

Algorithms

THIRD EDITION

IN C

Parts I-4

FUNDAMENTALS
DATA STRUCTURES
SORTING
SEARCHING

ROBERT SEDGEWICK

Algorithms

THIRD EDITION

in C

PARTS 1–4

FUNDAMENTALS
DATA STRUCTURES
SORTING
SEARCHING

Robert Sedgewick

Princeton University



ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal

London • Munich • Paris • Madrid

Capetown • Sydney • Tokyo • Singapore • Mexico City

Publishing Partner: Peter S. Gordon

Associate Editor: Deborah Lafferty

Cover Designer: Andre Kuzniarek

Production Editor: Amy Willcutt

Copy Editor: Lyn Dupre

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher neither offers any warranties or representations, nor accepts any liabilities with respect to the programs or applications.

Library of Congress Cataloging-in-Publication Data

Sedgewick, Robert, 1946 -

Algorithms in C / Robert Sedgewick. — 3d ed.

720 p. 24 cm.

Includes bibliographical references and index.

Contents: v. 1, pts. 1-4. Fundamentals, data structures,
sorting, searching.

ISBN 0-201-31452-5

1. C (Computer program language) 2. Computer algorithms.

I. Title.

QA76.73.C15S43 1998

005.13'3—dc21

97-23418

CIP

Reproduced by Addison-Wesley from camera-ready copy supplied by
the author.

Copyright © 1998 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced,
stored in a retrieval system, or transmitted, in any form or by any
means, electronic, mechanical, photocopying, recording, or otherwise,
without the prior written permission of the publisher. Printed in the
United States of America.

Text printed on recycled and acid-free paper.

ISBN 0201314525

14 1516171819 CRS 08 07 06

14th Printing January 2006

Preface

THIS BOOK IS intended to survey the most important computer algorithms in use today, and to teach fundamental techniques to the growing number of people in need of knowing them. It can be used as a textbook for a second, third, or fourth course in computer science, after students have acquired basic programming skills and familiarity with computer systems, but before they have taken specialized courses in advanced areas of computer science or computer applications. The book also may be useful for self-study or as a reference for people engaged in the development of computer systems or applications programs, since it contains implementations of useful algorithms and detailed information on these algorithms' performance characteristics. The broad perspective taken makes the book an appropriate introduction to the field.

I have completely rewritten the text for this new edition, and I have added more than a thousand new exercises, more than a hundred new figures, and dozens of new programs. I have also added detailed commentary on all the figures and programs. This new material provides both coverage of new topics and fuller explanations of many of the classic algorithms. A new emphasis on abstract data types throughout the book makes the programs more broadly useful and relevant in modern object-oriented programming environments. People who have read old editions of the book will find a wealth of new information throughout; all readers will find a wealth of pedagogical material that provides effective access to essential concepts.

Due to the large amount of new material, we have split the new edition into two volumes (each about the size of the old edition) of which this is the first. This volume covers fundamental concepts, data structures, sorting algorithms, and searching algorithms; the second volume covers advanced algorithms and applications, building on the basic abstractions and methods developed here. Nearly all the material on fundamentals and data structures in this edition is new.

This book is not just for programmers and computer-science students. Nearly everyone who uses a computer wants it to run faster or to solve larger problems. The algorithms in this book represent a body of knowledge developed over the last 50 years that has become indispensable in the efficient use of the computer, for a broad variety of applications. From N -body simulation problems in physics to genetic-sequencing problems in molecular biology, the basic methods described here have become essential in scientific research; and from database systems to Internet search engines, they have become essential parts of modern software systems. As the scope of computer applications becomes more widespread, so grows the impact of many of the basic methods covered here. The goal of this book is to serve as a resource for students and professionals interested in knowing and making intelligent use of these fundamental algorithms as basic tools for whatever computer application they might undertake.

Scope

The book contains 16 chapters grouped into four major parts: fundamentals, data structures, sorting, and searching. The descriptions here are intended to give readers an understanding of the basic properties of as broad a range of fundamental algorithms as possible. Ingenious methods ranging from binomial queues to patricia tries are described, all related to basic paradigms at the heart of computer science. The second volume consists of four additional parts that cover strings, geometry, graphs, and advanced topics. My primary goal in developing these books has been to bring together the fundamental methods from these diverse areas, to provide access to the best methods known for solving problems by computer.

You will most appreciate the material in this book if you have had one or two previous courses in computer science or have had equivalent programming experience: one course in programming in a high-level language such as C, Java, or C++, and perhaps another course that teaches fundamental concepts of programming systems. This book is thus intended for anyone conversant with a modern programming language and with the basic features of modern computer systems. References that might help to fill in gaps in your background are suggested in the text.

Most of the mathematical material supporting the analytic results is self-contained (or is labeled as beyond the scope of this book), so little specific preparation in mathematics is required for the bulk of the book, although mathematical maturity is definitely helpful.

Use in the Curriculum

There is a great deal of flexibility in how the material here can be taught, depending on the taste of the instructor and the preparation of the students. The algorithms described here have found widespread use for years, and represent an essential body of knowledge for both the practicing programmer and the computer-science student. There is sufficient coverage of basic material for the book to be used for a course on data structures, and there is sufficient detail and coverage of advanced material for the book to be used for a course on algorithms. Some instructors may wish to emphasize implementations and practical concerns; others may wish to emphasize analysis and theoretical concepts.

A complete set of slide masters for use in lectures, sample programming assignments, interactive exercises for students, and other course materials may be found via the book's home page.

An elementary course on data structures and algorithms might emphasize the basic data structures in Part 2 and their use in the implementations in Parts 3 and 4. A course on design and analysis of algorithms might emphasize the fundamental material in Part 1 and Chapter 5, then study the ways in which the algorithms in Parts 3 and 4 achieve good asymptotic performance. A course on software engineering might omit the mathematical and advanced algorithmic material, and emphasize how to integrate the implementations given here into large programs or systems. A course on algorithms might take a survey approach and introduce concepts from all these areas.

Earlier editions of this book have been used in recent years at scores of colleges and universities around the world as a text for the second or third course in computer science and as supplemental reading for other courses. At Princeton, our experience has been that the breadth of coverage of material in this book provides our majors with an introduction to computer science that can be expanded upon in later courses on analysis of algorithms, systems programming and

theoretical computer science, while providing the growing group of students from other disciplines with a large set of techniques that these people can immediately put to good use.

The exercises—most of which are new to this edition—fall into several types. Some are intended to test understanding of material in the text, and simply ask readers to work through an example or to apply concepts described in the text. Others involve implementing and putting together the algorithms, or running empirical studies to compare variants of the algorithms and to learn their properties. Still others are a repository for important information at a level of detail that is not appropriate for the text. Reading and thinking about the exercises will pay dividends for every reader.

Algorithms of Practical Use

Anyone wanting to use a computer more effectively can use this book for reference or for self-study. People with programming experience can find information on specific topics throughout the book. To a large extent, you can read the individual chapters in the book independently of the others, although, in some cases, algorithms in one chapter make use of methods from a previous chapter.

The orientation of the book is to study algorithms likely to be of practical use. The book provides information about the tools of the trade to the point that readers can confidently implement, debug, and put to work algorithms to solve a problem or to provide functionality in an application. Full implementations of the methods discussed are included, as are descriptions of the operations of these programs on a consistent set of examples. Because we work with real code, rather than write pseudo-code, the programs can be put to practical use quickly. Program listings are available from the book's home page.

Indeed, one practical application of the algorithms has been to produce the hundreds of figures throughout the book. Many algorithms are brought to light on an intuitive level through the visual dimension provided by these figures.

Characteristics of the algorithms and of the situations in which they might be useful are discussed in detail. Although not emphasized, connections to the analysis of algorithms and theoretical computer science are developed in context. When appropriate, empirical and

analytic results are presented to illustrate why certain algorithms are preferred. When interesting, the relationship of the practical algorithms being discussed to purely theoretical results is described. Specific information on performance characteristics of algorithms and implementations is synthesized, encapsulated, and discussed throughout the book.

Programming Language

The programming language used for all of the implementations is C. Any particular language has advantages and disadvantages; we use C because it is widely available and provides the features needed for our implementations. The programs can be translated easily to other modern programming languages, since relatively few constructs are unique to C. We use standard C idioms when appropriate, but this book is not intended to be a reference work on C programming.

There are many new programs in this edition, and many of the old ones have been reworked, primarily to make them more readily useful as abstract-data-type implementations. Extensive comparative empirical tests on the programs are discussed throughout the text.

Previous editions of the book have presented basic programs in Pascal, C++, and Modula-3. This code is available through the book home page on the web; code for new programs and code in new languages such as Java will be added as appropriate.

A goal of this book is to present the algorithms in as simple and direct a form as possible. The style is consistent whenever possible, so that programs that are similar look similar. For many of the algorithms in this book, the similarities hold regardless of the language: Quicksort is quicksort (to pick one prominent example), whether expressed in Algol-60, Basic, Fortran, Smalltalk, Ada, Pascal, C, PostScript, Java, or countless other programming languages and environments where it has proved to be an effective sorting method.

We strive for elegant, compact, and portable implementations, but we take the point of view that efficiency matters, so we try to be aware of the performance characteristics of our code at all stages of development. Chapter 1 constitutes a detailed example of this approach to developing efficient C implementations of our algorithms, and sets the stage for the rest of the book.

Acknowledgments

Many people gave me helpful feedback on earlier versions of this book. In particular, hundreds of students at Princeton and Brown have suffered through preliminary drafts over the years. Special thanks are due to Trina Avery and Tom Freeman for their help in producing the first edition; to Janet Incerpi for her creativity and ingenuity in persuading our early and primitive digital computerized typesetting hardware and software to produce the first edition; to Marc Brown for his part in the algorithm visualization research that was the genesis of so many of the figures in the book; and to Dave Hanson for his willingness to answer all of my questions about C. I would also like to thank the many readers who have provided me with detailed comments about various editions, including Guy Almes, Jon Bentley, Marc Brown, Jay Gischer, Allan Heydon, Kennedy Lemke, Udi Manber, Dana Richards, John Reif, M. Rosenfeld, Stephen Seidman, Michael Quinn, and William Ward.

To produce this new edition, I have had the pleasure of working with Peter Gordon and Debbie Lafferty at Addison-Wesley, who have patiently shepherded this project as it has evolved from a standard update to a massive rewrite. It has also been my pleasure to work with several other members of the professional staff at Addison-Wesley. The nature of this project made the book a somewhat unusual challenge for many of them, and I much appreciate their forbearance.

I have gained two new mentors in writing this book, and particularly want to express my appreciation to them. First, Steve Summit carefully checked early versions of the manuscript on a technical level, and provided me with literally thousands of detailed comments, particularly on the programs. Steve clearly understood my goal of providing elegant, efficient, and effective implementations, and his comments not only helped me to provide a measure of consistency across the implementations, but also helped me to improve many of them substantially. Second, Lyn Dupre also provided me with thousands of detailed comments on the manuscript, which were invaluable in helping me not only to correct and avoid grammatical errors, but also—more important—to find a consistent and coherent writing style that helps bind together the daunting mass of technical material here. I am extremely grateful

for the opportunity to learn from Steve and Lyn—their input was vital in the development of this book.

Much of what I have written here I have learned from the teaching and writings of Don Knuth, my advisor at Stanford. Although Don had no direct influence on this work, his presence may be felt in the book, for it was he who put the study of algorithms on the scientific footing that makes a work such as this possible. My friend and colleague Philippe Flajolet, who has been a major force in the development of the analysis of algorithms as a mature research area, has had a similar influence on this work.

I am deeply thankful for the support of Princeton University, Brown University, and the Institut National de Recherche en Informatique et Automatique (INRIA), where I did most of the work on the book; and of the Institute for Defense Analyses and the Xerox Palo Alto Research Center, where I did some work on the book while visiting. Many parts of the book are dependent on research that has been generously supported by the National Science Foundation and the Office of Naval Research. Finally, I thank Bill Bowen, Aaron Lemonick, and Neil Rudenstine for their support in building an academic environment at Princeton in which I was able to prepare this book, despite my numerous other responsibilities.

