minimum sum of weighted path lengths for the given set of leaves. A letter with high weight should have low depth, so that it will count the least against the total path length. As a result, another letter might be pushed deeper in the tree if it has less weight.

The process of building the Huffman tree for n letters is quite simple. First, create a collection of n initial Huffman trees, each of which is a single leaf node containing one of the letters. Put the n partial trees onto a priority queue organized by weight (frequency). Next, remove the first two trees (the ones with lowest weight) from the priority queue. Join these two trees together to create a new tree whose root has the two trees as children, and whose weight is the sum of the weights of the two trees. Put this new tree back into the priority queue. This process is repeated until all of the partial Huffman trees have been combined into one.

Example 5.8 Figure 5.25 illustrates part of the Huffman tree construction process for the eight letters of Figure 5.24. Ranking D and L arbitrarily by alphabetical order, the letters are ordered by frequency as

Letter	Z	K	M	C	U	D	L	E
Frequency	2	7	24	32	37	42	42	120

Because the first two letters on the list are Z and K, they are selected to be the first trees joined together.[6] They become the children of a root node with weight 9. Thus, a tree whose root has weight 9 is placed back on the list, where it takes up the first position. The next step is to take values 9 and 24 off the list (corresponding to the partial tree with two leaf nodes built in the last step, and the partial tree storing the letter M, respectively) and join them together. The resulting root node has weight 33, and so this tree is placed back into the list. Its priority will be between the trees with values 32 (for letter C) and 37 (for letter U). This process continues until a tree whose root has weight 306 is built. This tree is shown in Figure 5.26.

---

[6]For clarity, the examples for building Huffman trees show a sorted list to keep the letters ordered by frequency. But a real implementation would use a heap to implement the priority queue for efficiency.

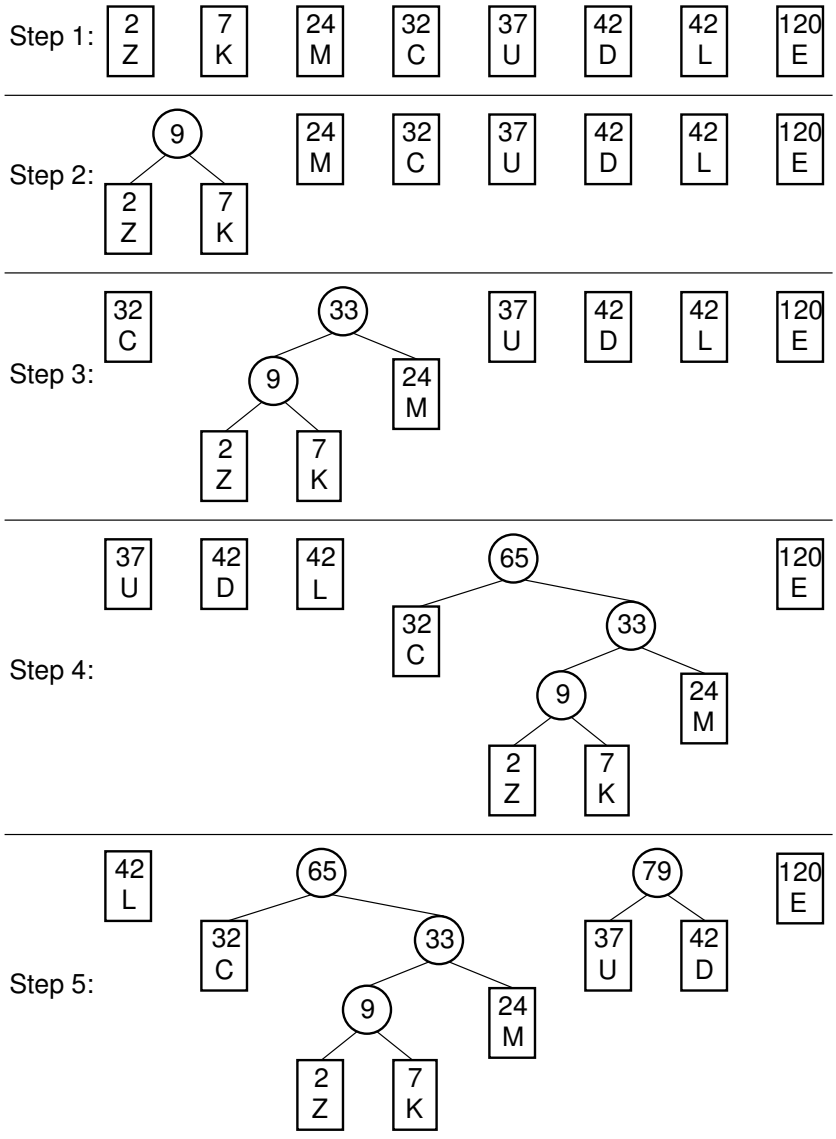


Figure 5.25 The first five steps of the building process for a sample Huffman tree.

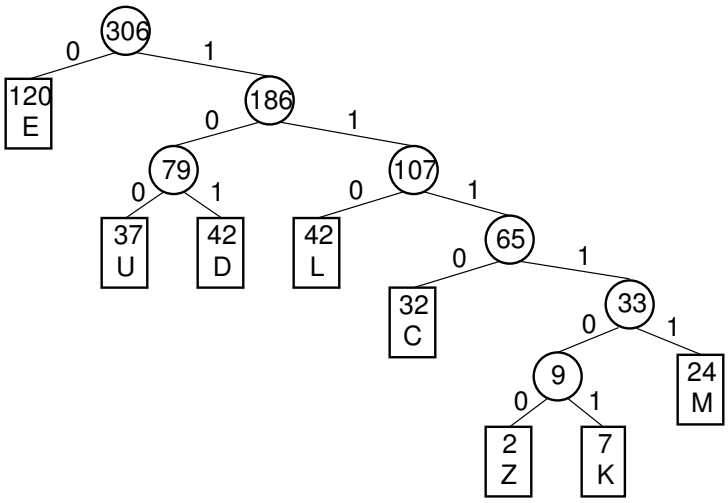


Figure 5.26 A Huffman tree for the letters of Figure 5.24.

Figure 5.27 shows an implementation for Huffman tree nodes. This implementation is similar to the **`VarBinNode`** implementation of Figure 5.10. There is an abstract base class, named **`HuffNode`**, and two subclasses, named **`LeafNode`** and **`IntlNode`**. This implementation reflects the fact that leaf and internal nodes contain distinctly different information.

Figure 5.28 shows the implementation for the Huffman tree. Figure 5.29 shows the **C**++ code for the tree-building process.

Huffman tree building is an example of a **greedy algorithm**. At each step, the algorithm makes a "greedy" decision to merge the two subtrees with least weight. This makes the algorithm simple, but does it give the desired result? This section concludes with a proof that the Huffman tree indeed gives the most efficient arrangement for the set of letters. The proof requires the following lemma.

Lemma 5.1 *For any Huffman tree built by function* **`buildHuff`** *containing at least two letters, the two letters with least frequency are stored in siblings nodes whose depth is at least as deep as any other leaf nodes in the tree.*

Proof: Call the two letters with least frequency l_1 and l_2. They must be siblings because **`buildHuff`** selects them in the first step of the construction process. Assume that l_1 and l_2 are not the deepest nodes in the tree. In this case, the Huffman tree must either look as shown in Figure 5.30, or in some sense be symmetrical to this. For this situation to occur, the parent of l_1 and l_2, labeled V, must have greater weight than the node labeled X. Otherwise, function **`buildHuff`** would have selected node V in place of node X as the child of node U. However, this is impossible because l_1 and l_2 are the letters with least frequency. □

```
// Huffman tree node abstract base class
template <typename E> class HuffNode {
public:
  virtual ~HuffNode() {}                // Base destructor
  virtual int weight() = 0;             // Return frequency
  virtual bool isLeaf() = 0;            // Determine type
};

template <typename E>   // Leaf node subclass
class LeafNode : public HuffNode<E> {
private:
  E it;                 // Value
  int wgt;              // Weight
public:
  LeafNode(const E& val, int freq)   // Constructor
    { it = val; wgt = freq; }
  int weight() { return wgt; }
  E val() { return it; }
  bool isLeaf() { return true; }
};

template <typename E>   // Internal node subclass
class IntlNode : public HuffNode<E> {
private:
  HuffNode<E>* lc;   // Left child
  HuffNode<E>* rc;   // Right child
  int wgt;              // Subtree weight
public:
  IntlNode(HuffNode<E>* l, HuffNode<E>* r)
    { wgt = l->weight() + r->weight(); lc = l; rc = r; }
  int weight() { return wgt; }
  bool isLeaf() { return false; }
  HuffNode<E>* left() const { return lc; }
  void setLeft(HuffNode<E>* b)
    { lc = (HuffNode<E>*)b; }
  HuffNode<E>* right() const { return rc; }
  void setRight(HuffNode<E>* b)
    { rc = (HuffNode<E>*)b; }
};
```

Figure 5.27 Implementation for Huffman tree nodes. Internal nodes and leaf nodes are represented by separate classes, each derived from an abstract base class.

```
// HuffTree is a template of two parameters: the element
//  type being coded and a comparator for two such elements.
template <typename E>
class HuffTree {
private:
  HuffNode<E>* Root;         // Tree root
public:
  HuffTree(E& val, int freq) // Leaf constructor
    { Root = new LeafNode<E>(val, freq); }
  // Internal node constructor
  HuffTree(HuffTree<E>* l, HuffTree<E>* r)
    { Root = new IntlNode<E>(l->root(), r->root()); }
  ~HuffTree() {}                          // Destructor
  HuffNode<E>* root() { return Root; }    // Get root
  int weight() { return Root->weight(); } // Root weight
};
```

Figure 5.28 Class declarations for the Huffman tree.

```
// Build a Huffman tree from a collection of frequencies
template <typename E> HuffTree<E>*
buildHuff(HuffTree<E>** TreeArray, int count) {
  heap<HuffTree<E>*,minTreeComp>* forest =
    new heap<HuffTree<E>*, minTreeComp>(TreeArray,
                                        count, count);
  HuffTree<char> *temp1, *temp2, *temp3 = NULL;
  while (forest->size() > 1) {
    temp1 = forest->removefirst();   // Pull first two trees
    temp2 = forest->removefirst();   //   off the list
    temp3 = new HuffTree<E>(temp1, temp2);
    forest->insert(temp3);  // Put the new tree back on list
    delete temp1;          // Must delete the remnants
    delete temp2;          //   of the trees we created
  }
  return temp3;
}
```

Figure 5.29 Implementation for the Huffman tree construction function. **buildHuff** takes as input **fl**, the min-heap of partial Huffman trees, which initially are single leaf nodes as shown in Step 1 of Figure 5.25. The body of function **buildTree** consists mainly of a **for** loop. On each iteration of the **for** loop, the first two partial trees are taken off the heap and placed in variables **temp1** and **temp2**. A tree is created (**temp3**) such that the left and right subtrees are **temp1** and **temp2**, respectively. Finally, **temp3** is returned to **fl**.