*Robert Sedgewick
Marly-le-Roi, France, February, 1983
Princeton, New Jersey, January, 1990
Jamestown, Rhode Island, August, 1997*

*To Adam, Andrew, Brett, Robbie,
and especially Linda*

Notes on Exercises

Classifying exercises is an activity fraught with peril, because readers of a book such as this come to the material with various levels of knowledge and experience. Nonetheless, guidance is appropriate, so many of the exercises carry one of four annotations, to help you decide how to approach them.

Exercises that *test your understanding* of the material are marked with an open triangle, as follows:

- ▷ 9.57 Give the binomial queue that results when the keys E A S Y
Q U E S T I O N are inserted into an initially empty binomial queue.

Most often, such exercises relate directly to examples in the text. They should present no special difficulty, but working them might teach you a fact or concept that may have eluded you when you read the text.

Exercises that *add new and thought-provoking* information to the material are marked with an open circle, as follows:

- 14.20 Write a program that inserts N random integers into a table of size $N/100$ using separate chaining, then finds the length of the shortest and longest lists, for $N = 10^3, 10^4, 10^5$, and 10^6 .

Such exercises encourage you to think about an important concept that is related to the material in the text, or to answer a question that may have occurred to you when you read the text. You may find it worthwhile to read these exercises, even if you do not have the time to work them through.

Exercises that are intended to *challenge you* are marked with a black dot, as follows:

- 8.46 Suppose that mergesort is implemented to split the file at a *random* position, rather than exactly in the middle. How many comparisons are used by such a method to sort N elements, on the average?

Such exercises may require a substantial amount of time to complete, depending upon your experience. Generally, the most productive approach is to work on them in a few different sittings.

A few exercises that are *extremely difficult* (by comparison with most others) are marked with two black dots, as follows:

- 15.29 Prove that the height of a trie built from N random bit-strings is about $2 \lg N$.

These exercises are similar to questions that might be addressed in the research literature, but the material in the book may prepare you to enjoy trying to solve them (and perhaps succeeding).

The annotations are intended to be neutral with respect to your programming and mathematical ability. Those exercises that require expertise in programming or in mathematical analysis are self-evident. All readers are encouraged to test their understanding of the algorithms by implementing them. Still, an exercise such as this one is straightforward for a practicing programmer or a student in a programming course, but may require substantial work for someone who has not recently programmed:

- 1.23** Modify Program 1.4 to generate random pairs of integers between 0 and $N - 1$ instead of reading them from standard input, and to loop until $N - 1$ *union* operations have been performed. Run your program for $N = 10^3, 10^4, 10^5$, and 10^6 and print out the total number of edges generated for each value of N .

In a similar vein, all readers are encouraged to strive to appreciate the analytic underpinnings of our knowledge about properties of algorithms. Still, an exercise such as this one is straightforward for a scientist or a student in a discrete mathematics course, but may require substantial work for someone who has not recently done mathematical analysis:

- 1.13** Compute the *average* distance from a node to the root in a worst-case tree of 2^n nodes built by the weighted quick-union algorithm.

There are far too many exercises for you to read and assimilate them all; my hope is that there are enough exercises here to stimulate you to strive to come to a broader understanding on the topics that interest you than you can glean by simply reading the text.

Contents

Fundamentals

Chapter 1. Introduction	3
1.1 Algorithms · 4	
1.2 A Sample Problem—Connectivity · 6	
1.3 Union-Find Algorithms · 11	
1.4 Perspective · 22	
1.5 Summary of Topics · 23	
Chapter 2. Principles of Algorithm Analysis	27
2.1 Implementation and Empirical Analysis · 28	
2.2 Analysis of Algorithms · 33	
2.3 Growth of Functions · 36	
2.4 Big-Oh notation · 44	
2.5 Basic Recurrences · 49	
2.6 Examples of Algorithm Analysis · 53	
2.7 Guarantees, Predictions, and Limitations · 59	

Data Structures

Chapter 3. Elementary Data Structures	69
3.1 Building Blocks · 70	
3.2 Arrays · 82	
3.3 Linked Lists · 90	
3.4 Elementary List Processing · 96	
3.5 Memory Allocation for Lists · 105	
3.6 Strings · 109	
3.7 Compound Data Structures · 115	
Chapter 4. Abstract Data Types	127
4.1 Abstract Objects and Collections of Objects · 131	
4.2 Pushdown Stack ADT · 135	
4.3 Examples of Stack ADT Clients · 138	
4.4 Stack ADT Implementations · 144	
4.5 Creation of a New ADT · 149	
4.6 FIFO Queues and Generalized Queues · 153	
4.7 Duplicate and Index Items · 161	
4.8 First-Class ADTs · 165	
4.9 Application-Based ADT Example · 178	
4.10 Perspective · 184	
Chapter 5. Recursion and Trees	187
5.1 Recursive Algorithms · 188	
5.2 Divide and Conquer · 196	
5.3 Dynamic Programming · 208	
5.4 Trees · 216	
5.5 Mathematical Properties of Trees · 226	
5.6 Tree Traversal · 230	
5.7 Recursive Binary-Tree Algorithms · 235	
5.8 Graph Traversal · 241	
5.9 Perspective · 247	

Sorting

Chapter 6. Elementary Sorting Methods 253

- 6.1 Rules of the Game · 255
- 6.2 Selection Sort · 261
- 6.3 Insertion Sort · 262
- 6.4 Bubble Sort · 265
- 6.5 Performance Characteristics of Elementary Sorts · 267
- 6.6 Shellsort · 273
- 6.7 Sorting Other Types of Data · 281
- 6.8 Index and Pointer Sorting · 287
- 6.9 Sorting of Linked Lists · 294
- 6.10 Key-Indexed Counting · 298

Chapter 7. Quicksort 303

- 7.1 The Basic Algorithm · 304
- 7.2 Performance Characteristics of Quicksort · 309
- 7.3 Stack Size · 313
- 7.4 Small Subfiles · 316
- 7.5 Median-of-Three Partitioning · 319
- 7.6 Duplicate Keys · 324
- 7.7 Strings and Vectors · 327
- 7.8 Selection · 329

Chapter 8. Merging and Mergesort 335

- 8.1 Two-Way Merging · 336
- 8.2 Abstract In-place Merge · 339
- 8.3 Top-Down Mergesort · 341
- 8.4 Improvements to the Basic Algorithm · 344
- 8.5 Bottom-Up Mergesort · 347
- 8.6 Performance Characteristics of Mergesort · 351
- 8.7 Linked-List Implementations of Mergesort · 354
- 8.8 Recursion Revisited · 357

Chapter 9. Priority Queues and Heapsort 361

- 9.1 Elementary Implementations · 365
- 9.2 Heap Data Structure · 368

9.3 Algorithms on Heaps · 371	
9.4 Heapsort · 376	
9.5 Priority-Queue ADT · 383	
9.6 Priority Queues for Index Items · 389	
9.7 Binomial Queues · 392	
Chapter 10. Radix Sorting	403
10.1 Bits, Bytes, and Words · 405	
10.2 Binary Quicksort · 409	
10.3 MSD Radix Sort · 413	
10.4 Three-Way Radix Quicksort · 421	
10.5 LSD Radix Sort · 425	
10.6 Performance Characteristics of Radix Sorts · 428	
10.7 Sublinear-Time Sorts · 433	
Chapter 11. Special-Purpose Sorts	439
11.1 Batcher's Odd-Even Mergesort · 441	
11.2 Sorting Networks · 446	
11.3 External Sorting · 454	
11.4 Sort-Merge Implementations · 460	
11.5 Parallel Sort/Merge · 466	

Searching

Chapter 12. Symbol Tables and BSTs	477
12.1 Symbol-Table Abstract Data Type · 479	
12.2 Key-Indexed Search · 485	
12.3 Sequential Search · 489	
12.4 Binary Search · 497	
12.5 Binary Search Trees (BSTs) · 502	
12.6 Performance Characteristics of BSTs · 508	
12.7 Index Implementations with Symbol Tables · 511	
12.8 Insertion at the Root in BSTs · 516	
12.9 BST Implementations of Other ADT Functions · 519	

Chapter 13. Balanced Trees	529
13.1 Randomized BSTs · 533	
13.2 Splay BSTs · 540	
13.3 Top-Down 2-3-4 Trees · 546	
13.4 Red-Black Trees · 551	
13.5 Skip Lists · 561	
13.6 Performance Characteristics · 569	
Chapter 14. Hashing	573
14.1 Hash Functions · 574	
14.2 Separate Chaining · 583	
14.3 Linear Probing · 588	
14.4 Double Hashing · 594	
14.5 Dynamic Hash Tables · 599	
14.6 Perspective · 603	
Chapter 15. Radix Search	609
15.1 Digital Search Trees · 610	
15.2 Tries · 614	
15.3 Patricia Tries · 623	
15.4 Multiway Tries and TSTs · 632	
15.5 Text String Index Algorithms · 648	
Chapter 16. External Searching	655
16.1 Rules of the Game · 657	
16.2 Indexed Sequential Access · 660	
16.3 B Trees · 662	
16.4 Extendible Hashing · 676	
16.5 Perspective · 688	
Index	693

P A R T
O N E

Fundamentals



CHAPTER ONE

Introduction

THE OBJECTIVE OF this book is to study a broad variety of important and useful *algorithms*: methods for solving problems that are suited for computer implementation. We shall deal with many different areas of application, always concentrating on fundamental algorithms that are important to know and interesting to study. We shall spend enough time on each algorithm to understand its essential characteristics and to respect its subtleties. Our goal is to learn a large number of the most important algorithms used on computers today, well enough to be able to use and appreciate them.

The strategy that we use for understanding the programs presented in this book is to implement and test them, to experiment with their variants, to discuss their operation on small examples, and to try them out on larger examples similar to what we might encounter in practice. We shall use the C programming language to describe the algorithms, thus providing useful implementations at the same time. Our programs have a uniform style that is amenable to translation into other modern programming languages, as well.

We also pay careful attention to performance characteristics of our algorithms, to help us develop improved versions, compare different algorithms for the same task, and predict or guarantee performance for large problems. Understanding how the algorithms perform might require experimentation or mathematical analysis or both. We consider detailed information for many of the most important algorithms, developing analytic results directly when feasible, or calling on results from the research literature when necessary.

To illustrate our general approach to developing algorithmic solutions, we consider in this chapter a detailed example comprising a number of algorithms that solve a particular problem. The problem that we consider is not a toy problem; it is a fundamental computational task, and the solution that we develop is of use in a variety of applications. We start with a simple solution, then seek to understand that solution's performance characteristics, which help us to see how to improve the algorithm. After a few iterations of this process, we come to an efficient and useful algorithm for solving the problem. This prototypical example sets the stage for our use of the same general methodology throughout the book.

We conclude the chapter with a short discussion of the contents of the book, including brief descriptions of what the major parts of the book are and how they relate to one another.

1.1 Algorithms

When we write a computer program, we are generally implementing a method that has been devised previously to solve some problem. This method is often independent of the particular computer to be used—it is likely to be equally appropriate for many computers and many computer languages. It is the method, rather than the computer program itself, that we must study to learn how the problem is being attacked. The term *algorithm* is used in computer science to describe a problem-solving method suitable for implementation as a computer program. Algorithms are the stuff of computer science: They are central objects of study in many, if not most, areas of the field.

Most algorithms of interest involve methods of organizing the data involved in the computation. Objects created in this way are called *data structures*, and they also are central objects of study in computer science. Thus, algorithms and data structures go hand in hand. In this book we take the view that data structures exist as the byproducts or end products of algorithms, and thus that we must study them in order to understand the algorithms. Simple algorithms can give rise to complicated data structures and, conversely, complicated algorithms can use simple data structures. We shall study the properties of many data structures in this book; indeed, the book might well have been called *Algorithms and Data Structures in C*.

When we use a computer to help us solve a problem, we typically are faced with a number of possible different approaches. For small problems, it hardly matters which approach we use, as long as we have one that solves the problem correctly. For huge problems (or applications where we need to solve huge numbers of small problems), however, we quickly become motivated to devise methods that use time or space as efficiently as possible.

The primary reason for us to learn about algorithm design is that this discipline gives us the potential to reap huge savings, even to the point of making it possible to do tasks that would otherwise be impossible. In an application where we are processing millions of objects, it is not unusual to be able to make a program millions of times faster by using a well-designed algorithm. We shall see such an example in Section 1.2 and on numerous other occasions throughout the book. By contrast, investing additional money or time to buy and install a new computer holds the potential for speeding up a program by perhaps a factor of only 10 or 100. Careful algorithm design is an extremely effective part of the process of solving a huge problem, whatever the applications area.