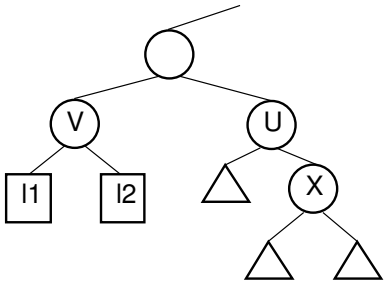


Figure 5.30 An impossible Huffman tree, showing the situation where the two nodes with least weight, l_1 and l_2, are not the deepest nodes in the tree. Triangles represent subtrees.

Theorem 5.3 *Function* **buildHuff** *builds the Huffman tree with the minimum external path weight for the given set of letters.*

Proof: The proof is by induction on n, the number of letters.

- **Base Case**: For $n = 2$, the Huffman tree must have the minimum external path weight because there are only two possible trees, each with identical weighted path lengths for the two leaves.
- **Induction Hypothesis**: Assume that any tree created by **buildHuff** that contains $n - 1$ leaves has minimum external path length.
- **Induction Step**: Given a Huffman tree **T** built by **buildHuff** with n leaves, $n \geq 2$, suppose that $w_1 \leq w_2 \leq \cdots \leq w_n$ where w_1 to w_n are the weights of the letters. Call V the parent of the letters with frequencies w_1 and w_2. From the lemma, we know that the leaf nodes containing the letters with frequencies w_1 and w_2 are as deep as any nodes in **T**. If any other leaf nodes in the tree were deeper, we could reduce their weighted path length by swapping them with w_1 or w_2. But the lemma tells us that no such deeper nodes exist. Call $\mathbf{T}'$ the Huffman tree that is identical to **T** except that node V is replaced with a leaf node V' whose weight is $w_1 + w_2$. By the induction hypothesis, $\mathbf{T}'$ has minimum external path length. Returning the children to V' restores tree **T**, which must also have minimum external path length.

Thus by mathematical induction, function **buildHuff** creates the Huffman tree with minimum external path length. □

5.6.2 Assigning and Using Huffman Codes

Once the Huffman tree has been constructed, it is an easy matter to assign codes to individual letters. Beginning at the root, we assign either a '0' or a '1' to each edge in the tree. '0' is assigned to edges connecting a node with its left child, and '1' to edges connecting a node with its right child. This process is illustrated by

Letter	Freq	Code	Bits
C	32	1110	4
D	42	101	3
E	120	0	1
K	7	111101	6
L	42	110	3
M	24	11111	5
U	37	100	3
Z	2	111100	6

Figure 5.31 The Huffman codes for the letters of Figure 5.24.

Figure 5.26. The Huffman code for a letter is simply a binary number determined by the path from the root to the leaf corresponding to that letter. Thus, the code for E is '0' because the path from the root to the leaf node for E takes a single left branch. The code for K is '111101' because the path to the node for K takes four right branches, then a left, and finally one last right. Figure 5.31 lists the codes for all eight letters.

Given codes for the letters, it is a simple matter to use these codes to encode a text message. We simply replace each letter in the string with its binary code. A lookup table can be used for this purpose.

Example 5.9 Using the code generated by our example Huffman tree, the word "DEED" is represented by the bit string "10100101" and the word "MUCK" is represented by the bit string "111111001110111101."

Decoding the message is done by looking at the bits in the coded string from left to right until a letter is decoded. This can be done by using the Huffman tree in a reverse process from that used to generate the codes. Decoding a bit string begins at the root of the tree. We take branches depending on the bit value — left for '0' and right for '1' — until reaching a leaf node. This leaf contains the first character in the message. We then process the next bit in the code restarting at the root to begin the next character.

Example 5.10 To decode the bit string "1011001110111101" we begin at the root of the tree and take a right branch for the first bit which is '1.' Because the next bit is a '0' we take a left branch. We then take another right branch (for the third bit '1'), arriving at the leaf node corresponding to the letter D. Thus, the first letter of the coded word is D. We then begin again at the root of the tree to process the fourth bit, which is a '1.' Taking a right branch, then two left branches (for the next two bits which are '0'), we reach the leaf node corresponding to the letter U. Thus, the second letter

is U. In similar manner we complete the decoding process to find that the last two letters are C and K, spelling the word "DUCK."

A set of codes is said to meet the **prefix property** if no code in the set is the prefix of another. The prefix property guarantees that there will be no ambiguity in how a bit string is decoded. In other words, once we reach the last bit of a code during the decoding process, we know which letter it is the code for. Huffman codes certainly have the prefix property because any prefix for a code would correspond to an internal node, while all codes correspond to leaf nodes. For example, the code for M is '11111.' Taking five right branches in the Huffman tree of Figure 5.26 brings us to the leaf node containing M. We can be sure that no letter can have code '111' because this corresponds to an internal node of the tree, and the tree-building process places letters only at the leaf nodes.

How efficient is Huffman coding? In theory, it is an optimal coding method whenever the true frequencies are known, and the frequency of a letter is independent of the context of that letter in the message. In practice, the frequencies of letters in an English text document do change depending on context. For example, while E is the most commonly used letter of the alphabet in English documents, T is more common as the first letter of a word. This is why most commercial compression utilities do not use Huffman coding as their primary coding method, but instead use techniques that take advantage of the context for the letters.

Another factor that affects the compression efficiency of Huffman coding is the relative frequencies of the letters. Some frequency patterns will save no space as compared to fixed-length codes; others can result in great compression. In general, Huffman coding does better when there is large variation in the frequencies of letters. In the particular case of the frequencies shown in Figure 5.31, we can determine the expected savings from Huffman coding if the actual frequencies of a coded message match the expected frequencies.

Example 5.11 Because the sum of the frequencies in Figure 5.31 is 306 and E has frequency 120, we expect it to appear 120 times in a message containing 306 letters. An actual message might or might not meet this expectation. Letters D, L, and U have code lengths of three, and together are expected to appear 121 times in 306 letters. Letter C has a code length of four, and is expected to appear 32 times in 306 letters. Letter M has a code length of five, and is expected to appear 24 times in 306 letters. Finally, letters K and Z have code lengths of six, and together are expected to appear only 9 times in 306 letters. The average expected cost per character is simply the sum of the cost for each character (c_i) times the probability of its occurring (p_i), or

$$c_1p_1 + c_2p_2 + \cdots + c_np_n.$$

This can be reorganized as

$$\frac{c_1 f_1 + c_2 f_2 + \cdots + c_n f_n}{f_T}$$

where f_i is the (relative) frequency of letter i and f_T is the total for all letter frequencies. For this set of frequencies, the expected cost per letter is

$$[(1\times 120)+(3\times 121)+(4\times 32)+(5\times 24)+(6\times 9)]/306 = 785/306 \approx 2.57$$

A fixed-length code for these eight characters would require $\log 8 = 3$ bits per letter as opposed to about 2.57 bits per letter for Huffman coding. Thus, Huffman coding is expected to save about 14% for this set of letters.

Huffman coding for all ASCII symbols should do better than this. The letters of Figure 5.31 are atypical in that there are too many common letters compared to the number of rare letters. Huffman coding for all 26 letters would yield an expected cost of 4.29 bits per letter. The equivalent fixed-length code would require about five bits. This is somewhat unfair to fixed-length coding because there is actually room for 32 codes in five bits, but only 26 letters. More generally, Huffman coding of a typical text file will save around 40% over ASCII coding if we charge ASCII coding at eight bits per character. Huffman coding for a binary file (such as a compiled executable) would have a very different set of distribution frequencies and so would have a different space savings. Most commercial compression programs use two or three coding schemes to adjust to different types of files.

In the preceding example, "DEED" was coded in 8 bits, a saving of 33% over the twelve bits required from a fixed-length coding. However, "MUCK" requires 18 bits, more space than required by the corresponding fixed-length coding. The problem is that "MUCK" is composed of letters that are not expected to occur often. If the message does not match the expected frequencies of the letters, than the length of the encoding will not be as expected either.

5.6.3 Search in Huffman Trees

When we decode a character using the Huffman coding tree, we follow a path through the tree dictated by the bits in the code string. Each '0' bit indicates a left branch while each '1' bit indicates a right branch. Now look at Figure 5.26 and consider this structure in terms of searching for a given letter (whose key value is its Huffman code). We see that all letters with codes beginning with '0' are stored in the left branch, while all letters with codes beginning with '1' are stored in the right branch. Contrast this with storing records in a BST. There, all records with key value less than the root value are stored in the left branch, while all records with key values greater than the root are stored in the right branch.

If we view all records stored in either of these structures as appearing at some point on a number line representing the key space, we can see that the splitting behavior of these two structures is very different. The BST splits the space based on the key values as they are encountered when going down the tree. But the splits in the key space are predetermined for the Huffman tree. Search tree structures whose splitting points in the key space are predetermined are given the special name **trie** to distinguish them from the type of search tree (like the BST) whose splitting points are determined by the data. Tries are discussed in more detail in Chapter 13.

5.7 Further Reading

See Shaffer and Brown [SB93] for an example of a tree implementation where an internal node pointer field stores the value of its child instead of a pointer to its child when the child is a leaf node.

Many techniques exist for maintaining reasonably balanced BSTs in the face of an unfriendly series of insert and delete operations. One example is the AVL tree of Adelson-Velskii and Landis, which is discussed by Knuth [Knu98]. The AVL tree (see Section 13.2) is actually a BST whose insert and delete routines reorganize the tree structure so as to guarantee that the subtrees rooted by the children of any node will differ in height by at most one. Another example is the splay tree [ST85], also discussed in Section 13.2.

See Bentley's Programming Pearl "Thanks, Heaps" [Ben85, Ben88] for a good discussion on the heap data structure and its uses.

The proof of Section 5.6.1 that the Huffman coding tree has minimum external path weight is from Knuth [Knu97]. For more information on data compression techniques, see *Managing Gigabytes* by Witten, Moffat, and Bell [WMB99], and *Codes and Cryptography* by Dominic Welsh [Wel88]. Tables 5.23 and 5.24 are derived from Welsh [Wel88].

5.8 Exercises

5.1 Section 5.1.1 claims that a full binary tree has the highest number of leaf nodes among all trees with n internal nodes. Prove that this is true.

5.2 Define the **degree** of a node as the number of its non-empty children. Prove by induction that the number of degree 2 nodes in any binary tree is one less than the number of leaves.

5.3 Define the **internal path length** for a tree as the sum of the depths of all internal nodes, while the **external path length** is the sum of the depths of all leaf nodes in the tree. Prove by induction that if tree **T** is a full binary tree with n internal nodes, I is **T**'s internal path length, and E is **T**'s external path length, then $E = I + 2n$ for $n \geq 0$.

5.4 Explain why function **`preorder2`** from Section 5.2 makes half as many recursive calls as function **`preorder`**. Explain why it makes twice as many accesses to left and right children.

5.5 **(a)** Modify the preorder traversal of Section 5.2 to perform an inorder traversal of a binary tree.

(b) Modify the preorder traversal of Section 5.2 to perform a postorder traversal of a binary tree.

5.6 Write a recursive function named **`search`** that takes as input the pointer to the root of a binary tree (*not* a BST!) and a value K, and returns **`true`** if value K appears in the tree and **`false`** otherwise.

5.7 Write an algorithm that takes as input the pointer to the root of a binary tree and prints the node values of the tree in **level** order. Level order first prints the root, then all nodes of level 1, then all nodes of level 2, and so on. *Hint*: Preorder traversals make use of a stack through recursive calls. Consider making use of another data structure to help implement the level-order traversal.

5.8 Write a recursive function that returns the height of a binary tree.

5.9 Write a recursive function that returns a count of the number of leaf nodes in a binary tree.

5.10 Assume that a given binary tree stores integer values in its nodes. Write a recursive function that sums the values of all nodes in the tree.

5.11 Assume that a given binary tree stores integer values in its nodes. Write a recursive function that traverses a binary tree, and prints the value of every node who's grandparent has a value that is a multiple of five.

5.12 Write a recursive function that traverses a binary tree, and prints the value of every node which has at least four great-grandchildren.

5.13 Compute the overhead fraction for each of the following full binary tree implementations.

(a) All nodes store data, two child pointers, and a parent pointer. The data field requires four bytes and each pointer requires four bytes.

(b) All nodes store data and two child pointers. The data field requires sixteen bytes and each pointer requires four bytes.

(c) All nodes store data and a parent pointer, and internal nodes store two child pointers. The data field requires eight bytes and each pointer requires four bytes.

(d) Only leaf nodes store data; internal nodes store two child pointers. The data field requires eight bytes and each pointer requires four bytes.

5.14 Why is the BST Property defined so that nodes with values equal to the value of the root appear only in the right subtree, rather than allow equal-valued nodes to appear in either subtree?

5.15 **(a)** Show the BST that results from inserting the values 15, 20, 25, 18, 16, 5, and 7 (in that order).
(b) Show the enumerations for the tree of (a) that result from doing a preorder traversal, an inorder traversal, and a postorder traversal.

5.16 Draw the BST that results from adding the value 5 to the BST shown in Figure 5.13(a).

5.17 Draw the BST that results from deleting the value 7 from the BST of Figure 5.13(b).

5.18 Write a function that prints out the node values for a BST in sorted order from highest to lowest.

5.19 Write a recursive function named **`smallcount`** that, given the pointer to the root of a BST and a key K, returns the number of nodes having key values less than or equal to K. Function **`smallcount`** should visit as few nodes in the BST as possible.

5.20 Write a recursive function named **`printRange`** that, given the pointer to the root of a BST, a low key value, and a high key value, prints in sorted order all records whose key values fall between the two given keys. Function **`printRange`** should visit as few nodes in the BST as possible.

5.21 Write a recursive function named **`checkBST`** that, given the pointer to the root of a binary tree, will return **`true`** if the tree is a BST, and **`false`** if it is not.

5.22 Describe a simple modification to the BST that will allow it to easily support finding the Kth smallest value in $\Theta(\log n)$ average case time. Then write a pseudo-code function for finding the Kth smallest value in your modified BST.

5.23 What are the minimum and maximum number of elements in a heap of height h?

5.24 Where in a max-heap might the smallest element reside?

5.25 Show the max-heap that results from running **`buildHeap`** on the following values stored in an array:

$$10 \quad 5 \quad 12 \quad 3 \quad 2 \quad 1 \quad 8 \quad 7 \quad 9 \quad 4$$

5.26 **(a)** Show the heap that results from deleting the maximum value from the max-heap of Figure 5.20b.
(b) Show the heap that results from deleting the element with value 5 from the max-heap of Figure 5.20b.

5.27 Revise the heap definition of Figure 5.19 to implement a min-heap. The member function **`removemax`** should be replaced by a new function called **`removemin`**.

5.28 Build the Huffman coding tree and determine the codes for the following set of letters and weights:

Letter	A	B	C	D	E	F	G	H	I	J	K	L
Frequency	2	3	5	7	11	13	17	19	23	31	37	41

What is the expected length in bits of a message containing n characters for this frequency distribution?

5.29 What will the Huffman coding tree look like for a set of sixteen characters all with equal weight? What is the average code length for a letter in this case? How does this differ from the smallest possible fixed length code for sixteen characters?

5.30 A set of characters with varying weights is assigned Huffman codes. If one of the characters is assigned code 001, then,

(a) Describe all codes that *cannot* have been assigned.
(b) Describe all codes that *must* have been assigned.

5.31 Assume that a sample alphabet has the following weights:

Letter	Q	Z	F	M	T	S	O	E
Frequency	2	3	10	10	10	15	20	30

(a) For this alphabet, what is the worst-case number of bits required by the Huffman code for a string of n letters? What string(s) have the worst-case performance?
(b) For this alphabet, what is the best-case number of bits required by the Huffman code for a string of n letters? What string(s) have the best-case performance?
(c) What is the average number of bits required by a character using the Huffman code for this alphabet?

5.32 You must keep track of some data. Your options are:

(1) A linked-list maintained in sorted order.
(2) A linked-list of unsorted records.
(3) A binary search tree.
(4) An array-based list maintained in sorted order.
(5) An array-based list of unsorted records.

For each of the following scenarios, which of these choices would be best? Explain your answer.

(a) The records are guaranteed to arrive already sorted from lowest to highest (i.e., whenever a record is inserted, its key value will always be greater than that of the last record inserted). A total of 1000 inserts will be interspersed with 1000 searches.
(b) The records arrive with values having a uniform random distribution (so the BST is likely to be well balanced). 1,000,000 insertions are performed, followed by 10 searches.

(c) The records arrive with values having a uniform random distribution (so the BST is likely to be well balanced). 1000 insertions are interspersed with 1000 searches.

(d) The records arrive with values having a uniform random distribution (so the BST is likely to be well balanced). 1000 insertions are performed, followed by 1,000,000 searches.

5.9 Projects

5.1 Re-implement the composite design for the binary tree node class of Figure 5.11 using a flyweight in place of **`NULL`** pointers to empty nodes.

5.2 One way to deal with the "problem" of **`NULL`** pointers in binary trees is to use that space for some other purpose. One example is the **threaded** binary tree. Extending the node implementation of Figure 5.7, the threaded binary tree stores with each node two additional bit fields that indicate if the child pointers **`lc`** and **`rc`** are regular pointers to child nodes or threads. If **`lc`** is not a pointer to a non-empty child (i.e., if it would be **`NULL`** in a regular binary tree), then it instead stores a pointer to the **inorder predecessor** of that node. The inorder predecessor is the node that would be printed immediately before the current node in an inorder traversal. If **`rc`** is not a pointer to a child, then it instead stores a pointer to the node's **inorder successor**. The inorder successor is the node that would be printed immediately after the current node in an inorder traversal. The main advantage of threaded binary trees is that operations such as inorder traversal can be implemented without using recursion or a stack.

Re-implement the BST as a threaded binary tree, and include a non-recursive version of the preorder traversal

5.3 Implement a city database using a BST to store the database records. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x- and y-coordinates. The BST should be organized by city name. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. Another operation that should be supported is to print all records within a given distance of a specified point. Collect running-time statistics for each operation. Which operations can be implemented reasonably efficiently (i.e., in $\Theta(\log n)$ time in the average case) using a BST? Can the database system be made more efficient by using one or more additional BSTs to organize the records by location?

5.4 Create a binary tree ADT that includes generic traversal methods that take a visitor, as described in Section 5.2. Write functions **`count`** and **`BSTcheck`** of Section 5.2 as visitors to be used with the generic traversal method.

5.5 Implement a priority queue class based on the max-heap class implementation of Figure 5.19. The following methods should be supported for manipulating the priority queue:

```
void enqueue(int ObjectID, int priority);
int dequeue();
void changeweight(int ObjectID, int newPriority);
```

Method **enqueue** inserts a new object into the priority queue with ID number **ObjectID** and priority **priority**. Method **dequeue** removes the object with highest priority from the priority queue and returns its object ID. Method **changeweight** changes the priority of the object with ID number **ObjectID** to be **newPriority**. The type for **E** should be a class that stores the object ID and the priority for that object. You will need a mechanism for finding the position of the desired object within the heap. Use an array, storing the object with **ObjectID** i in position i. (Be sure in your testing to keep the **ObjectID**s within the array bounds.) You must also modify the heap implementation to store the object's position in the auxiliary array so that updates to objects in the heap can be updated as well in the array.

5.6 The Huffman coding tree function **buildHuff** of Figure 5.29 manipulates a sorted list. This could result in a $\Theta(n^2)$ algorithm, because placing an intermediate Huffman tree on the list could take $\Theta(n)$ time. Revise this algorithm to use a priority queue based on a min-heap instead of a list.

5.7 Complete the implementation of the Huffman coding tree, building on the code presented in Section 5.6. Include a function to compute and store in a table the codes for each letter, and functions to encode and decode messages. This project can be further extended to support file compression. To do so requires adding two steps: (1) Read through the input file to generate actual frequencies for all letters in the file; and (2) store a representation for the Huffman tree at the beginning of the encoded output file to be used by the decoding function. If you have trouble with devising such a representation, see Section 6.5.

6

Non-Binary Trees

Many organizations are hierarchical in nature, such as the military and most businesses. Consider a company with a president and some number of vice presidents who report to the president. Each vice president has some number of direct subordinates, and so on. If we wanted to model this company with a data structure, it would be natural to think of the president in the root node of a tree, the vice presidents at level 1, and their subordinates at lower levels in the tree as we go down the organizational hierarchy.

Because the number of vice presidents is likely to be more than two, this company's organization cannot easily be represented by a binary tree. We need instead to use a tree whose nodes have an arbitrary number of children. Unfortunately, when we permit trees to have nodes with an arbitrary number of children, they become much harder to implement than binary trees. We consider such trees in this chapter. To distinguish them from binary trees, we use the term **general tree**.

Section 6.1 presents general tree terminology. Section 6.2 presents a simple representation for solving the important problem of processing equivalence classes. Several pointer-based implementations for general trees are covered in Section 6.3. Aside from general trees and binary trees, there are also uses for trees whose internal nodes have a fixed number K of children where K is something other than two. Such trees are known as K-ary trees. Section 6.4 generalizes the properties of binary trees to K-ary trees. Sequential representations, useful for applications such as storing trees on disk, are covered in Section 6.5.