When a huge or complex computer program is to be developed, a great deal of effort must go into understanding and defining the problem to be solved, managing its complexity, and decomposing it into smaller subtasks that can be implemented easily. Often, many of the algorithms required after the decomposition are trivial to implement. In most cases, however, there are a few algorithms whose choice is critical because most of the system resources will be spent running those algorithms. Those are the types of algorithms on which we concentrate in this book. We shall study a variety of fundamental algorithms that are useful for solving huge problems in a broad variety of applications areas.

The sharing of programs in computer systems is becoming more widespread, so, although we might expect to be *using* a large fraction of the algorithms in this book, we also might expect to have to *implement* only a smaller fraction of them. However, implementing simple versions of basic algorithms helps us to understand them better and thus to use advanced versions more effectively. More important, the opportunity to reimplement basic algorithms arises frequently. The primary reason to do so is that we are faced, all too often, with com-

pletely new computing environments (hardware and software) with new features that old implementations may not use to best advantage. In other words, we often implement basic algorithms tailored to our problem, rather than depending on a system routine, to make our solutions more portable and longer lasting. Another common reason to reimplement basic algorithms is that mechanisms for sharing software on many computer systems are not always sufficiently powerful to allow us to tailor standard programs to perform effectively on specific tasks (or it may not be convenient to do so), so it is sometimes easier to do a new implementation.

Computer programs are often overoptimized. It may not be worthwhile to take pains to ensure that an implementation of a particular algorithm is the most efficient possible unless the algorithm is to be used for an enormous task or is to be used many times. Otherwise, a careful, relatively simple implementation will suffice: We can have some confidence that it will work, and it is likely to run perhaps five or 10 times slower at worst than the best possible version, which means that it may run for an extra few seconds. By contrast, the proper choice of algorithm in the first place can make a difference of a factor of 100 or 1000 or more, which might translate to minutes, hours, or even more in running time. In this book, we concentrate on the simplest reasonable implementations of the best algorithms.

The choice of the best algorithm for a particular task can be a complicated process, perhaps involving sophisticated mathematical analysis. The branch of computer science that comprises the study of such questions is called *analysis of algorithms*. Many of the algorithms that we study have been shown through analysis to have excellent performance; others are simply known to work well through experience. Our primary goal is to learn reasonable algorithms for important tasks, yet we shall also pay careful attention to comparative performance of the methods. We should not use an algorithm without having an idea of what resources it might consume, and we strive to be aware of how our algorithms might be expected to perform.

1.2 A Sample Problem: Connectivity

Suppose that we are given a sequence of pairs of integers, where each integer represents an object of some type and we are to interpret the

pair $p-q$ as meaning “ p is connected to q .” We assume the relation “is connected to” to be transitive: If p is connected to q , and q is connected to r , then p is connected to r . Our goal is to write a program to filter out extraneous pairs from the set: When the program inputs a pair $p-q$, it should output the pair only if the pairs it has seen to that point do *not* imply that p is connected to q . If the previous pairs do imply that p is connected to q , then the program should ignore $p-q$ and should proceed to input the next pair. Figure 1.1 gives an example of this process.

Our problem is to devise a program that can remember sufficient information about the pairs it has seen to be able to decide whether or not a new pair of objects is connected. Informally, we refer to the task of designing such a method as the *connectivity problem*. This problem arises in a number of important applications. We briefly consider three examples here to indicate the fundamental nature of the problem.

For example, the integers might represent computers in a large network, and the pairs might represent connections in the network. Then, our program might be used to determine whether we need to establish a new direct connection for p and q to be able to communicate, or whether we could use existing connections to set up a communications path. In this kind of application, we might **need** to process millions of points and billions of connections, or more. As we shall see, it would be impossible to solve the problem for such an application without an efficient algorithm.

Similarly, the integers might represent contact points in an electrical network, and the pairs might represent wires connecting the points. In this case, we could use our program to find a way to connect all the points without any extraneous connections, if that is possible. There is no guarantee that the edges in the list will suffice to connect all the points—indeed, we shall soon see that determining whether or not they will could be a prime application of our program.

Figure 1.2 illustrates these two types of applications in a larger example. Examination of this figure gives us an appreciation for the difficulty of the connectivity problem: How can we arrange to tell quickly whether *any* given two points in such a network are connected?

Still another example arises in certain programming environments where it is possible to declare two variable names as equivalent. The problem is to be able to determine whether two given names are

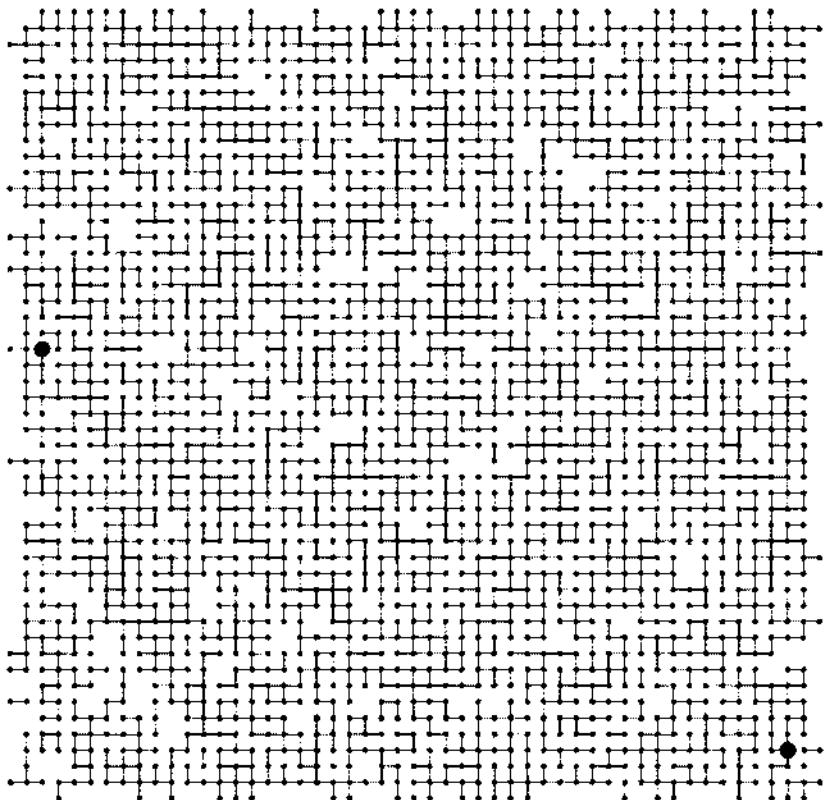
3-4	3-4	
4-9	4-9	
8-0	8-0	
2-3	2-3	
5-6	5-6	
2-9		2-3-4-9
5-9	5-9	
7-3	7-3	
4-8	4-8	
5-6		5-6
0-2		0-8-4-3-2
6-1	6-1	

Figure 1.1
Connectivity example

Given a sequence of pairs of integers representing connections between objects (left), the task of a connectivity algorithm is to output those pairs that provide new connections (center). For example, the pair 2-9 is not part of the output because the connection 2-3-4-9 is implied by previous connections (this evidence is shown at right).

Figure 1.2**A large connectivity example**

The objects in a connectivity problem might represent connection points, and the pairs might be connections between them, as indicated in this idealized example that might represent wires connecting buildings in a city or components on a computer chip. This graphical representation makes it possible for a human to spot nodes that are not connected, but the algorithm has to work with only the pairs of integers that it is given. Are the two nodes marked with the large black dots connected?



equivalent, after a sequence of such declarations. This application is an early one that motivated the development of several of the algorithms that we are about to consider. It directly relates our problem to a simple abstraction that provides us with a way to make our algorithms useful for a wide variety of applications, as we shall see.

Applications such as the variable-name-equivalence problem described in the previous paragraph require that we associate an integer with each distinct variable name. This association is also implicit in the network-connection and circuit-connection applications that we have described. We shall be considering a host of algorithms in Chapters 10 through 16 that can provide this association in an efficient manner. Thus, we can assume in this chapter, without loss of generality, that we have N objects with integer names, from 0 to $N - 1$.

We are asking for a program that does a specific and well-defined task. There are many other related problems that we might want to have solved, as well. One of the first tasks that we face in developing an algorithm is to be sure that we have specified the *problem* in a reasonable manner. The more we require of an algorithm, the more time and space we may expect it to need to finish the task. It is impossible to quantify this relationship *a priori*, and we often modify a problem specification on finding that it is difficult or expensive to solve, or, in happy circumstances, on finding that an algorithm can provide information more useful than was called for in the original specification.

For example, our connectivity-problem specification requires only that our program somehow know whether or not any given pair $p-q$ is connected, and not that it be able to demonstrate any or all ways to connect that pair. Adding a requirement for such a specification makes the problem more difficult, and would lead us to a different family of algorithms, which we consider briefly in Chapter 5 and in detail in Part 7.

The specifications mentioned in the previous paragraph ask us for *more* information than our original one did; we could also ask for *less* information. For example, we might simply want to be able to answer the question: “Are the M connections sufficient to connect together all N objects?” This problem illustrates that, to develop efficient algorithms, we often need to do high-level reasoning about the abstract objects that we are processing. In this case, a fundamental result from graph theory implies that all N objects are connected if and only if the number of pairs output by the connectivity algorithm is precisely $N - 1$ (see Section 5.4). In other words, a connectivity algorithm will never output more than $N - 1$ pairs, because, once it has output $N - 1$ pairs, any pair that it encounters from that point on will be connected. Accordingly, we can get a program that answers the yes–no question just posed by changing a program that solves the connectivity problem to one that increments a counter, rather than writing out each pair that was not previously connected, answering “yes” when the counter reaches $N - 1$ and “no” if it never does. This question is but one example of a host of questions that we might wish to answer regarding connectivity. The set of pairs in the input is called a *graph*, and the set of pairs output is called a *spanning tree* for

that graph, which connects all the objects. We consider properties of graphs, spanning trees, and all manner of related algorithms in Part 7.

It is worthwhile to try to identify the fundamental operations that we will be performing, and so to make any algorithm that we develop for the connectivity task useful for a variety of similar tasks. Specifically, each time that we get a new pair, we have first to determine whether it represents a new connection, then to incorporate the information that the connection has been seen into its understanding about the connectivity of the objects such that it can check connections to be seen in the future. We encapsulate these two tasks as *abstract operations* by considering the integer input values to represent elements in abstract sets, and then design algorithms and data structures that can

- *Find* the set containing a given item.
- Replace the sets containing two given items by their *union*.

Organizing our algorithms in terms of these abstract operations does not seem to foreclose any options in solving the connectivity problem, and the operations may be useful for solving other problems. Developing ever more powerful layers of abstraction is an essential process in computer science in general and in algorithm design in particular, and we shall turn to it on numerous occasions throughout this book. In this chapter, we use abstract thinking in an informal way to guide us in designing programs to solve the connectivity problem; in Chapter 4, we shall see how to encapsulate abstractions in C code.

The connectivity problem is easily solved in terms of the *find* and *union* abstract operations. After reading a new pair p-q from the input, we perform a *find* operation for each member of the pair. If the members of the pair are in the same set, we move on to the next pair; if they are not, we do a *union* operation and write out the pair. The sets represent *connected components*: subsets of the objects with the property that any two objects in a given component are connected. This approach reduces the development of an algorithmic solution for connectivity to the tasks of defining a data structure representing the sets and developing *union* and *find* algorithms that efficiently use that data structure.

There are many possible ways to represent and process abstract sets, which we consider in more detail in Chapter 4. In this chapter, our focus is on finding a representation that can support efficiently

the *union* and *find* operations that we see in solving the connectivity problem.

Exercises

1.1 Give the output that a connectivity algorithm should produce when given the input 0–2, 1–4, 2–5, 3–6, 0–4, 6–0, and 1–3.

1.2 List all the different ways to connect two different objects for the example in Figure 1.1.

1.3 Describe a simple method for counting the number of sets remaining after using the *union* and *find* operations to solve the connectivity problem as described in the text.

1.3 Union–Find Algorithms

The first step in the process of developing an efficient algorithm to solve a given problem is to *implement a simple algorithm that solves the problem*. If we need to solve a few particular problem instances that turn out to be easy, then the simple implementation may finish the job for us. If a more sophisticated algorithm is called for, then the simple implementation provides us with a correctness check for small cases and a baseline for evaluating performance characteristics. We always care about efficiency, but our primary concern in developing the first program that we write to solve a problem is to make sure that the program is a *correct* solution to the problem.

The first idea that might come to mind is somehow to save all the input pairs, then to write a function to pass through them to try to discover whether the next pair of objects is connected. We shall use a different approach. First, the number of pairs might be sufficiently large to preclude our saving them all in memory in practical applications. Second, and more to the point, no simple method immediately suggests itself for determining whether two objects are connected from the set of all the connections, even if we could save them all! We consider a basic method that takes this approach in Chapter 5, but the methods that we shall consider in this chapter are simpler, because they solve a less difficult problem, and are more efficient, because they do not require saving all the pairs. They all use an array of integers—one corresponding to each object—to hold the requisite information to be able to implement *union* and *find*.

p \ q	0	1	2	3	4	5	6	7	8	9
3 4	0	1	2	4	4	5	6	7	8	9
4 9	0	1	2	9	9	5	6	7	8	9
8 0	0	1	2	9	9	5	6	7	0	9
2 3	0	1	9	9	9	5	6	7	0	9
5 6	0	1	9	9	9	6	6	7	0	9
2 9	0	1	9	9	9	6	6	7	0	9
5 9	0	1	9	9	9	9	9	7	0	9
7 3	0	1	9	9	9	9	9	9	0	9
4 8	0	1	0	0	0	0	0	0	0	0
3 6	0	1	0	0	0	0	0	0	0	0
0 2	0	1	0	0	0	0	0	0	0	0
6 1	1	1	1	1	1	1	1	1	1	1

Figure 1.3
Example of quick find (slow union)

This sequence depicts the contents of the *id* array after each of the pairs at left is processed by the quick-find algorithm (Program 1.1). Shaded entries are those that change for the *union* operation. When we process the pair *p q*, we change all entries with the value *id[p]* to have the value *id[q]*.

Program 1.1 Quick-find solution to connectivity problem

This program reads a sequence of pairs of nonnegative integers less than N from standard input (interpreting the pair p, q to mean “connect object p to object q ”) and prints out pairs representing objects that are not yet connected. It maintains an array id that has an entry for each object, with the property that $\text{id}[p]$ and $\text{id}[q]$ are equal if and only if p and q are connected. For simplicity, we define N as a compile-time constant. Alternatively, we could take it from the input and allocate the id array dynamically (see Section 3.2).

```
#include <stdio.h>
#define N 10000
main()
{ int i, p, q, t, id[N];
  for (i = 0; i < N; i++) id[i] = i;
  while (scanf("%d %d\n", &p, &q) == 2)
  {
    if (id[p] == id[q]) continue;
    for (t = id[p], i = 0; i < N; i++)
      if (id[i] == t) id[i] = id[q];
    printf(" %d %d\n", p, q);
  }
}
```

Arrays are elementary data structures that we shall discuss in detail in Section 3.2. Here, we use them in their simplest form: we declare that we expect to use, say, 1000 integers, by writing $\text{a}[1000]$; then we refer to the i th integer in the array by writing $\text{a}[i]$ for $0 \leq i < 1000$.

Program 1.1 is an implementation of a simple algorithm called the *quick-find algorithm* that solves the connectivity problem. The basis of this algorithm is an array of integers with the property that p and q are connected if and only if the p th and q th array entries are equal. We initialize the i th array entry to i for $0 \leq i < N$. To implement the *union* operation for p and q , we go through the array, changing all the entries with the same name as p to have the same name as q . This choice is arbitrary—we could have decided to change all the entries with the same name as q to have the same name as p .

Figure 1.3 shows the changes to the array for the *union* operations in the example in Figure 1.1. To implement *find*, we just test the indicated array entries for equality—hence the name *quick find*. The *union* operation, on the other hand, involves scanning through the whole array for each input pair.

Property 1.1 *The quick-find algorithm executes at least MN instructions to solve a connectivity problem with N objects that involves M union operations.*

For each of the M *union* operations, we iterate the for loop N times. Each iteration requires at least one instruction (if only to check whether the loop is finished). ■

We can execute tens or hundreds of millions of instructions per second on modern computers, so this cost is not noticeable if M and N are small, but we also might find ourselves with millions of objects and billions of input pairs to process in a modern application. The inescapable conclusion is that we cannot feasibly solve such a problem using the quick-find algorithm (see Exercise 1.10). We consider the process of quantifying such a conclusion precisely in Chapter 2.

Figure 1.4 shows a graphical representation of Figure 1.3. We may think of some of the objects as representing the set to which they belong, and all of the other objects as pointing to the representative in their set. The reason for moving to this graphical representation of the array will become clear soon. Observe that the connections between objects in this representation are *not* necessarily the same as the connections in the input pairs—they are the information that the algorithm chooses to remember to be able to know whether future pairs are connected.

The next algorithm that we consider is a complementary method called the *quick-union algorithm*. It is based on the same data structure—an array indexed by object names—but it uses a different interpretation of the values that leads to more complex abstract structures. Each object points to another object in the same set, in a structure with no cycles. To determine whether two objects are in the same set, we follow pointers for each until we reach an object that points to itself. The objects are in the same set if and only if this process leads them to the same object. If they are not in the same set, we wind up at different objects (which point to themselves). To form

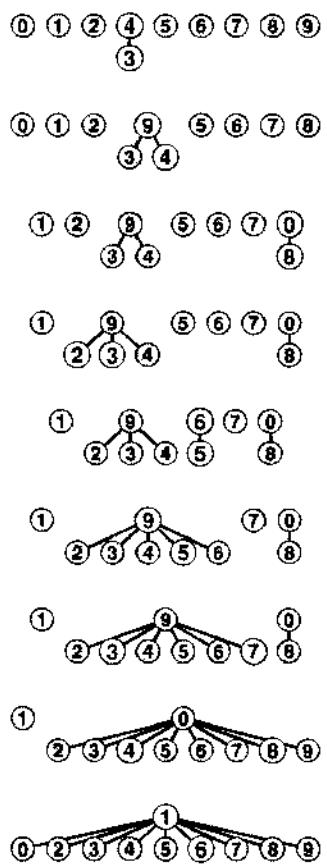


Figure 1.4
Tree representation of quick find

This figure depicts graphical representations for the example in Figure 1.3. The connections in these figures do not necessarily represent the connections in the input. For example, the structure at the bottom has the connection 1-7, which is not in the input, but which is made because of the string of connections 7-3-4-9-5-6-1.

the union, then we just link one to the other to perform the *union* operation; hence the name *quick-union*.

Figure 1.5 shows the graphical representation that corresponds to Figure 1.4 for the operation of the quick-union algorithm on the example of Figure 1.1, and Figure 1.6 shows the corresponding changes to the *id* array. The graphical representation of the data structure makes it relatively easy to understand the operation of the algorithm—input pairs that are known to be connected in the data are also connected to one another in the data structure. As mentioned previously, it is important to note at the outset that the connections in the data structure are not necessarily the same as the connections in the application implied by the input pairs; rather, they are constructed by the algorithm to facilitate efficient implementation of *union* and *find*.

The connected components depicted in Figure 1.5 are called *trees*; they are fundamental combinatorial structures that we shall encounter on numerous occasions throughout the book. We shall consider the properties of trees in detail in Chapter 5. For the *union* and *find* operations, the trees in Figure 1.5 are useful because they are quick to build and have the property that two objects are connected in the tree if and only if the objects are connected in the input. By moving up the tree, we can easily find the root of the tree containing each object, so we have a way to find whether or not they are connected. Each tree has precisely one object that points to itself, which is called the *root* of the tree. The self-pointer is not shown in the diagrams. When we start at any object in the tree, move to the object to which it points, then move to the object to which that object points, and so forth, we eventually end up at the root, always. We can prove this property to be true by induction: It is true after the array is initialized to have every object point to itself, and if it is true before a given *union* operation, it is certainly true afterward.

The diagrams in Figure 1.4 for the quick-find algorithm have the same properties as those described in the previous paragraph. The difference between the two is that we reach the root from all the nodes in the quick-find trees after following just one link, whereas we might need to follow several links to get to the root in a quick-union tree.

Program 1.2 is an implementation of the *union* and *find* operations that comprise the quick-union algorithm to solve the connectivity problem. The quick-union algorithm would seem to be faster than the

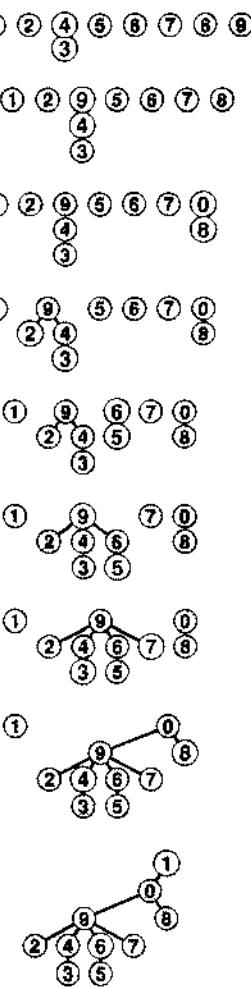


Figure 1.5
Tree representation of quick union

This figure is a graphical representation of the example in Figure 1.3. We draw a line from object *i* to object *id[i]*.

Program 1.2 Quick-union solution to connectivity problem

If we replace the body of the `while` loop in Program 1.1 by this code, we have a program that meets the same specifications as Program 1.1, but does less computation for the *union* operation at the expense of more computation for the *find* operation. The `for` loops and subsequent `if` statement in this code specify the necessary and sufficient conditions on the `id` array for `p` and `q` to be connected. The assignment statement `id[i] = j` implements the *union* operation.

```
for (i = p; i != id[i]; i = id[i]) ;
for (j = q; j != id[j]; j = id[j]) ;
if (i == j) continue;
id[i] = j;
printf("%d %d\n", p, q);
```

quick-find algorithm, because it does not have to go through the entire array for each input pair; but how much faster is it? This question is more difficult to answer here than it was for quick find, because the running time is much more dependent on the nature of the input. By running empirical studies or doing mathematical analysis (see Chapter 2), we can convince ourselves that Program 1.2 is far more efficient than Program 1.1, and that it is feasible to consider using Program 1.2 for huge practical problems. We shall discuss one such empirical study at the end of this section. For the moment, we can regard quick union as an improvement because it removes quick find's main liability (that the program requires at least NM instructions to process M *union* operations among N objects).

This difference between quick union and quick find certainly represents an improvement, but quick union still has the liability that we cannot *guarantee* it to be substantially faster than quick find in every case, because the input data could conspire to make the *find* operation slow.

Property 1.2 *For $M > N$, the quick-union algorithm could take more than $MN/2$ instructions to solve a connectivity problem with M pairs of N objects.*

Suppose that the input pairs come in the order 1-2, then 2-3, then 3-4, and so forth. After $N - 1$ such pairs, we have N objects all in the same set, and the tree that is formed by the quick-union algorithm

p \ q	0	1	2	3	4	5	6	7	8	9
3 4	0	1	2	4	4	5	6	7	8	9
4 9	0	1	2	4	9	5	6	7	8	9
8 0	0	1	2	4	9	5	6	7	0	9
2 3	0	1	9	4	9	5	6	7	0	9
5 6	0	1	9	4	9	6	6	7	0	9
2 9	0	1	9	4	9	6	6	7	0	9
5 9	0	1	9	4	9	6	9	7	0	9
7 3	0	1	9	4	9	6	9	9	0	9
4 8	0	1	9	4	9	6	9	9	0	0
5 6	0	1	9	4	9	6	9	9	0	0
0 2	0	1	9	4	9	6	9	9	0	0
6 1	1	1	9	4	9	6	9	9	0	0
5 8	1	1	9	4	9	6	9	9	0	0

Figure 1.6
Example of quick union (not-too-quick find)

This sequence depicts the contents of the `id` array after each of the pairs at left are processed by the quick-union algorithm (Program 1.2). Shaded entries are those that change for the *union* operation (just one per operation). When we process the pair `p q`, we follow pointers from `p` to get an entry `i` with `id[i] == i`; then, we follow pointers from `q` to get an entry `j` with `id[j] == j`; then, if `i` and `j` differ, we set `id[i] = id[j]`. For the *find* operation for the pair `5-8` (final line), `i` takes on the values `5 6 9 0 1`, and `j` takes on the values `8 0 1`.