6.1 General Tree Definitions and Terminology

A **tree T** is a finite set of one or more nodes such that there is one designated node *R*, called the root of **T**. If the set $(\mathbf{T} - \{R\})$ is not empty, these nodes are partitioned into $n > 0$ disjoint subsets $\mathbf{T}_0$, $\mathbf{T}_1$, ..., $\mathbf{T}_{n-1}$, each of which is a tree, and whose roots R_1, R_2, ..., R_n, respectively, are children of *R*. The subsets $\mathbf{T}_i$ $(0 \leq i < n)$ are said to be **subtrees** of **T**. These subtrees are ordered in that $\mathbf{T}_i$ is said to come before

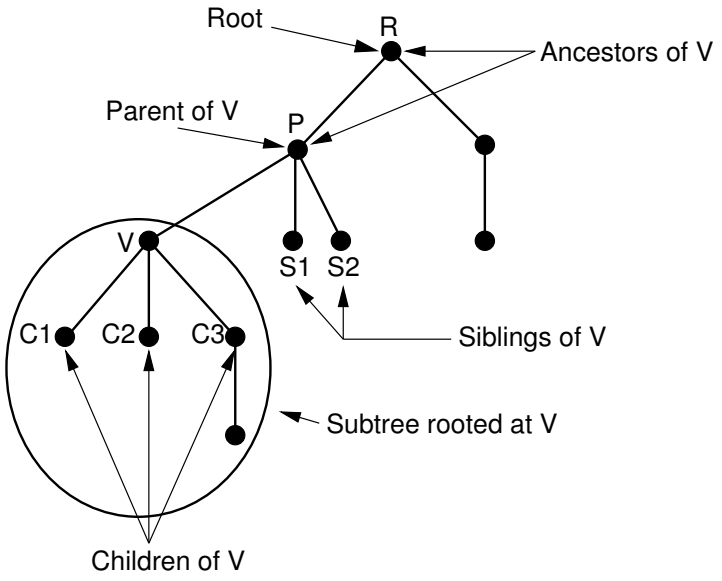


Figure 6.1 Notation for general trees. Node *P* is the parent of nodes *V*, *S1*, and *S2*. Thus, *V*, *S1*, and *S2* are children of *P*. Nodes *R* and *P* are ancestors of *V*. Nodes *V*, *S1*, and *S2* are called **siblings**. The oval surrounds the subtree having *V* as its root.

$\mathbf{T}_j$ if $i < j$. By convention, the subtrees are arranged from left to right with subtree $\mathbf{T}_0$ called the leftmost child of *R*. A node's **out degree** is the number of children for that node. A **forest** is a collection of one or more trees. Figure 6.1 presents further tree notation generalized from the notation for binary trees presented in Chapter 5.

Each node in a tree has precisely one parent, except for the root, which has no parent. From this observation, it immediately follows that a tree with n nodes must have $n - 1$ edges because each node, aside from the root, has one edge connecting that node to its parent.

6.1.1 An ADT for General Tree Nodes

Before discussing general tree implementations, we should first make precise what operations such implementations must support. Any implementation must be able to initialize a tree. Given a tree, we need access to the root of that tree. There must be some way to access the children of a node. In the case of the ADT for binary tree nodes, this was done by providing member functions that give explicit access to the left and right child pointers. Unfortunately, because we do not know in advance how many children a given node will have in the general tree, we cannot give explicit functions to access each child. An alternative must be found that works for an unknown number of children.

```
// General tree node ADT
template <typename E> class GTNode {
public:
  E value();                      // Return node's value
  bool isLeaf();                     // True if node is a leaf
  GTNode* parent();                  // Return parent
  GTNode* leftmostChild();           // Return first child
  GTNode* rightSibling();            // Return right sibling
  void setValue(E&);              // Set node's value
  void insertFirst(GTNode<E>*); // Insert first child
  void insertNext(GTNode<E>*);  // Insert next sibling
  void removeFirst();                // Remove first child
  void removeNext();                 // Remove right sibling
};

// General tree ADT
template <typename E> class GenTree {
public:
  void clear();             // Send all nodes to free store
  GTNode<E>* root();    // Return the root of the tree
  // Combine two subtrees
  void newroot(E&, GTNode<E>*, GTNode<E>*);
  void print();             // Print a tree
};
```

Figure 6.2 Definitions for the general tree and general tree node

One choice would be to provide a function that takes as its parameter the index for the desired child. That combined with a function that returns the number of children for a given node would support the ability to access any node or process all children of a node. Unfortunately, this view of access tends to bias the choice for node implementations in favor of an array-based approach, because these functions favor random access to a list of children. In practice, an implementation based on a linked list is often preferred.

An alternative is to provide access to the first (or leftmost) child of a node, and to provide access to the next (or right) sibling of a node. Figure 6.2 shows class declarations for general trees and their nodes. Based on these two access functions, the children of a node can be traversed like a list. Trying to find the next sibling of the rightmost sibling would return **NULL**.

6.1.2 General Tree Traversals

In Section 5.2, three tree traversals were presented for binary trees: preorder, postorder, and inorder. For general trees, preorder and postorder traversals are defined with meanings similar to their binary tree counterparts. Preorder traversal of a general tree first visits the root of the tree, then performs a preorder traversal of each subtree from left to right. A postorder traversal of a general tree performs a postorder traversal of the root's subtrees from left to right, then visits the root. Inorder

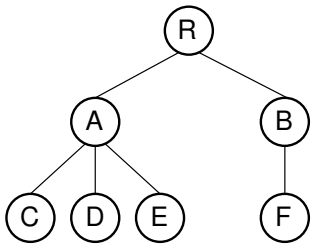


Figure 6.3 An example of a general tree.

traversal does not have a natural definition for the general tree, because there is no particular number of children for an internal node. An arbitrary definition — such as visit the leftmost subtree in inorder, then the root, then visit the remaining subtrees in inorder — can be invented. However, inorder traversals are generally not useful with general trees.

Example 6.1 A preorder traversal of the tree in Figure 6.3 visits the nodes in order $RACDEBF$.

A postorder traversal of this tree visits the nodes in order $CDEAFBR$.

To perform a preorder traversal, it is necessary to visit each of the children for a given node (say *R*) from left to right. This is accomplished by starting at *R*'s leftmost child (call it *T*). From *T*, we can move to *T*'s right sibling, and then to that node's right sibling, and so on.

Using the ADT of Figure 6.2, here is a C++ implementation to print the nodes of a general tree in preorder. Note the **for** loop at the end, which processes the list of children by beginning with the leftmost child, then repeatedly moving to the next child until calling **next** returns **NULL**.

```
// Print using a preorder traversal
void printhelp(GTNode<E>* root) {
  if (root->isLeaf()) cout << "Leaf: ";
  else cout << "Internal: ";
  cout << root->value() << "\n";
  // Now process the children of "root"
  for (GTNode<E>* temp = root->leftmostChild();
       temp != NULL; temp = temp->rightSibling())
    printhelp(temp);
}
```

6.2 The Parent Pointer Implementation

Perhaps the simplest general tree implementation is to store for each node only a pointer to that node's parent. We will call this the **parent pointer** implementation. Clearly this implementation is not general purpose, because it is inadequate for such important operations as finding the leftmost child or the right sibling for a node. Thus, it may seem to be a poor idea to implement a general tree in this way. However, the parent pointer implementation stores precisely the information required to answer the following, useful question: "Given two nodes, are they in the same tree?" To answer the question, we need only follow the series of parent pointers from each node to its respective root. If both nodes reach the same root, then they must be in the same tree. If the roots are different, then the two nodes are not in the same tree. The process of finding the ultimate root for a given node we will call **FIND**.

The parent pointer representation is most often used to maintain a collection of disjoint sets. Two disjoint sets share no members in common (their intersection is empty). A collection of disjoint sets partitions some objects such that every object is in exactly one of the disjoint sets. There are two basic operations that we wish to support:

(1) determine if two objects are in the same set, and

(2) merge two sets together.

Because two merged sets are united, the merging operation is called UNION and the whole process of determining if two objects are in the same set and then merging the sets goes by the name "UNION/FIND."

To implement UNION/FIND, we represent each disjoint set with a separate general tree. Two objects are in the same disjoint set if they are in the same tree. Every node of the tree (except for the root) has precisely one parent. Thus, each node requires the same space to represent it. The collection of objects is typically stored in an array, where each element of the array corresponds to one object, and each element stores the object's value. The objects also correspond to nodes in the various disjoint trees (one tree for each disjoint set), so we also store the parent value with each object in the array. Those nodes that are the roots of their respective trees store an appropriate indicator. Note that this representation means that a single array is being used to implement a collection of trees. This makes it easy to merge trees together with UNION operations.

Figure 6.4 shows the parent pointer implementation for the general tree, called **`ParPtrTree`**. This class is greatly simplified from the declarations of Figure 6.2 because we need only a subset of the general tree operations. Instead of implementing a separate node class, **`ParPtrTree`** simply stores an array where each array element corresponds to a node of the tree. Each position i of the array stores the value for node i and the array position for the parent of node i. Class **`ParPtrTree`**

```
// General tree representation for UNION/FIND
class ParPtrTree {
private:
  int* array;                    // Node array
  int size;                      // Size of node array
  int FIND(int) const;           // Find root
public:
  ParPtrTree(int);                   // Constructor
  ~ParPtrTree() { delete [] array; } // Destructor
  void UNION(int, int);          // Merge equivalences
  bool differ(int, int);         // True if not in same tree
};

int ParPtrTree::FIND(int curr) const { // Find root
  while (array[curr] != ROOT) curr = array[curr];
  return curr;  // At root
}
```

Figure 6.4 General tree implementation using parent pointers for the UNION/FIND algorithm.

is given two new methods, **differ** and **UNION**. Method **differ** checks if two objects are in different sets, and method **UNION** merges two sets together. A private method **FIND** is used to find the ultimate root for an object.

An application using the UNION/FIND operations should store a set of n objects, where each object is assigned a unique index in the range 0 to $n-1$. The indices refer to the corresponding parent pointers in the array. Class **ParPtrTree** creates and initializes the UNION/FIND array, and methods **differ** and **UNION** take array indices as inputs.

Figure 6.5 illustrates the parent pointer implementation. Note that the nodes can appear in any order within the array, and the array can store up to n separate trees. For example, Figure 6.5 shows two trees stored in the same array. Thus, a single array can store a collection of items distributed among an arbitrary (and changing) number of disjoint subsets.

Consider the problem of assigning the members of a set to disjoint subsets called **equivalence classes**. Recall from Section 2.1 that an equivalence relation is reflexive, symmetric, and transitive. Thus, if objects *A* and *B* are equivalent, and objects *B* and *C* are equivalent, we must be able to recognize that objects *A* and *C* are also equivalent.

There are many practical uses for disjoint sets and representing equivalences. For example, consider Figure 6.6 which shows a graph of ten nodes labeled *A* through *J*. Notice that for nodes *A* through *I*, there is some series of edges that connects any pair of the nodes, but node *J* is disconnected from the rest of the nodes. Such a graph might be used to represent connections such as wires between components on a circuit board, or roads between cities. We can consider two nodes of the graph to be equivalent if there is a path between them. Thus,

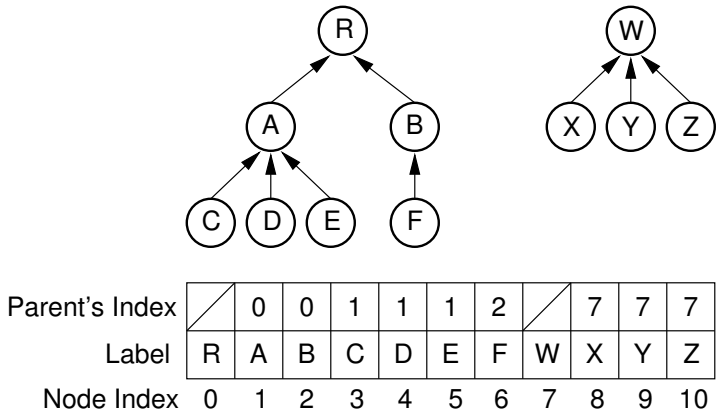


Figure 6.5 The parent pointer array implementation. Each node corresponds to a position in the node array, which stores its value and a pointer to its parent. The parent pointers are represented by the position in the array of the parent. The root of any tree stores **ROOT**, represented graphically by a slash in the "Parent's Index" box. This figure shows two trees stored in the same parent pointer array, one rooted at *R*, and the other rooted at *W*.

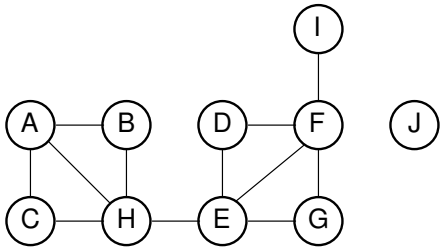


Figure 6.6 A graph with two connected components.

nodes *A*, *H*, and *E* would be equivalent in Figure 6.6, but *J* is not equivalent to any other. A subset of equivalent (connected) edges in a graph is called a **connected component**. The goal is to quickly classify the objects into disjoint sets that correspond to the connected components. Another application for UNION/FIND occurs in Kruskal's algorithm for computing the minimal cost spanning tree for a graph (Section 11.5.2).

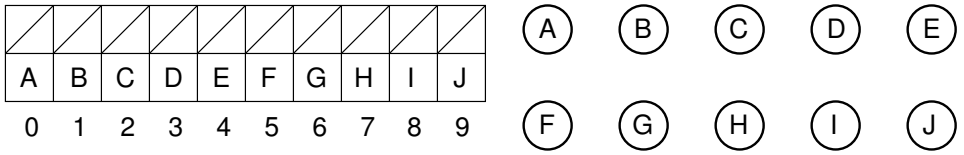
The input to the UNION/FIND algorithm is typically a series of equivalence pairs. In the case of the connected components example, the equivalence pairs would simply be the set of edges in the graph. An equivalence pair might say that object *C* is equivalent to object *A*. If so, *C* and *A* are placed in the same subset. If a later equivalence relates *A* and *B*, then by implication *C* is also equivalent to *B*. Thus, an equivalence pair may cause two subsets to merge, each of which contains several objects.

Equivalence classes can be managed efficiently with the UNION/FIND algorithm. Initially, each object is at the root of its own tree. An equivalence pair is processed by checking to see if both objects of the pair are in the same tree using method **`differ`**. If they are in the same tree, then no change need be made because the objects are already in the same equivalence class. Otherwise, the two equivalence classes should be merged by the **`UNION`** method.

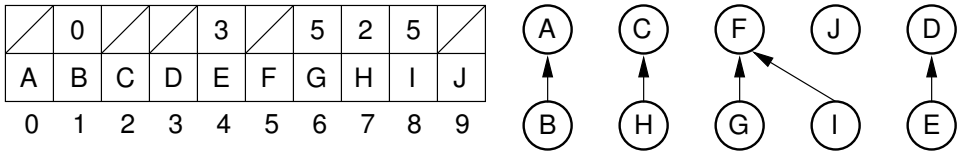
Example 6.2 As an example of solving the equivalence class problem, consider the graph of Figure 6.6. Initially, we assume that each node of the graph is in a distinct equivalence class. This is represented by storing each as the root of its own tree. Figure 6.7(a) shows this initial configuration using the parent pointer array representation. Now, consider what happens when equivalence relationship (*A*, *B*) is processed. The root of the tree containing *A* is *A*, and the root of the tree containing *B* is *B*. To make them equivalent, one of these two roots is set to be the parent of the other. In this case it is irrelevant which points to which, so we arbitrarily select the first in alphabetical order to be the root. This is represented in the parent pointer array by setting the parent field of *B* (the node in array position 1 of the array) to store a pointer to *A*. Equivalence pairs (*C*, *H*), (*G*, *F*), and (*D*, *E*) are processed in similar fashion. When processing the equivalence pair (*I*, *F*), because *I* and *F* are both their own roots, *I* is set to point to *F*. Note that this also makes *G* equivalent to *I*. The result of processing these five equivalences is shown in Figure 6.7(b).

The parent pointer representation places no limit on the number of nodes that can share a parent. To make equivalence processing as efficient as possible, the distance from each node to the root of its respective tree should be as small as possible. Thus, we would like to keep the height of the trees small when merging two equivalence classes together. Ideally, each tree would have all nodes pointing directly to the root. Achieving this goal all the time would require too much additional processing to be worth the effort, so we must settle for getting as close as possible.

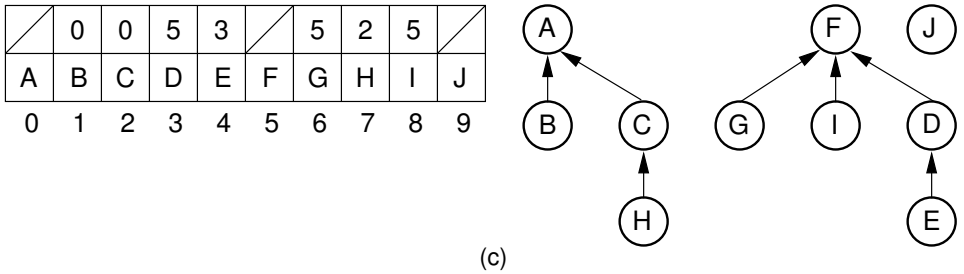
A low-cost approach to reducing the height is to be smart about how two trees are joined together. One simple technique, called the **weighted union rule**, joins the tree with fewer nodes to the tree with more nodes by making the smaller tree's root point to the root of the bigger tree. This will limit the total depth of the tree to $\mathrm{O}(\log n)$, because the depth of nodes only in the smaller tree will now increase by one, and the depth of the deepest node in the combined tree can only be at most one deeper than the deepest node before the trees were combined. The total number of nodes in the combined tree is therefore at least twice the number in the smaller



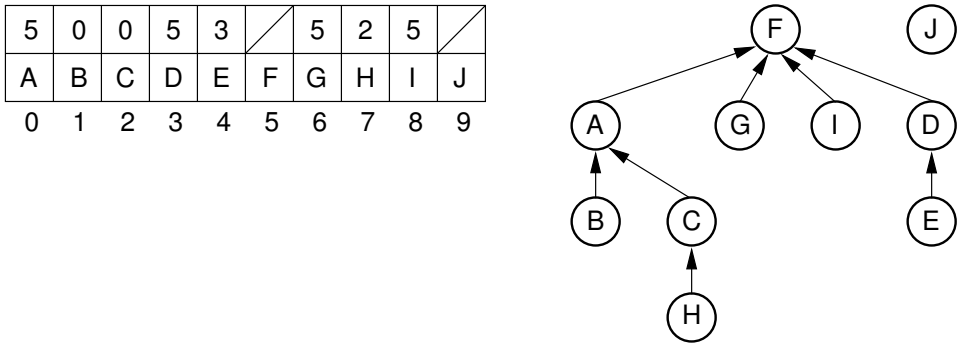
(a)



(b)



(c)



(d)

Figure 6.7 An example of equivalence processing. (a) Initial configuration for the ten nodes of the graph in Figure 6.6. The nodes are placed into ten independent equivalence classes. (b) The result of processing five edges: (A, B), (C, H), (G, F), (D, E), and (I, F). (c) The result of processing two more edges: (H, A) and (E, G). (d) The result of processing edge (H, E).

subtree. Thus, the depth of any node can be increased at most $\log n$ times when n equivalences are processed.

Example 6.3 When processing equivalence pair (*I*, *F*) in Figure 6.7(b), *F* is the root of a tree with two nodes while *I* is the root of a tree with only one node. Thus, *I* is set to point to *F* rather than the other way around. Figure 6.7(c) shows the result of processing two more equivalence pairs: (*H*, *A*) and (*E*, *G*). For the first pair, the root for *H* is *C* while the root for *A* is itself. Both trees contain two nodes, so it is an arbitrary decision as to which node is set to be the root for the combined tree. In the case of equivalence pair (*E*, *G*), the root of *E* is *D* while the root of *G* is *F*. Because *F* is the root of the larger tree, node *D* is set to point to *F*.

Not all equivalences will combine two trees. If equivalence (*F*, *G*) is processed when the representation is in the state shown in Figure 6.7(c), no change will be made because *F* is already the root for *G*.

The weighted union rule helps to minimize the depth of the tree, but we can do better than this. **Path compression** is a method that tends to create extremely shallow trees. Path compression takes place while finding the root for a given node *X*. Call this root *R*. Path compression resets the parent of every node on the path from *X* to *R* to point directly to *R*. This can be implemented by first finding *R*. A second pass is then made along the path from *X* to *R*, assigning the parent field of each node encountered to *R*. Alternatively, a recursive algorithm can be implemented as follows. This version of **FIND** not only returns the root of the current node, but also makes all ancestors of the current node point to the root.

```
// FIND with path compression
int ParPtrTree::FIND(int curr) const {
  if (array[curr] == ROOT) return curr; // At root
  array[curr] = FIND(array[curr]);
  return array[curr];
}
```

Example 6.4 Figure 6.7(d) shows the result of processing equivalence pair (*H*, *E*) on the the representation shown in Figure 6.7(c) using the standard weighted union rule without path compression. Figure 6.8 illustrates the path compression process for the same equivalence pair. After locating the root for node *H*, we can perform path compression to make *H* point directly to root object *A*. Likewise, *E* is set to point directly to its root, *F*. Finally, object *A* is set to point to root object *F*.

Note that path compression takes place during the FIND operation, *not* during the UNION operation. In Figure 6.8, this means that nodes *B*, *C*, and *H* have node *A* remain as their parent, rather than changing their parent to

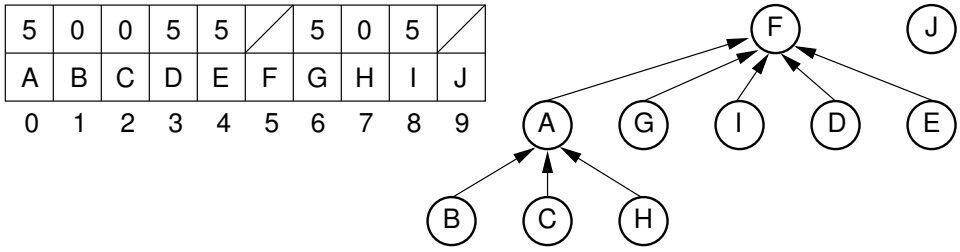


Figure 6.8 An example of path compression, showing the result of processing equivalence pair (*H*, *E*) on the representation of Figure 6.7(c).

be *F*. While we might prefer to have these nodes point to *F*, to accomplish this would require that additional information from the FIND operation be passed back to the UNION operation. This would not be practical.

Path compression keeps the cost of each FIND operation very close to constant. To be more precise about what is meant by "very close to constant," the cost of path compression for n FIND operations on n nodes (when combined with the weighted union rule for joining sets) is approximately[1] $\Theta(n \log^* n)$. The notation "$\log^* n$" means the number of times that the log of n must be taken before $n \leq 1$. For example, $\log^* 65536$ is 4 because $\log 65536 = 16$, $\log 16 = 4$, $\log 4 = 2$, and finally $\log 2 = 1$. Thus, $\log^* n$ grows *very* slowly, so the cost for a series of n FIND operations is very close to n.