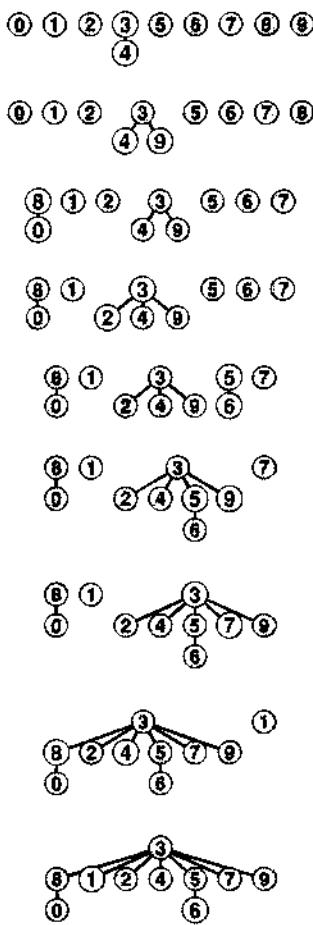


Figure 1.7
Tree representation of weighted quick union

This sequence depicts the result of changing the quick-union algorithm to link the root of the smaller of the two trees to the root of the larger of the two trees. The distance from each node to the root of its tree is small, so the find operation is efficient.

is a straight line, with N pointing to $N - 1$, which points to $N - 2$, which points to $N - 3$, and so forth. To execute the *find* operation for object N , the program has to follow $N - 1$ pointers. Thus, the average number of pointers followed for the first N pairs is

$$(0 + 1 + \dots + (N - 1))/N = (N - 1)/2.$$

Now suppose that the remainder of the pairs all connect N to some other object. The *find* operation for each of these pairs involves at least $(N - 1)$ pointers. The grand total for the M *find* operations for this sequence of input pairs is certainly greater than $MN/2$. ■

Fortunately, there is an easy modification to the algorithm that allows us to guarantee that bad cases such as this one do not occur. Rather than arbitrarily connecting the second tree to the first for *union*, we keep track of the number of nodes in each tree and always connect the smaller tree to the larger. This change requires slightly more code and another array to hold the node counts, as shown in Program 1.3, but it leads to substantial improvements in efficiency. We refer to this algorithm as the *weighted quick-union algorithm*.

Figure 1.7 shows the forest of trees constructed by the weighted union–find algorithm for the example input in Figure 1.1. Even for this small example, the paths in the trees are substantially shorter than for the unweighted version in Figure 1.5. Figure 1.8 illustrates what happens in the worst case, when the sizes of the sets to be merged in the *union* operation are always equal (and a power of 2). These tree structures look complex, but they have the simple property that the maximum number of pointers that we need to follow to get to the root in a tree of 2^n nodes is n . Furthermore, when we merge two trees of 2^n nodes, we get a tree of 2^{n+1} nodes, and we increase the maximum distance to the root to $n + 1$. This observation generalizes to provide a proof that the weighted algorithm is substantially more efficient than the unweighted algorithm.

Property 1.3 *The weighted quick-union algorithm follows at most $2 \lg N$ pointers to determine whether two of N objects are connected.*

We can prove that the *union* operation preserves the property that the number of pointers followed from any node to the root in a set of k objects is no greater than $\lg k$ (we do not count the self-pointer at the root). When we combine a set of i nodes with a set of j nodes with

Program 1.3 Weighted version of quick union

This program is a modification to the quick-union algorithm (see Program 1.2) that keeps an additional array `sz` for the purpose of maintaining, for each object with `id[i] == i`, the number of nodes in the associated tree, so that the *union* operation can link the smaller of the two specified trees to the larger, thus preventing the growth of long paths in the trees.

```
#include <stdio.h>
#define N 10000
main()
{ int i, j, p, q, id[N], sz[N];
  for (i = 0; i < N; i++)
    { id[i] = i; sz[i] = 1; }
  while (scanf("%d %d\n", &p, &q) == 2)
  {
    for (i = p; i != id[i]; i = id[i])
      for (j = q; j != id[j]; j = id[j])
        if (i == j) continue;
        if (sz[i] < sz[j])
          { id[i] = j; sz[j] += sz[i]; }
        else { id[j] = i; sz[i] += sz[j]; }
    printf(" %d %d\n", p, q);
  }
}
```

$i \leq j$, we increase the number of pointers that must be followed in the smaller set by 1, but they are now in a set of size $i + j$, so the property is preserved because $1 + \lg i = \lg(i + i) \leq \lg(i + j)$. ■

The practical implication of Property 1.3 is that the weighted quick-union algorithm uses *at most* a constant times $M \lg N$ instructions to process M edges on N objects (see Exercise 1.9). This result is in stark contrast to our finding that quick find always (and quick union sometimes) uses *at least* $MN/2$ instructions. The conclusion is that, with weighted quick union, we can guarantee that we can solve huge practical problems in a reasonable amount of time (see Exercise 1.11). For the price of a few extra lines of code, we get a program that is

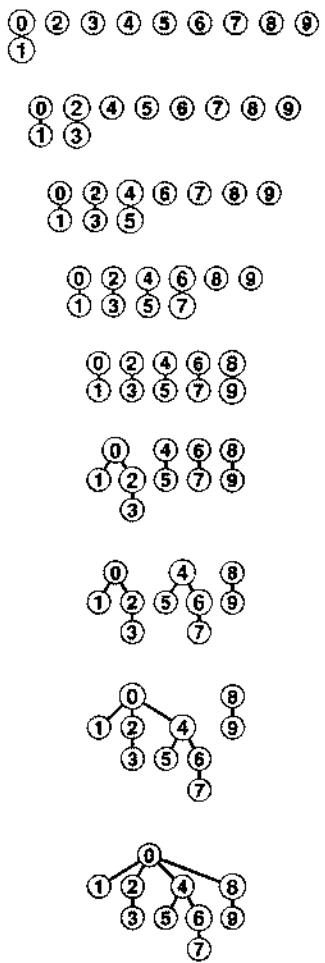


Figure 1.8
Weighted quick union (worst case)

The worst scenario for the weighted quick-union algorithm is that each union operation links trees of equal size. If the number of objects is less than 2^n , the distance from any node to the root of its tree is less than n .

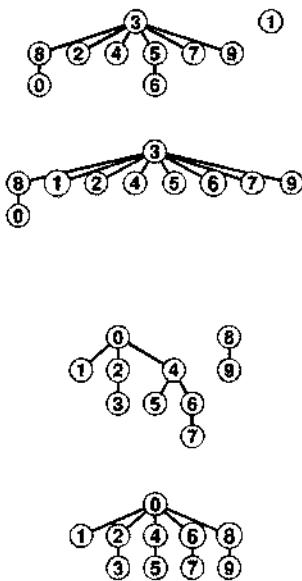


Figure 1.9
Path compression

We can make paths in the trees even shorter by simply making all the objects that we touch point to the root of the new tree for the union operation, as shown in these two examples. The example at the top shows the result corresponding to Figure 1.7. For short paths, path compression has no effect, but when we process the pair 1,6, we make 1, 5, and 6 all point to 3 and get a tree flatter than the one in Figure 1.7. The example at the bottom shows the result corresponding to Figure 1.8. Paths that are longer than one or two links can develop in the trees, but whenever we traverse them, we flatten them. Here, when we process the pair 6,8, we flatten the tree by making 4, 6, and 8 all point to 0.

literally millions of times faster than the simpler algorithms for the huge problems that we might encounter in practical applications.

It is evident from the diagrams that relatively few nodes are far from the root; indeed, empirical studies on huge problems tell us that the weighted quick-union algorithm of Program 1.3 typically can solve practical problems in *linear* time. That is, the cost of running the algorithm is within a constant factor of the cost of reading the input. We could hardly expect to find a more efficient algorithm.

We immediately come to the question of whether or not we can find an algorithm that has *guaranteed* linear performance. This question is an extremely difficult one that plagued researchers for many years (see Section 2.7). There are a number of easy ways to improve the weighted quick-union algorithm further. Ideally, we would like every node to point directly to the root of its tree, but we do not want to pay the price of changing a large number of pointers, as we did in the quick-union algorithm. We can approach the ideal simply by making all the nodes that we do examine point to the root. This step seems drastic at first blush, but it is easy to implement, and there is nothing sacrosanct about the structure of these trees: If we can modify them to make the algorithm more efficient, we should do so. We can implement this method, called *path compression*, easily, by adding another pass through each path during the *union* operation, setting the *id* entry corresponding to each vertex encountered along the way to point to the root. The net result is to flatten the trees almost completely, approximating the ideal achieved by the quick-find algorithm, as illustrated in Figure 1.9. The analysis that establishes this fact is extremely complex, but the method is simple and effective. Figure 1.11 shows the result of path compression for a large example.

There are many other ways to implement path compression. For example, Program 1.4 is an implementation that compresses the paths by making each link skip to the next node in the path on the way up the tree, as depicted in Figure 1.10. This method is slightly easier to implement than full path compression (see Exercise 1.16), and achieves the same net result. We refer to this variant as *weighted quick-union with path compression by halving*. Which of these methods is the more effective? Is the savings achieved worth the extra time required to implement path compression? Is there some other technique that we should consider? To answer these questions, we need to look more

Program 1.4 Path compression by halving

If we replace the `for` loops in Program 1.3 by this code, we halve the length of any path that we traverse. The net result of this change is that the trees become almost completely flat after a long sequence of operations.

```
for (i = p; i != id[i]; i = id[i])
    id[i] = id[id[i]];
for (j = q; j != id[j]; j = id[j])
    id[j] = id[id[j]];
```

carefully at the algorithms and implementations. We shall return to this topic in Chapter 2, in the context of our discussion of basic approaches to the analysis of algorithms.

The end result of the succession of algorithms that we have considered to solve the connectivity problem is about the best that we could hope for in any practical sense. We have algorithms that are easy to implement whose running time is guaranteed to be within a constant factor of the cost of gathering the data. Moreover, the algorithms are *online* algorithms that consider each edge once, using space proportional to the number of objects, so there is no limitation on the number of edges that they can handle. The empirical studies in Table 1.1 validate our conclusion that Program 1.3 and its path-compression variations are useful even for huge practical applications. Choosing which is the best among these algorithms requires careful and sophisticated analysis (see Chapter 2).

Exercises

- ▷ 1.4 Show the contents of the `id` array after each *union* operation when you use the quick-find algorithm (Program 1.1) to solve the connectivity problem for the sequence 0–2, 1–4, 2–5, 3–6, 0–4, 6–0, and 1–3. Also give the number of times the program accesses the `id` array for each input pair.
- ▷ 1.5 Do Exercise 1.4, but use the quick-union algorithm (Program 1.2).
- ▷ 1.6 Give the contents of the `id` array after each *union* operation for the weighted quick-union algorithm running on the examples corresponding to Figure 1.7 and Figure 1.8.
- ▷ 1.7 Do Exercise 1.4, but use the weighted quick-union algorithm (Program 1.3).

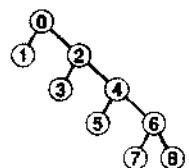


Figure 1.10
Path compression by halving

We can nearly halve the length of paths on the way up the tree by taking two links at a time, and setting the bottom one to point to the same node as the top one, as shown in this example. The net result of performing this operation on every path that we traverse is asymptotically the same as full path compression.

Table 1.1 Empirical study of union-find algorithms

These relative timings for solving random connectivity problems using various union–find algorithms demonstrate the effectiveness of the weighted version of the quick union algorithm. The added incremental benefit due to path compression is less important. In these experiments, M is the number of random connections generated until all N objects were connected. This process involves substantially more *find* operations than *union* operations, so quick union is substantially slower than quick find. Neither quick find nor quick union is feasible for huge N . The running time for the weighted methods is evidently proportional to N , as it approximately doubles when N is doubled.