Note that this does not mean that the tree resulting from processing n equivalence pairs necessarily has depth $\Theta(\log^* n)$. One can devise a series of equivalence operations that yields $\Theta(\log n)$ depth for the resulting tree. However, many of the equivalences in such a series will look only at the roots of the trees being merged, requiring little processing time. The *total* amount of processing time required for n operations will be $\Theta(n \log^* n)$, yielding nearly constant time for each equivalence operation. This is an example of amortized analysis, discussed further in Section 14.3.

6.3 General Tree Implementations

We now tackle the problem of devising an implementation for general trees that allows efficient processing for all member functions of the ADTs shown in Figure 6.2. This section presents several approaches to implementing general trees. Each implementation yields advantages and disadvantages in the amount of space required to store a node and the relative ease with which key operations can be performed. General tree implementations should place no restriction on how many

[1]To be more precise, this cost has been found to grow in time proportional to the inverse of Ackermann's function. See Section 6.6.

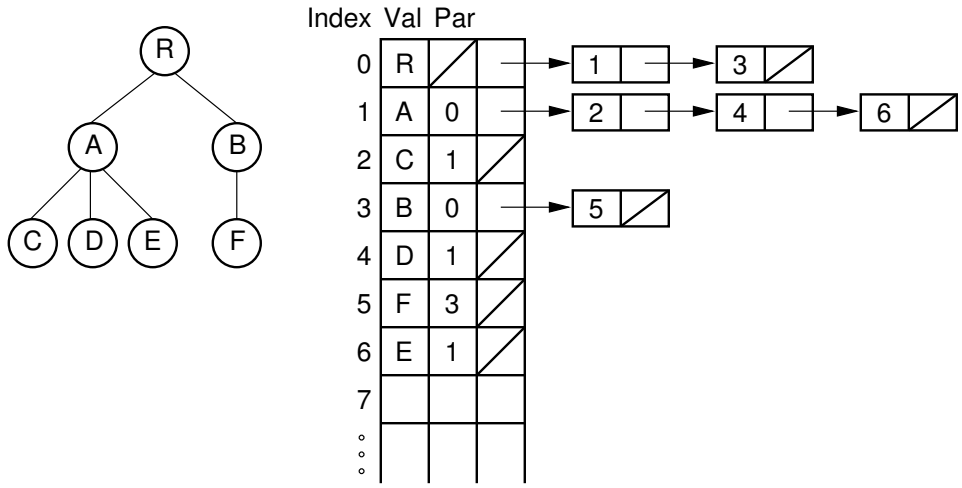


Figure 6.9 The "list of children" implementation for general trees. The column of numbers to the left of the node array labels the array indices. The column labeled "Val" stores node values. The column labeled "Par" stores indices (or pointers) to the parents. The last column stores pointers to the linked list of children for each internal node. Each element of the linked list stores a pointer to one of the node's children (shown as the array index of the target node).

children a node may have. In some applications, once a node is created the number of children never changes. In such cases, a fixed amount of space can be allocated for the node when it is created, based on the number of children for the node. Matters become more complicated if children can be added to or deleted from a node, requiring that the node's space allocation be adjusted accordingly.

6.3.1 List of Children

Our first attempt to create a general tree implementation is called the "list of children" implementation for general trees. It simply stores with each internal node a linked list of its children. This is illustrated by Figure 6.9.

The "list of children" implementation stores the tree nodes in an array. Each node contains a value, a pointer (or index) to its parent, and a pointer to a linked list of the node's children, stored in order from left to right. Each linked list element contains a pointer to one child. Thus, the leftmost child of a node can be found directly because it is the first element in the linked list. However, to find the right sibling for a node is more difficult. Consider the case of a node M and its parent P. To find M's right sibling, we must move down the child list of P until the linked list element storing the pointer to M has been found. Going one step further takes us to the linked list element that stores a pointer to M's right sibling. Thus, in the worst case, to find M's right sibling requires that all children of M's parent be searched.

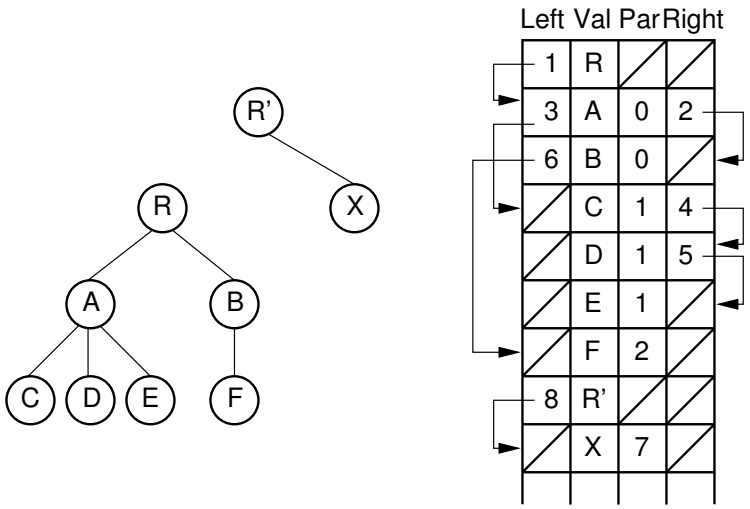


Figure 6.10 The "left-child/right-sibling" implementation.

Combining trees using this representation is difficult if each tree is stored in a separate node array. If the nodes of both trees are stored in a single node array, then adding tree **T** as a subtree of node *R* is done by simply adding the root of **T** to *R*'s list of children.

6.3.2 The Left-Child/Right-Sibling Implementation

With the "list of children" implementation, it is difficult to access a node's right sibling. Figure 6.10 presents an improvement. Here, each node stores its value and pointers to its parent, leftmost child, and right sibling. Thus, each of the basic ADT operations can be implemented by reading a value directly from the node. If two trees are stored within the same node array, then adding one as the subtree of the other simply requires setting three pointers. Combining trees in this way is illustrated by Figure 6.11. This implementation is more space efficient than the "list of children" implementation, and each node requires a fixed amount of space in the node array.

6.3.3 Dynamic Node Implementations

The two general tree implementations just described use an array to store the collection of nodes. In contrast, our standard implementation for binary trees stores each node as a separate dynamic object containing its value and pointers to its two children. Unfortunately, nodes of a general tree can have any number of children, and this number may change during the life of the node. A general tree node implementation must support these properties. One solution is simply to limit the number of children permitted for any node and allocate pointers for exactly that number of

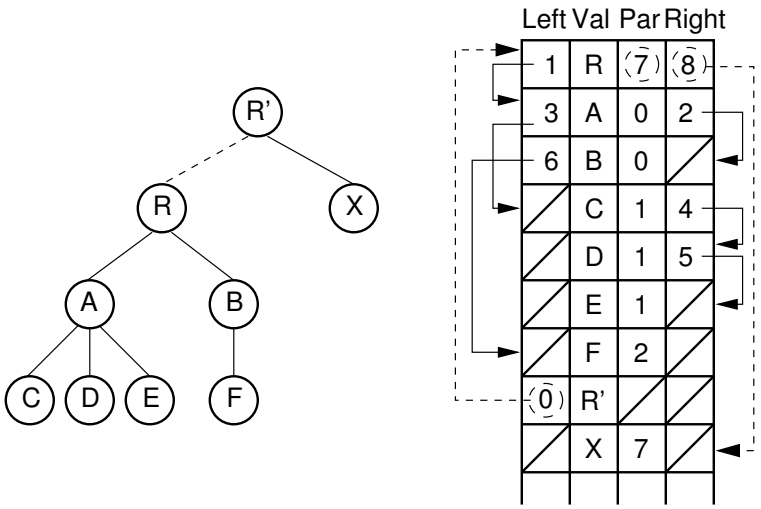


Figure 6.11 Combining two trees that use the "left-child/right-sibling" implementation. The subtree rooted at R in Figure 6.10 now becomes the first child of R'. Three pointers are adjusted in the node array: The left-child field of R' now points to node R, while the right-sibling field for R points to node X. The parent field of node R points to node R'.

children. There are two major objections to this. First, it places an undesirable limit on the number of children, which makes certain trees unrepresentable by this implementation. Second, this might be extremely wasteful of space because most nodes will have far fewer children and thus leave some pointer positions empty.

The alternative is to allocate variable space for each node. There are two basic approaches. One is to allocate an array of child pointers as part of the node. In essence, each node stores an array-based list of child pointers. Figure 6.12 illustrates the concept. This approach assumes that the number of children is known when the node is created, which is true for some applications but not for others. It also works best if the number of children does not change. If the number of children does change (especially if it increases), then some special recovery mechanism must be provided to support a change in the size of the child pointer array. One possibility is to allocate a new node of the correct size from free store and return the old copy of the node to free store for later reuse. This works especially well in a language with built-in garbage collection such as Java. For example, assume that a node M initially has two children, and that space for two child pointers is allocated when M is created. If a third child is added to M, space for a new node with three child pointers can be allocated, the contents of M is copied over to the new space, and the old space is then returned to free store. As an alternative to relying on the system's garbage collector, a memory manager for variable size storage units can be implemented, as described in Section 12.3. Another possibility is to use a collection of free lists, one for each array size, as described in Section 4.1.2. Note

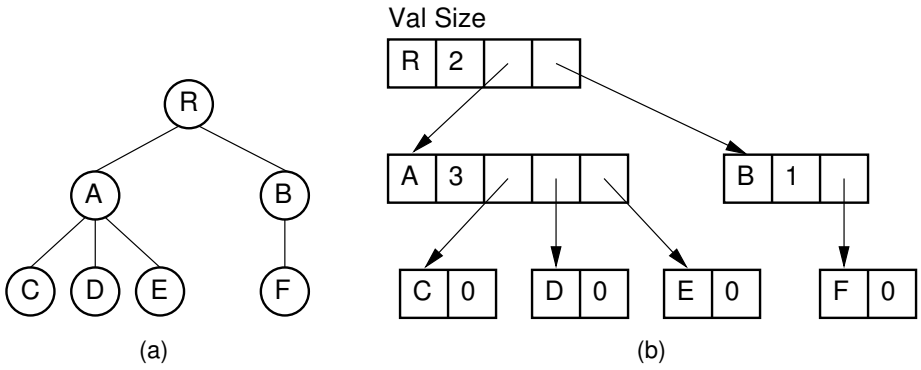


Figure 6.12 A dynamic general tree representation with fixed-size arrays for the child pointers. (a) The general tree. (b) The tree representation. For each node, the first field stores the node value while the second field stores the size of the child pointer array.