N	M	F	U	W	P	H
1000	6206	14	25	6	5	3
2500	20236	82	210	13	15	12
5000	41913	304	1172	46	26	25
10000	83857	1216	4577	91	73	50
25000	309802			219	208	216
50000	708701			469	387	497
100000	1545119			1071	1106	1096

Key:

- F quick find (Program 1.1)
 - U quick union (Program 1.2)
 - W weighted quick union (Program 1.3)
 - P weighted quick union with path compression (Exercise 1.16)
 - H weighted quick union with halving (Program 1.4)
-

▷ 1.8 Do Exercise 1.4, but use the weighted quick-union algorithm with path compression by halving (Program 1.4).

1.9 Prove an upper bound on the number of machine instructions required to process M connections on N objects using Program 1.3. You may assume, for example, that any C assignment statement always requires less than c instructions, for some fixed constant c .

1.10 Estimate the minimum amount of time (in days) that would be required for quick find (Program 1.1) to solve a problem with 10^6 objects and 10^9 input pairs, on a computer capable of executing 10^9 instructions per second. Assume that each iteration of the while loop requires at least 10 instructions.

1.11 Estimate the maximum amount of time (in seconds) that would be required for weighted quick union (Program 1.3) to solve a problem with 10^6 objects and 10^9 input pairs, on a computer capable of executing 10^9 instructions per second. Assume that each iteration of the `while` loop requires at most 100 instructions.

1.12 Compute the *average* distance from a node to the root in a worst-case tree of 2^n nodes built by the weighted quick-union algorithm.

▷ 1.13 Draw a diagram like Figure 1.10, starting with eight nodes instead of nine.

○ 1.14 Give a sequence of input pairs that causes the weighted quick-union algorithm (Program 1.3) to produce a path of length 4.

● 1.15 Give a sequence of input pairs that causes the weighted quick-union algorithm with path compression by halving (Program 1.4) to produce a path of length 4.

1.16 Show how to modify Program 1.3 to implement *full* path compression, where we complete each `union` operation by making every node that we touch point to the root of the new tree.

▷ 1.17 Answer Exercise 1.4, but using the weighted quick-union algorithm with full path compression (Exercise 1.16).

●● 1.18 Give a sequence of input pairs that causes the weighted quick-union algorithm with full path compression (Exercise 1.16) to produce a path of length 4.

○ 1.19 Give an example showing that modifying quick union (Program 1.2) to implement full path compression (see Exercise 1.16) is not sufficient to ensure that the trees have no long paths.

● 1.20 Modify Program 1.3 to use the *height* of the trees (longest path from any node to the root), instead of the weight, to decide whether to set `id[i] = j` or `id[j] = i`. Run empirical studies to compare this variant with Program 1.3.

●● 1.21 Show that Property 1.3 holds for the algorithm described in Exercise 1.20.

● 1.22 Modify Program 1.4 to generate random pairs of integers between 0 and $N - 1$ instead of reading them from standard input, and to loop until $N - 1$ `union` operations have been performed. Run your program for $N = 10^3, 10^4, 10^5$, and 10^6 and print out the total number of edges generated for each value of N .

● 1.23 Modify your program from Exercise 1.22 to plot the number of edges needed to connect N items, for $100 \leq N \leq 1000$.

●● 1.24 Give an approximate formula for the number of random edges that are required to connect N objects, as a function of N .

Figure 1.11

A large example of the effect of path compression

This sequence depicts the result of processing random pairs from 100 objects with the weighted quick-union algorithm with path compression. All but two of the nodes in the tree are one or two steps from the root.

1.4 Perspective

Each of the algorithms that we considered in Section 1.3 seems to be an improvement over the previous in some intuitive sense, but the process is perhaps artificially smooth because we have the benefit of hindsight in looking over the development of the algorithms as they were studied by researchers over the years (see reference section). The implementations are simple and the problem is well specified, so we can evaluate the various algorithms directly by running empirical studies. Furthermore, we can validate these studies and quantify the comparative performance of these algorithms (see Chapter 2). Not all the problem domains in this book are as well developed as this one, and we certainly can run into complex algorithms that are difficult to compare and mathematical problems that are difficult to solve. We strive to make objective scientific judgements about the algorithms that we use, while gaining experience learning the properties of implementations running on actual data from applications or random test data.

The process is prototypical of the way that we consider various algorithms for fundamental problems throughout the book. When possible, we follow the same basic steps that we took for union–find algorithms in Section 1.2, some of which are highlighted in this list:

- Decide on a complete and specific problem statement, including identifying fundamental abstract operations that are intrinsic to the problem.
- Carefully develop a succinct implementation for a straightforward algorithm.
- Develop improved implementations through a process of step-wise refinement, validating the efficacy of ideas for improvement through empirical analysis, mathematical analysis, or both.
- Find high-level abstract representations of data structures or algorithms in operation that enable effective high-level design of improved versions.
- Strive for worst-case performance guarantees when possible, but accept good performance on actual data when available.

The potential for spectacular performance improvements for practical problems such as those that we saw in Section 1.2 makes algorithm design a compelling field of study; few other design activities hold the potential to reap savings factors of millions or billions, or more.

More important, as the scale of our computational power and our applications increases, the gap between a fast algorithm and a slow one grows. A new computer might be 10 times faster and be able to process 10 times as much data as an old one, but if we are using a quadratic algorithm such as quick find, the new computer will take 10 times as long on the new job as the old one took to finish the old job! This statement seems counterintuitive at first, but it is easily verified by the simple identity $(10N)^2/10 = 10N^2$, as we shall see in Chapter 2. As computational power increases to allow us to take on larger and larger problems, the importance of having efficient algorithms increases, as well.

Developing an efficient algorithm is an intellectually satisfying activity that can have direct practical payoff. As the connectivity problem indicates, a simply stated problem can lead us to study numerous algorithms that are not only both useful and interesting, but also intricate and challenging to understand. We shall encounter many ingenious algorithms that have been developed over the years for a host of practical problems. As the scope of applicability of computational solutions to scientific and commercial problems widens, so also grows the importance of being able to apply efficient algorithms to solve known problems and of being able to develop efficient solutions to new problems.

Exercises

1.25 Suppose that we use weighted quick union to process 10 times as many connections on a new computer that is 10 times as fast as an old one. How much longer would it take the new computer to finish the new job than it took the old one to finish the old job?

1.26 Answer Exercise 1.25 for the case where we use an algorithm that requires N^3 instructions.

1.5 Summary of Topics

This section comprises brief descriptions of the major parts of the book, giving specific topics covered and an indication of our general

orientation toward the material. This set of topics is intended to touch on as many fundamental algorithms as possible. Some of the areas covered are core computer-science areas that we study in depth to learn basic algorithms of wide applicability. Other algorithms that we discuss are from advanced fields of study within computer science and related fields, such as numerical analysis and operations research—in these cases, our treatment serves as an introduction to these fields through examination of basic methods.

The first four parts of the book, which are contained in this volume, cover the most widely used set of algorithms and data structures, a first level of abstraction for collections of objects with keys that can support a broad variety of important fundamental algorithms. The algorithms that we consider are the products of decades of research and development, and continue to play an essential role in the ever-expanding applications of computation.

Fundamentals (Part 1) in the context of this book are the basic principles and methodology that we use to implement, analyze, and compare algorithms. The material in Chapter 1 motivates our study of algorithm design and analysis; in Chapter 2, we consider basic methods of obtaining quantitative information about the performance of algorithms.

Data Structures (Part 2) go hand-in-hand with algorithms: we shall develop a thorough understanding of data representation methods for use throughout the rest of the book. We begin with an introduction to basic concrete data structures in Chapter 3, including arrays, linked lists, and strings; then we consider recursive programs and data structures in Chapter 5, in particular trees and algorithms for manipulating them. In Chapter 4, we consider fundamental abstract data types (ADTs) such as stacks and queues, including implementations using elementary data structures.

Sorting algorithms (Part 3) for rearranging files into order are of fundamental importance. We consider a variety of algorithms in considerable depth, including Shellsort, quicksort, mergesort, heapsort, and radix sorts. We shall encounter algorithms for several related problems, including priority queues, selection, and merging. Many of these algorithms will find application as the basis for other algorithms later in the book.

Searching algorithms (Part 4) for finding specific items among large collections of items are also of fundamental importance. We discuss basic and advanced methods for searching using trees and digital key transformations, including binary search trees, balanced trees, hashing, digital search trees and tries, and methods appropriate for huge files. We note relationships among these methods, comparative performance statistics, and correspondences to sorting methods.

Parts 5 through 8, which are contained in a separate volume, cover advanced applications of the algorithms described here for a diverse set of applications—a second level of abstractions specific to a number of important applications areas. We also delve more deeply into techniques of algorithm design and analysis. Many of the problems that we touch on are the subject of ongoing research.

String Processing algorithms (Part 5) include a range of methods for processing (long) sequences of characters. String searching leads to pattern matching, which leads to parsing. File-compression techniques are also considered. Again, an introduction to advanced topics is given through treatment of some elementary problems that are important in their own right.

Geometric Algorithms (Part 6) are methods for solving problems involving points and lines (and other simple geometric objects) that have only recently come into use. We consider algorithms for finding the convex hull of a set of points, for finding intersections among geometric objects, for solving closest-point problems, and for multidimensional searching. Many of these methods nicely complement the more elementary sorting and searching methods.

Graph Algorithms (Part 7) are useful for a variety of difficult and important problems. A general strategy for searching in graphs is developed and applied to fundamental connectivity problems, including shortest path, minimum spanning tree, network flow, and matching. A unified treatment of these algorithms shows that they are all based on the same procedure, and that this procedure depends on the basic priority queue ADT.

Advanced Topics (Part 8) are discussed for the purpose of relating the material in the book to several other advanced fields of study. We begin with major approaches to the design and analysis of algorithms, including divide-and-conquer, dynamic programming, randomization,

and amortization. We survey linear programming, the fast Fourier transform, NP-completeness, and other advanced topics from an introductory viewpoint to gain appreciation for the interesting advanced fields of study suggested by the elementary problems confronted in this book.

The study of algorithms is interesting because it is a new field (almost all the algorithms that we study are less than 50 years old, and some were just recently discovered) with a rich tradition (a few algorithms have been known for thousands of years). New discoveries are constantly being made, but few algorithms are completely understood. In this book we shall consider intricate, complicated, and difficult algorithms as well as elegant, simple, and easy algorithms. Our challenge is to understand the former and to appreciate the latter in the context of many different potential applications. In doing so, we shall explore a variety of useful tools and develop a style of algorithmic thinking that will serve us well in computational challenges to come.

CHAPTER TWO

Principles of Algorithm Analysis

ANALYSIS IS THE key to being able to understand algorithms sufficiently well that we can apply them effectively to practical problems. Although we cannot do extensive experimentation and deep mathematical analysis on each and every program that we run, we can work within a basic framework involving both empirical testing and approximate analysis that can help us to know the important facts about the performance characteristics of our algorithms, so that we may compare those algorithms and can apply them to practical problems.

The very idea of describing the performance of a complex algorithm accurately with a mathematical analysis seems a daunting prospect at first, and we do often call on the research literature for results based on detailed mathematical study. Although it is not our purpose in this book to cover methods of analysis or even to summarize these results, it is important for us to be aware at the outset that we are on firm scientific ground when we want to compare different methods. Moreover, a great deal of detailed information is available about many of our most important algorithms through careful application of relatively few elementary techniques. We do highlight basic analytic results and methods of analysis throughout the book, particularly when such understanding helps us to understand the inner workings of fundamental algorithms. Our primary goal in this chapter is to provide the context and the tools that we need to work intelligently with the algorithms themselves.

The example in Chapter 1 provides a context that illustrates many of the basic concepts of algorithm analysis, so we frequently refer

back to the performance of union-find algorithms to make particular points concrete. We also consider a detailed pair of new examples, in Section 2.6.

Analysis plays a role at every point in the process of designing and implementing algorithms. At first, as we saw, we can save factors of thousands or millions in the running time with appropriate algorithm design choices. As we consider more efficient algorithms, we find it more of a challenge to choose among them, so we need to study their properties in more detail. In pursuit of the *best* (in some precise technical sense) algorithm, we find both algorithms that are useful in practice and theoretical questions that are challenging to resolve.

Complete coverage of methods for the analysis of algorithms is the subject of a book in itself (*see reference section*), but it is worthwhile for us to consider the basics here, so that we can

- Illustrate the process.
- Describe in one place the mathematical conventions that we use.
- Provide a basis for discussion of higher-level issues.
- Develop an appreciation for scientific underpinnings of the conclusions that we draw when comparing algorithms.