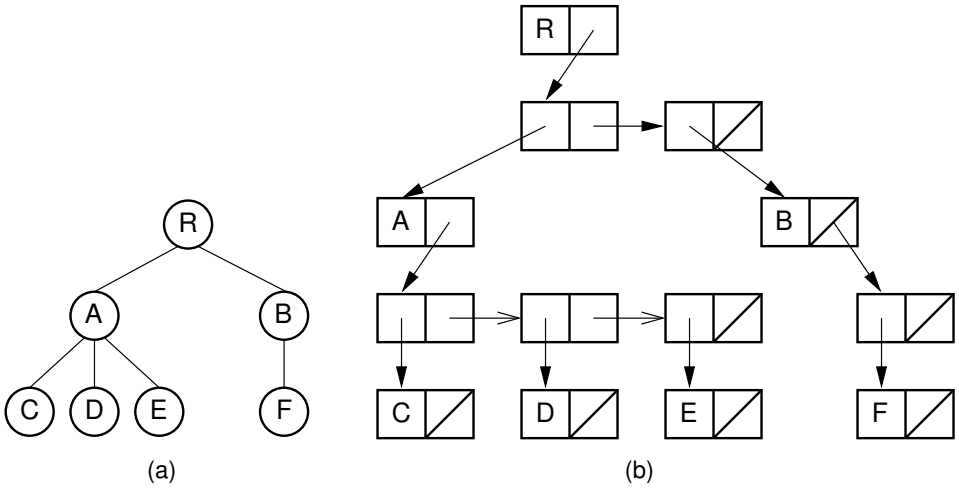


Figure 6.13 A dynamic general tree representation with linked lists of child pointers. (a) The general tree. (b) The tree representation.

in Figure 6.12 that the current number of children for each node is stored explicitly in a **`size`** field. The child pointers are stored in an array with **`size`** elements.

Another approach that is more flexible, but which requires more space, is to store a linked list of child pointers with each node as illustrated by Figure 6.13. This implementation is essentially the same as the "list of children" implementation of Section 6.3.1, but with dynamically allocated nodes rather than storing the nodes in an array.

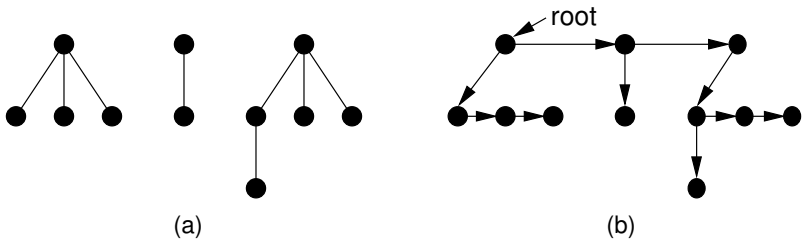


Figure 6.14 Converting from a forest of general trees to a single binary tree. Each node stores pointers to its left child and right sibling. The tree roots are assumed to be siblings for the purpose of converting.

6.3.4 Dynamic "Left-Child/Right-Sibling" Implementation

The "left-child/right-sibling" implementation of Section 6.3.2 stores a fixed number of pointers with each node. This can be readily adapted to a dynamic implementation. In essence, we substitute a binary tree for a general tree. Each node of the "left-child/right-sibling" implementation points to two "children" in a new binary tree structure. The left child of this new structure is the node's first child in the general tree. The right child is the node's right sibling. We can easily extend this conversion to a forest of general trees, because the roots of the trees can be considered siblings. Converting from a forest of general trees to a single binary tree is illustrated by Figure 6.14. Here we simply include links from each node to its right sibling and remove links to all children except the leftmost child. Figure 6.15 shows how this might look in an implementation with two pointers at each node. Compared with the implementation illustrated by Figure 6.13 which requires overhead of three pointers/node, the implementation of Figure 6.15 only requires two pointers per node. The representation of Figure 6.15 is likely to be easier to implement, space efficient, and more flexible than the other implementations presented in this section.

6.4 K-ary Trees

K-ary trees are trees whose internal nodes all have exactly K children. Thus, a full binary tree is a 2-ary tree. The PR quadtree discussed in Section 13.3 is an example of a 4-ary tree. Because K-ary tree nodes have a fixed number of children, unlike general trees, they are relatively easy to implement. In general, K-ary trees bear many similarities to binary trees, and similar implementations can be used for K-ary tree nodes. Note that as K becomes large, the potential number of **`NULL`** pointers grows, and the difference between the required sizes for internal nodes and leaf nodes increases. Thus, as K becomes larger, the need to choose separate implementations for the internal and leaf nodes becomes more pressing.

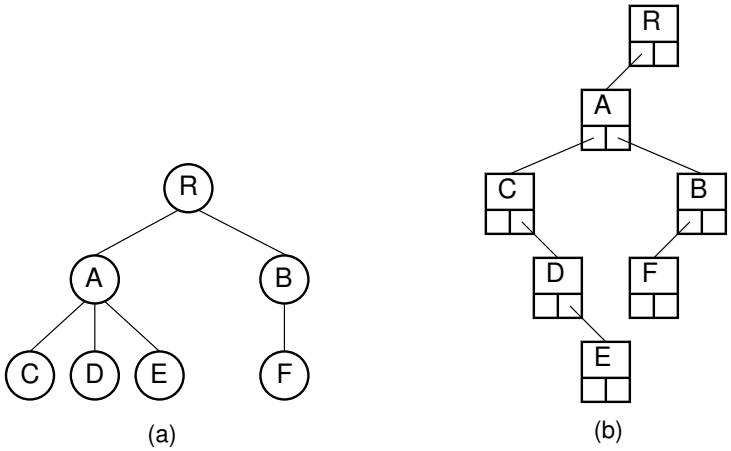


Figure 6.15 A general tree converted to the dynamic "left-child/right-sibling" representation. Compared to the representation of Figure 6.13, this representation requires less space.

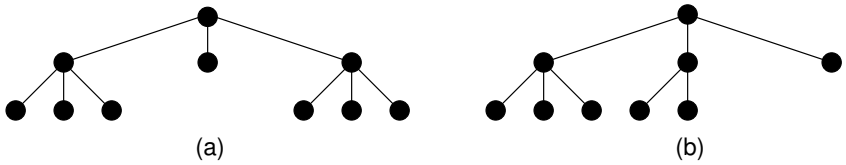


Figure 6.16 Full and complete 3-ary trees. (a) This tree is full (but not complete). (b) This tree is complete (but not full).

Full and **complete** K-ary trees are analogous to full and complete binary trees, respectively. Figure 6.16 shows full and complete K-ary trees for $K = 3$. In practice, most applications of K-ary trees limit them to be either full or complete.

Many of the properties of binary trees extend to K-ary trees. Equivalent theorems to those in Section 5.1.1 regarding the number of NULL pointers in a K-ary tree and the relationship between the number of leaves and the number of internal nodes in a K-ary tree can be derived. We can also store a complete K-ary tree in an array, using simple formulas to compute a node's relations in a manner similar to that used in Section 5.3.3.

6.5 Sequential Tree Implementations

Next we consider a fundamentally different approach to implementing trees. The goal is to store a series of node values with the minimum information needed to reconstruct the tree structure. This approach, known as a **sequential** tree implementation, has the advantage of saving space because no pointers are stored. It has

the disadvantage that accessing any node in the tree requires sequentially processing all nodes that appear before it in the node list. In other words, node access must start at the beginning of the node list, processing nodes sequentially in whatever order they are stored until the desired node is reached. Thus, one primary virtue of the other implementations discussed in this section is lost: efficient access (typically $\Theta(\log n)$ time) to arbitrary nodes in the tree. Sequential tree implementations are ideal for archiving trees on disk for later use because they save space, and the tree structure can be reconstructed as needed for later processing.

Sequential tree implementations can be used to **serialize** a tree structure. Serialization is the process of storing an object as a series of bytes, typically so that the data structure can be transmitted between computers. This capability is important when using data structures in a distributed processing environment.

A sequential tree implementation typically stores the node values as they would be enumerated by a preorder traversal, along with sufficient information to describe the tree's shape. If the tree has restricted form, for example if it is a full binary tree, then less information about structure typically needs to be stored. A general tree, because it has the most flexible shape, tends to require the most additional shape information. There are many possible sequential tree implementation schemes. We will begin by describing methods appropriate to binary trees, then generalize to an implementation appropriate to a general tree structure.

Because every node of a binary tree is either a leaf or has two (possibly empty) children, we can take advantage of this fact to implicitly represent the tree's structure. The most straightforward sequential tree implementation lists every node value as it would be enumerated by a preorder traversal. Unfortunately, the node values alone do not provide enough information to recover the shape of the tree. In particular, as we read the series of node values, we do not know when a leaf node has been reached. However, we can treat all non-empty nodes as internal nodes with two (possibly empty) children. Only **`NULL`** values will be interpreted as leaf nodes, and these can be listed explicitly. Such an augmented node list provides enough information to recover the tree structure.

Example 6.5 For the binary tree of Figure 6.17, the corresponding sequential representation would be as follows (assuming that '/' stands for **`NULL`**):

$$AB/D//CEG///FH//I// \tag{6.1}$$

To reconstruct the tree structure from this node list, we begin by setting node A to be the root. A's left child will be node B. Node B's left child is a **`NULL`** pointer, so node D must be B's right child. Node D has two **`NULL`** children, so node C must be the right child of node A.

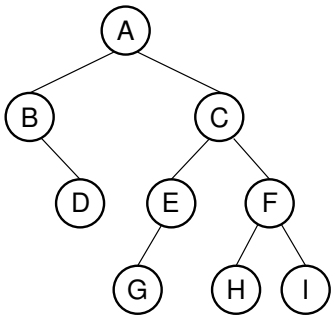


Figure 6.17 Sample binary tree for sequential tree implementation examples.

To illustrate the difficulty involved in using the sequential tree representation for processing, consider searching for the right child of the root node. We must first move sequentially through the node list of the left subtree. Only at this point do we reach the value of the root's right child. Clearly the sequential representation is space efficient, but not time efficient for descending through the tree along some arbitrary path.

Assume that each node value takes a constant amount of space. An example would be if the node value is a positive integer and **NULL** is indicated by the value zero. From the Full Binary Tree Theorem of Section 5.1.1, we know that the size of the node list will be about twice the number of nodes (i.e., the overhead fraction is 1/2). The extra space is required by the **NULL** pointers. We should be able to store the node list more compactly. However, any sequential implementation must recognize when a leaf node has been reached, that is, a leaf node indicates the end of a subtree. One way to do this is to explicitly list with each node whether it is an internal node or a leaf. If a node X is an internal node, then we know that its two children (which may be subtrees) immediately follow X in the node list. If X is a leaf node, then the next node in the list is the right child of some ancestor of X, not the right child of X. In particular, the next node will be the child of X's most recent ancestor that has not yet seen its right child. However, this assumes that each internal node does in fact have two children, in other words, that the tree is full. Empty children must be indicated in the node list explicitly. Assume that internal nodes are marked with a prime ($'$) and that leaf nodes show no mark. Empty children of internal nodes are indicated by '/', but the (empty) children of leaf nodes are not represented at all. Note that a full binary tree stores no **NULL** values with this implementation, and so requires less overhead.

Example 6.6 We can represent the tree of Figure 6.17 as follows:

$$A'B'/DC'E'G/F'HI \tag{6.2}$$

Note that slashes are needed for the empty children because this is not a full binary tree.