Most important, algorithms and their analyses are often intertwined. In this book, we do not delve into deep and difficult mathematical derivations, but we do use sufficient mathematics to be able to understand what our algorithms are and how we can use them effectively.

2.1 Implementation and Empirical Analysis

We design and develop algorithms by layering abstract operations that help us to understand the essential nature of the computational problems that we want to solve. In theoretical studies, this process, although valuable, can take us far afield from the real-world problems that we need to consider. Thus, in this book, we keep our feet on the ground by expressing all the algorithms that we consider in an actual programming language: C. This approach sometimes leaves us with a blurred distinction between an algorithm and its implementation, but that is small price to pay for the ability to work with and to learn from a concrete implementation.

Indeed, carefully constructed programs in an actual programming language provide an effective means of expressing our algorithms.

In this book, we consider a large number of important and efficient algorithms that we describe in implementations that are both concise and precise in C. English-language descriptions or abstract high-level representations of algorithms are all too often vague or incomplete; actual implementations force us to discover economical representations to avoid being inundated in detail.

We express our algorithms in C, but this book is about algorithms, rather than about C programming. Certainly, we consider C implementations for many important tasks, and, when there is a particularly convenient or efficient way to do a task in C, we will take advantage of it. But the vast majority of the implementation decisions that we make are worth considering in any modern programming environment. Translating the programs in Chapter 1, and most of the other programs in this book, to another modern programming language is a straightforward task. On occasion, we also note when some other language provides a particularly effective mechanism suited to the task at hand. Our goal is to use C as a vehicle for expressing the algorithms that we consider, rather than to dwell on implementation issues specific to C.

If an algorithm is to be implemented as part of a large system, we use abstract data types or a similar mechanism to make it possible to change algorithms or implementations after we determine what part of the system deserves the most attention. From the start, however, we need to have an understanding of each algorithm's performance characteristics, because design requirements of the system may have a major influence on algorithm performance. Such initial design decisions must be made with care, because it often does turn out, in the end, that the performance of the whole system depends on the performance of some basic algorithm, such as those discussed in this book.

Implementations of the algorithms in this book have been put to effective use in a wide variety of large programs, operating systems, and applications systems. Our intention is to describe the algorithms and to encourage a focus on their dynamic properties through experimentation with the implementations given. For some applications, the implementations may be quite useful exactly as given; for other applications, however, more work may be required. For example, using a more defensive programming style than the one that we use in this

book is justified when we are building real systems. Error conditions must be checked and reported, and programs must be implemented such that they can be changed easily, read and understood quickly by other programmers, interface well with other parts of the system, and be amenable to being moved to other environments.

Notwithstanding all these comments, we take the position when analyzing each algorithm that performance is of critical importance, to focus our attention on the algorithm's essential performance characteristics. We assume that we are always interested in knowing about algorithms with substantially better performance, particularly if they are simpler.

To use an algorithm effectively, whether our goal is to solve a huge problem that could not otherwise be solved, or whether our goal is to provide an efficient implementation of a critical part of a system, we need to have an understanding of its performance characteristics. Developing such an understanding is the goal of algorithmic analysis.

One of the first steps that we take to understand the performance of algorithms is to do *empirical analysis*. Given two algorithms to solve the same problem, there is no mystery in the method: We run them both to see which one takes longer! This concept might seem too obvious to mention, but it is an all-too-common omission in the comparative study of algorithms. The fact that one algorithm is 10 times faster than another is unlikely to escape the notice of someone who waits 3 seconds for one to finish and 30 seconds for the other to finish, but it is easy to overlook as a small constant overhead factor in a mathematical analysis. When we monitor the performance of careful implementations on typical input, we get performance results that not only give us a direct indicator of efficiency, but also provide us with the information that we need to compare algorithms and to validate any mathematical analyses that may apply (see, for example, Table 1.1). When empirical studies start to consume a significant amount of time, mathematical analysis is called for. Waiting an hour or a day for a program to finish is hardly a productive way to find out that it is slow, particularly when a straightforward analysis can give us the same information.

The first challenge that we face in empirical analysis is to develop a correct and complete implementation. For some complex algorithms, this challenge may present a significant obstacle. Accordingly, we

typically want to have, through analysis or through experience with similar programs, some indication of how efficient a program might be before we invest too much effort in getting it to work.

The second challenge that we face in empirical analysis is to determine the nature of the input data and other factors that have direct influence on the experiments to be performed. Typically, we have three basic choices: use *actual* data, *random* data, or *perverse* data. Actual data enable us truly to measure the cost of the program in use; random data assure us that our experiments test the algorithm, not the data; and perverse data assure us that our programs can handle any input presented them. For example, when we test sorting algorithms, we run them on data such as the words in *Moby Dick*, on randomly generated integers, and on files of numbers that are all the same value. This problem of determining which input data to use to compare algorithms also arises when we analyze the algorithms.

It is easy to make mistakes when we compare implementations, particularly if differing machines, compilers, or systems are involved, or if huge programs with ill-specified inputs are being compared. The principal danger in comparing programs empirically is that one implementation may be coded more carefully than the other. The inventor of a proposed new algorithm is likely to pay careful attention to every aspect of its implementation, and not to expend so much effort on the details of implementing a classical competing algorithm. To be confident of the accuracy of an empirical study comparing algorithms, we must be sure to give the same attention to each implementation.

One approach that we often use in this book, as we saw in Chapter 1, is to derive algorithms by making relatively minor modifications to other algorithms for the same problem, so comparative studies really are valid. More generally, we strive to identify essential abstract operations, and start by comparing algorithms on the basis of their use of such operations. For example, the comparative empirical results that we examined in Table 1.1 are likely to be robust across programming languages and environments, as they involve programs that are similar and that make use of the same set of basic operations. For a particular programming environment, we can easily relate these numbers to actual running times. Most often, we simply want to know which of two programs is likely to be faster, or to what extent a certain change will improve the time or space requirements of a certain program.

Choosing among algorithms to solve a given problem is tricky business. Perhaps the most common mistake made in selecting an algorithm is to ignore performance characteristics. Faster algorithms are often more complicated than brute-force solutions, and implementors are often willing to accept a slower algorithm to avoid having to deal with added complexity. As we saw with union-find algorithms, however, we can sometimes reap huge savings with just a few lines of code. Users of a surprising number of computer systems lose substantial time waiting for simple quadratic algorithms to finish when $N \log N$ algorithms are available that are only slightly more complicated and could run in a fraction of the time. When we are dealing with huge problem sizes, we have no choice but to seek a better algorithm, as we shall see.

Perhaps the second most common mistake made in selecting an algorithm is to pay too much attention to performance characteristics. Improving the running time of a program by a factor of 10 is inconsequential if the program takes only a few microseconds. Even if a program takes a few minutes, it may not be worth the time and effort required to make it run 10 times faster, particularly if we expect to use the program only a few times. The total time required to implement and debug an improved algorithm might be substantially more than the time required simply to run a slightly slower one—we may as well let the computer do the work. Worse, we may spend a considerable amount of time and effort implementing ideas that should improve a program but actually do not do so.

We cannot run empirical tests for a program that is not yet written, but we can analyze properties of the program and estimate the potential effectiveness of a proposed improvement. Not all putative improvements actually result in performance gains, and we need to understand the extent of the savings realized at each step. Moreover, we can include parameters in our implementations, and can use analysis to help us set the parameters. Most important, by understanding the fundamental properties of our programs and the basic nature of the programs' resource usage, we hold the potentials to evaluate their effectiveness on computers not yet built and to compare them against new algorithms not yet designed. In Section 2.2, we outline our methodology for developing a basic understanding of algorithm performance.

Exercises

2.1 Translate the programs in Chapter 1 to another programming language, and answer Exercise 1.22 for your implementations.

2.2 How long does it take to count to 1 billion (ignoring overflow)? Determine the amount of time it takes the program

```
int i, j, k, count = 0;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            count++;
```

to complete in your programming environment, for $N = 10, 100$, and 1000 . If your compiler has optimization features that are supposed to make programs more efficient, check whether or not they do so for this program.

2.2 Analysis of Algorithms

In this section, we outline the framework within which mathematical analysis can play a role in the process of comparing the performance of algorithms, to lay a foundation for us to be able to consider basic analytic results as they apply to the fundamental algorithms that we consider throughout the book. We shall consider the basic mathematical tools that are used in the analysis of algorithms, both to allow us to study classical analyses of fundamental algorithms and to make use of results from the research literature that help us understand the performance characteristics of our algorithms.

The following are among the reasons that we perform mathematical analysis of algorithms:

- To compare different algorithms for the same task
- To predict performance in a new environment
- To set values of algorithm parameters

We shall see many examples of each of these reasons throughout the book. Empirical analysis might suffice for some of these tasks, but mathematical analysis can be more informative (and less expensive!), as we shall see.

The analysis of algorithms can be challenging indeed. Some of the algorithms in this book are well understood, to the point that accurate mathematical formulas are known that can be used to predict running time in practical situations. People develop such formulas by carefully studying the program, to find the running time in terms of fundamental

mathematical quantities, and then doing a mathematical analysis of the quantities involved. On the other hand, the performance properties of other algorithms in this book are not fully understood—perhaps their analysis leads to unsolved mathematical questions, or perhaps known implementations are too complex for a detailed analysis to be reasonable, or (most likely) perhaps the types of input that they encounter cannot be characterized accurately.

Several important factors in a precise analysis are usually outside a given programmer's domain of influence. First, C programs are translated into machine code for a given computer, and it can be a challenging task to figure out exactly how long even one C statement might take to execute (especially in an environment where resources are being shared, so even the same program can have varying performance characteristics at two different times). Second, many programs are extremely sensitive to their input data, and performance might fluctuate wildly depending on the input. Third, many programs of interest are not well understood, and specific mathematical results may not be available. Finally, two programs might not be comparable at all: one may run much more efficiently on one particular kind of input, the other runs efficiently under other circumstances.

All these factors notwithstanding, it is often possible to predict precisely how long a particular program will take, or to know that one program will do better than another in particular situations. Moreover, we can often acquire such knowledge by using one of a relatively small set of mathematical tools. It is the task of the algorithm analyst to discover as much information as possible about the performance of algorithms; it is the task of the programmer to apply such information in selecting algorithms for particular applications. In this and the next several sections, we concentrate on the idealized world of the analyst. To make effective use of our best algorithms, we need to be able to step into this world, on occasion.

The first step in the analysis of an algorithm is to identify the abstract operations on which the algorithm is based, to separate the analysis from the implementation. Thus, for example, we separate the study of how many times one of our *union-find* implementations executes the code fragment `i = a[i]` from the analysis of how many nanoseconds might be required to execute that particular code fragment on our computer. We need both these elements to determine

the actual running time of the program on a particular computer. The former is determined by properties of the algorithm; the latter by properties of the computer. This separation often allows us to compare algorithms in a way that is independent of particular implementations or of particular computers.

Although the number of abstract operations involved can be large, in principle, the performance of an algorithm typically depends on only a few quantities, and typically the most important quantities to analyze are easy to identify. One way to identify them is to use a profiling mechanism (a mechanism available in many C implementations that gives instruction-frequency counts) to determine the most frequently executed parts of the program for some sample runs. Or, like the union-find algorithms of Section 1.3, our implementation might be built on a few abstract operations. In either case, the analysis amounts to determining the frequency of execution of a few fundamental operations. Our modus operandi will be to look for rough estimates of these quantities, secure in the knowledge that we can undertake a fuller analysis for important programs when necessary. Moreover, as we shall see, we can often use approximate analytic results in conjunction with empirical studies to predict performance accurately.

We also have to study the data, and to model the input that might be presented to the algorithm. Most often, we consider one of two approaches to the analysis: we either assume that the input is random, and study the *average-case* performance of the program, or we look for perverse input, and study the *worst-case* performance of the program. The process of characterizing random inputs is difficult for many algorithms, but for many other algorithms it is straightforward and leads to analytic results that provide useful information. The average case might be a mathematical fiction that is not representative of the data on which the program is being used, and the worst case might be a bizarre construction that would never occur in practice, but these analyses give useful information on performance in most cases. For example, we can test analytic results against empirical results (see Section 2.1). If they match, we have increased confidence in both; if they do not match, we can learn about the algorithm and the model by studying the discrepancies.