Storing n extra bits can be a considerable savings over storing n **`NULL`** values. In Example 6.6, each node is shown with a mark if it is internal, or no mark if it is a leaf. This requires that each node value has space to store the mark bit. This might be true if, for example, the node value were stored as a 4-byte integer but the range of the values sored was small enough so that not all bits are used. An example would be if all node values must be positive. Then the high-order (sign) bit of the integer value could be used as the mark bit.

Another approach is to store a separate bit vector to represent the status of each node. In this case, each node of the tree corresponds to one bit in the bit vector. A value of '1' could indicate an internal node, and '0' could indicate a leaf node.

Example 6.7 The bit vector for the tree if Figure 6.17 (including positions for the null children of nodes *B* and *E*) would be

$$11001100100 \tag{6.3}$$

Storing general trees by means of a sequential implementation requires that more explicit structural information be included with the node list. Not only must the general tree implementation indicate whether a node is leaf or internal, it must also indicate how many children the node has. Alternatively, the implementation can indicate when a node's child list has come to an end. The next example dispenses with marks for internal or leaf nodes. Instead it includes a special mark (we will use the ")" symbol) to indicate the end of a child list. All leaf nodes are followed by a ")" symbol because they have no children. A leaf node that is also the last child for its parent would indicate this by two or more successive ")" symbols.

Example 6.8 For the general tree of Figure 6.3, we get the sequential representation

$$RAC)D)E))BF))) \tag{6.4}$$

Note that *F* is followed by three ")" marks, because it is a leaf, the last node of *B*'s rightmost subtree, and the last node of *R*'s rightmost subtree.

Note that this representation for serializing general trees cannot be used for binary trees. This is because a binary tree is not merely a restricted form of general tree with at most two children. Every binary tree node has a left and a right child, though either or both might be empty. For example, the representation of Example 6.8 cannot let us distinguish whether node *D* in Figure 6.17 is the left or right child of node *B*.

6.6 Further Reading

The expression $\log^* n$ cited in Section 6.2 is closely related to the inverse of Ackermann's function. For more information about Ackermann's function and the cost of path compression for UNION/FIND, see Robert E. Tarjan's paper "On the efficiency of a good but not linear set merging algorithm" [Tar75]. The article "Data Structures and Algorithms for Disjoint Set Union Problems" by Galil and Italiano [GI91] covers many aspects of the equivalence class problem.

Foundations of Multidimensional and Metric Data Structures by Hanan Samet [Sam06] treats various implementations of tree structures in detail within the context of K-ary trees. Samet covers sequential implementations as well as the linked and array implementations such as those described in this chapter and Chapter 5. While these books are ostensibly concerned with spatial data structures, many of the concepts treated are relevant to anyone who must implement tree structures.

6.7 Exercises

6.1 Write an algorithm to determine if two general trees are identical. Make the algorithm as efficient as you can. Analyze your algorithm's running time.

6.2 Write an algorithm to determine if two binary trees are identical when the ordering of the subtrees for a node is ignored. For example, if a tree has root node with value *R*, left child with value *A* and right child with value *B*, this would be considered identical to another tree with root node value *R*, left child value *B*, and right child value *A*. Make the algorithm as efficient as you can. Analyze your algorithm's running time. How much harder would it be to make this algorithm work on a general tree?

6.3 Write a postorder traversal function for general trees, similar to the preorder traversal function named **`preorder`** given in Section 6.1.2.

6.4 Write a function that takes as input a general tree and returns the number of nodes in that tree. Write your function to use the **`GenTree`** and **`GTNode`** ADTs of Figure 6.2.

6.5 Describe how to implement the weighted union rule efficiently. In particular, describe what information must be stored with each node and how this information is updated when two trees are merged. Modify the implementation of Figure 6.4 to support the weighted union rule.

6.6 A potential alternative to the weighted union rule for combining two trees is the height union rule. The height union rule requires that the root of the tree with greater height become the root of the union. Explain why the height union rule can lead to worse average time behavior than the weighted union rule.

6.7 Using the weighted union rule and path compression, show the array for the parent pointer implementation that results from the following series of

equivalences on a set of objects indexed by the values 0 through 15. Initially, each element in the set should be in a separate equivalence class. When two trees to be merged are the same size, make the root with greater index value be the child of the root with lesser index value.

(0, 2) (1, 2) (3, 4) (3, 1) (3, 5) (9, 11) (12, 14) (3, 9) (4, 14) (6, 7) (8, 10) (8, 7) (7, 0) (10, 15) (10, 13)

6.8 Using the weighted union rule and path compression, show the array for the parent pointer implementation that results from the following series of equivalences on a set of objects indexed by the values 0 through 15. Initially, each element in the set should be in a separate equivalence class. When two trees to be merged are the same size, make the root with greater index value be the child of the root with lesser index value.

(2, 3) (4, 5) (6, 5) (3, 5) (1, 0) (7, 8) (1, 8) (3, 8) (9, 10) (11, 14) (11, 10) (12, 13) (11, 13) (14, 1)

6.9 Devise a series of equivalence statements for a collection of sixteen items that yields a tree of height 5 when both the weighted union rule and path compression are used. What is the total number of parent pointers followed to perform this series?

6.10 One alternative to path compression that gives similar performance gains is called **path halving**. In path halving, when the path is traversed from the node to the root, we make the grandparent of every other node i on the path the new parent of i. Write a version of **`FIND`** that implements path halving. Your **`FIND`** operation should work as you move up the tree, rather than require the two passes needed by path compression.

6.11 Analyze the fraction of overhead required by the "list of children" implementation, the "left-child/right-sibling" implementation, and the two linked implementations of Section 6.3.3. How do these implementations compare in space efficiency?

6.12 Using the general tree ADT of Figure 6.2, write a function that takes as input the root of a general tree and returns a binary tree generated by the conversion process illustrated by Figure 6.14.

6.13 Use mathematical induction to prove that the number of leaves in a non-empty full K-ary tree is $(K-1)n+1$, where n is the number of internal nodes.

6.14 Derive the formulas for computing the relatives of a non-empty complete K-ary tree node stored in the complete tree representation of Section 5.3.3.

6.15 Find the overhead fraction for a full K-ary tree implementation with space requirements as follows:

(a) All nodes store data, K child pointers, and a parent pointer. The data field requires four bytes and each pointer requires four bytes.

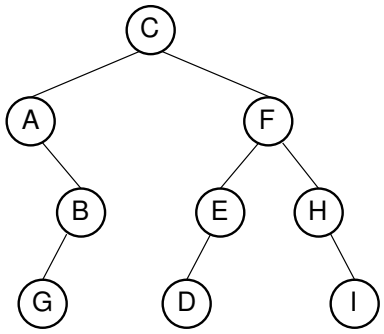


Figure 6.18 A sample tree for Exercise 6.16.

(b) All nodes store data and K child pointers. The data field requires sixteen bytes and each pointer requires four bytes.

(c) All nodes store data and a parent pointer, and internal nodes store K child pointers. The data field requires eight bytes and each pointer requires four bytes.

(d) Only leaf nodes store data; only internal nodes store K child pointers. The data field requires four bytes and each pointer requires two bytes.

6.16 **(a)** Write out the sequential representation for Figure 6.18 using the coding illustrated by Example 6.5.

(b) Write out the sequential representation for Figure 6.18 using the coding illustrated by Example 6.6.

6.17 Draw the binary tree representing the following sequential representation for binary trees illustrated by Example 6.5:

$$\mathrm{ABD//E//C/F//}$$

6.18 Draw the binary tree representing the following sequential representation for binary trees illustrated by Example 6.6:

$$\mathrm{A'/B'/C'D'G/E}$$

Show the bit vector for leaf and internal nodes (as illustrated by Example 6.7) for this tree.

6.19 Draw the general tree represented by the following sequential representation for general trees illustrated by Example 6.8:

$$\mathrm{XPC)Q)RV)M))))}$$

6.20 **(a)** Write a function to decode the sequential representation for binary trees illustrated by Example 6.5. The input should be the sequential representation and the output should be a pointer to the root of the resulting binary tree.

(b) Write a function to decode the sequential representation for full binary trees illustrated by Example 6.6. The input should be the sequential representation and the output should be a pointer to the root of the resulting binary tree.

(c) Write a function to decode the sequential representation for general trees illustrated by Example 6.8. The input should be the sequential representation and the output should be a pointer to the root of the resulting general tree.

6.21 Devise a sequential representation for Huffman coding trees suitable for use as part of a file compression utility (see Project 5.7).

6.8 Projects

6.1 Write classes that implement the general tree class declarations of Figure 6.2 using the dynamic "left-child/right-sibling" representation described in Section 6.3.4.

6.2 Write classes that implement the general tree class declarations of Figure 6.2 using the linked general tree implementation with child pointer arrays of Figure 6.12. Your implementation should support only fixed-size nodes that do not change their number of children once they are created. Then, reimplement these classes with the linked list of children representation of Figure 6.13. How do the two implementations compare in space and time efficiency and ease of implementation?

6.3 Write classes that implement the general tree class declarations of Figure 6.2 using the linked general tree implementation with child pointer arrays of Figure 6.12. Your implementation must be able to support changes in the number of children for a node. When created, a node should be allocated with only enough space to store its initial set of children. Whenever a new child is added to a node such that the array overflows, allocate a new array from free store that can store twice as many children.

6.4 Implement a BST file archiver. Your program should take a BST created in main memory using the implementation of Figure 5.14 and write it out to disk using one of the sequential representations of Section 6.5. It should also be able to read in disk files using your sequential representation and create the equivalent main memory representation.

6.5 Use the UNION/FIND algorithm to implement a solution to the following problem. Given a set of points represented by their xy-coordinates, assign the points to clusters. Any two points are defined to be in the same cluster if they are within a specified distance d of each other. For the purpose of this problem, clustering is an equivalence relationship. In other words, points *A*, *B*, and *C* are defined to be in the same cluster if the distance between *A* and *B*

is less than d and the distance between A and C is also less than d, even if the distance between B and C is greater than d. To solve the problem, compute the distance between each pair of points, using the equivalence processing algorithm to merge clusters whenever two points are within the specified distance. What is the asymptotic complexity of this algorithm? Where is the bottleneck in processing?

6.6 In this project, you will run some empirical tests to determine if some variations on path compression in the UNION/FIND algorithm will lead to improved performance. You should compare the following five implementations:

(a) Standard UNION/FIND with path compression and weighted union.

(b) Path compression and weighted union, except that path compression is done *after* the UNION, instead of during the FIND operation. That is, make all nodes along the paths traversed in both trees point directly to the root of the larger tree.

(c) Weighted union and path halving as described in Exercise 6.10.

(d) Weighted union and a simplified form of path compression. At the end of every FIND operation, make the node point to its tree's root (but don't change the pointers for other nodes along the path).

(e) Weighted union and a simplified form of path compression. Both nodes in the equivalence will be set to point directly to the root of the larger tree after the UNION operation. For example, consider processing the equivalence (A, B) where A' is the root of A and B' is the root of B. Assume the tree with root A' is bigger than the tree with root B'. At the end of the UNION/FIND operation, nodes A, B, and B' will all point directly to A'.