In the next three sections, we briefly survey the mathematical tools that we shall be using throughout the book. This material is

outside our primary narrative thrust, and readers with a strong background in mathematics or readers who are not planning to check our mathematical statements on the performance of algorithms in detail may wish to skip to Section 2.6 and to refer back to this material when warranted later in the book. The mathematical underpinnings that we consider, however, are generally not difficult to comprehend, and they are too close to core issues of algorithm design to be ignored by anyone wishing to use a computer effectively.

First, in Section 2.3, we consider the mathematical functions that we commonly need to describe the performance characteristics of algorithms. Next, in Section 2.4, we consider the *O*-notation, and the notion of *is proportional to*, which allow us to suppress detail in our mathematical analyses. Then, in Section 2.5, we consider *recurrence relations*, the basic analytic tool that we use to capture the performance characteristics of an algorithm in a mathematical equation. Following this survey, we consider examples where we use the basic tools to analyze specific algorithms, in Section 2.6.

Exercises

- 2.3 Develop an expression of the form $c_0 + c_1N + c_2N^2 + c_3N^3$ that accurately describes the running time of your program from Exercise 2.2. Compare the times predicted by this expression with actual times, for $N = 10, 100$, and 1000 .
- 2.4 Develop an expression that accurately describes the running time of Program 1.1 in terms of M and N .

2.3 Growth of Functions

Most algorithms have a *primary parameter* N that affects the running time most significantly. The parameter N might be the degree of a polynomial, the size of a file to be sorted or searched, the number of characters in a text string, or some other abstract measure of the size of the problem being considered; it is most often directly proportional to the size of the data set being processed. When there is more than one such parameter (for example, M and N in the *union-find* algorithms that we discussed in Section 1.3), we often reduce the analysis to just one parameter by expressing one of the parameters as a function of the other or by considering one parameter at a time (holding the other constant), so we can restrict ourselves to considering a single parameter

N without loss of generality. Our goal is to express the resource requirements of our programs (most often running time) in terms of N , using mathematical formulas that are as simple as possible and that are accurate for large values of the parameters. The algorithms in this book typically have running times proportional to one of the following functions:

- 1 Most instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that the program's running time is *constant*.
- $\log N$ When the running time of a program is *logarithmic*, the program gets slightly slower as N grows. This running time commonly occurs in programs that solve a big problem by transformation into a series of smaller problems, cutting the problem size by some constant fraction at each step. For our range of interest, we can consider the running time to be less than a large constant. The base of the logarithm changes the constant, but not by much: When N is 1 thousand, $\log N$ is 3 if the base is 10, or is about 10 if the base is 2; when N is 1 million, $\log N$ is only double these values. Whenever N doubles, $\log N$ increases by a constant, but $\log N$ does not double until N increases to N^2 .
- N When the running time of a program is *linear*, it is generally the case that a small amount of processing is done on each input element. When N is 1 million, then so is the running time. Whenever N doubles, then so does the running time. This situation is optimal for an algorithm that must process N inputs (or produce N outputs).
- $N \log N$ The $N \log N$ running time arises when algorithms solve a problem by breaking it up into smaller subproblems, solving them independently, and then combining the solutions. For lack of a better adjective (*linearithmic?*), we simply say that the running time of such an algorithm is $N \log N$. When N is 1 million, $N \log N$ is perhaps 20 million. When N doubles, the running time more (but not much more) than doubles.

- N^2 When the running time of an algorithm is *quadratic*, that algorithm is practical for use on only relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop). When N is 1 thousand, the running time is 1 million. Whenever N doubles, the running time increases fourfold.
- N^3 Similarly, an algorithm that processes triples of data items (perhaps in a triple-nested loop) has a *cubic* running time and is practical for use on only small problems. When N is 100, the running time is 1 million. Whenever N doubles, the running time increases eightfold.
- 2^N Few algorithms with *exponential* running time are likely to be appropriate for practical use, even though such algorithms arise naturally as brute-force solutions to problems. When N is 20, the running time is 1 million. Whenever N doubles, the running time squares!

seconds	
10^2	1.7 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
10^{11}	never

Figure 2.1
Seconds conversions

The vast difference between numbers such as 10^4 and 10^8 is more obvious when we consider them to measure time in seconds and convert to familiar units of time. We might let a program run for 2.8 hours, but we would be unlikely to contemplate running a program that would take at least 3.1 years to complete. Because 2^{10} is approximately 10^3 , this table is useful for powers of 2 as well. For example, 2^{32} seconds is about 124 years.

The running time of a particular program is likely to be some constant multiplied by one of these terms (the *leading term*) plus some smaller terms. The values of the constant coefficient and the terms included depend on the results of the analysis and on implementation details. Roughly, the coefficient of the leading term has to do with the number of instructions in the inner loop: At any level of algorithm design, it is prudent to limit the number of such instructions. For large N , the effect of the leading term dominates; for small N or for carefully engineered algorithms, more terms may contribute and comparisons of algorithms are more difficult. In most cases, we will refer to the running time of programs simply as “linear,” “ $N \log N$,” “cubic,” and so forth. We consider the justification for doing so in detail in Section 2.4.

Eventually, to reduce the total running time of a program, we focus on minimizing the number of instructions in the inner loop. Each instruction comes under scrutiny: Is it really necessary? Is there a more efficient way to accomplish the same task? Some programmers believe that the automatic tools provided by modern compilers can produce the best machine code; others believe that the best route is to hand-code inner loops into machine or assembly language. We normally

Table 2.1 Time to solve huge problems

For many applications, our only chance to be able to solve huge problem instances is to use an efficient algorithm. This table indicates the minimum amount of time required to solve problems of size 1 million and 1 billion, using linear, $N \log N$, and quadratic algorithms, on computers capable of executing 1 million, 1 billion, and 1 trillion instructions per second. A fast algorithm enables us to solve a problem on a slow machine, but a fast machine is no help when we are using a slow algorithm.

operations per second	problem size 1 million			problem size 1 billion		
	N	$N \lg N$	N^2	N	$N \lg N$	N^2
10^6	seconds	seconds	weeks	hours	hours	never
10^9	instant	instant	hours	seconds	seconds	decades
10^{12}	instant	instant	seconds	instant	instant	weeks

stop short of considering optimization at this level, although we do occasionally take note of how many machine instructions are required for certain operations, to help us understand why one algorithm might be faster than another in practice.

For small problems, it makes scant difference which method we use—a fast modern computer will complete the job in an instant. But as problem size increases, the numbers we deal with can become huge, as indicated in Table 2.2. As the number of instructions to be executed by a slow algorithm becomes truly huge, the time required to execute those instructions becomes infeasible, even for the fastest computers. Figure 2.1 gives conversion factors from large numbers of seconds to days, months, years, and so forth; Table 2.1 gives examples showing how fast algorithms are more likely than fast computers to be able to help us solve problems without facing outrageous running times.

A few other functions do arise. For example, an algorithm with N^2 inputs that has a running time proportional to N^3 is best thought of as an $N^{3/2}$ algorithm. Also, some algorithms have two stages of subproblem decomposition, which lead to running times proportional to $N \log^2 N$. It is evident from Table 2.2 that both of these functions are much closer to $N \log N$ than to N^2 .

Table 2.2 Values of commonly encountered functions

This table indicates the relative size of some of the functions that we encounter in the analysis of algorithms. The quadratic function clearly dominates, particularly for large N , and differences among smaller functions may not be as we might expect for small N . For example, $N^{3/2}$ should be greater than $N \lg^2 N$ for huge values of N , but $N \lg^2 N$ is greater for the smaller values of N that might occur in practice. A precise characterization of the running time of an algorithm might involve linear combinations of these functions. We can easily separate fast algorithms from slow ones because of vast differences between, for example, $\lg N$ and N or N and N^2 , but distinguishing among fast algorithms involves careful study.

$\lg N$	\sqrt{N}	N	$N \lg N$	$N(\lg N)^2$	$N^{3/2}$	N^2
3	3	10	33	110	32	100
7	10	100	664	4414	1000	10000
10	32	1000	9966	99317	31623	1000000
13	100	10000	132877	1765633	1000000	100000000
17	316	100000	1660964	27589016	31622777	10000000000
20	1000	1000000	19931569	397267426	10000000000	10000000000000

The logarithm function plays a special role in the design and analysis of algorithms, so it is worthwhile for us to consider it in detail. Because we often deal with analytic results only to within a constant factor, we use the notation “ $\log N$ ” without specifying the base. Changing the base from one constant to another changes the value of the logarithm by only a constant factor, but specific bases normally suggest themselves in particular contexts. In mathematics, the *natural logarithm* (base $e = 2.71828\dots$) is so important that a special abbreviation is commonly used: $\log_e N \equiv \ln N$. In computer science, the *binary logarithm* (base 2) is so important that the abbreviation $\log_2 N \equiv \lg N$ is commonly used.

Occasionally, we iterate the logarithm: We apply it successively to a huge number. For example, $\lg \lg 2^{256} = \lg 256 = 8$. As illustrated by this example, we generally regard $\log \log N$ as a constant, for practical purposes, because it is so small, even when N is huge.

The smallest integer larger than $\lg N$ is the number of bits required to represent N in binary, in the same way that the smallest integer larger than $\log_{10} N$ is the number of digits required to represent N in decimal. The C statement

```
for (lgN = 0; N > 0; lgN++, N /= 2) ;
```

is a simple way to compute the smallest integer larger than $\lg N$. A similar method for computing this function is

```
for (lgN = 0, t = 1; t < N; lgN++, t += t) ;
```

This version emphasizes that $2^n \leq N < 2^{n+1}$ when n is the smallest integer larger than $\lg N$.

We also frequently encounter a number of special functions and mathematical notations from classical analysis that are useful in providing concise descriptions of properties of programs. Table 2.3 summarizes the most familiar of these functions; we briefly discuss them and some of their most important properties in the following paragraphs.

Our algorithms and analyses most often deal with discrete units, so we often have need for the following special functions to convert real numbers to integers:

$\lfloor x \rfloor$: largest integer less than or equal to x

$\lceil x \rceil$: smallest integer greater than or equal to x .

For example, $\lfloor \pi \rfloor$ and $\lceil e \rceil$ are both equal to 3, and $\lceil \lg(N + 1) \rceil$ is the number of bits in the binary representation of N . Another important use of these functions arises when we want to divide a set of N objects in half. We cannot do so exactly if N is odd, so, to be precise, we divide into one subset with $\lfloor N/2 \rfloor$ objects and another subset with $\lceil N/2 \rceil$ objects. If N is even, the two subsets are equal in size ($\lfloor N/2 \rfloor = \lceil N/2 \rceil$); if N is odd, they differ in size by 1 ($\lfloor N/2 \rfloor + 1 = \lceil N/2 \rceil$). In C, we can compute these functions directly when we are operating on integers (for example, if $N \geq 0$, then $N/2$ is $\lfloor N/2 \rfloor$ and $N - (N/2)$ is $\lceil N/2 \rceil$), and we can use `floor` and `ceil` from `math.h` to compute them when we are operating on floating point numbers.

A discretized version of the natural logarithm function called the *harmonic numbers* often arises in the analysis of algorithms. The N th harmonic number is defined by the equation

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N}.$$

Table 2.3 Special functions and constants

This table summarizes the mathematical notation that we use for functions and constants that arise in formulas describing the performance of algorithms. The formulas for the approximate values extend to provide much more accuracy, if desired (see reference section).

<i>function</i>	<i>name</i>	<i>typical value</i>	<i>approximation</i>
$\lfloor x \rfloor$	floor function	$\lfloor 3.14 \rfloor = 3$	x
$\lceil x \rceil$	ceiling function	$\lceil 3.14 \rceil = 4$	x
$\lg N$	binary logarithm	$\lg 1024 = 10$	$1.44 \ln N$
F_N	Fibonacci numbers	$F_{10} = 55$	$\phi^N / \sqrt{5}$
H_N	harmonic numbers	$H_{10} \approx 2.9$	$\ln N + \gamma$
$N!$	factorial function	$10! = 3628800$	$(N/e)^N$
$\lg(N!)$		$\lg(100!) \approx 520$	$N \lg N - 1.44N$
$e = 2.71828\dots$			
$\gamma = 0.57721\dots$			
$\phi = (1 + \sqrt{5})/2 = 1.61803\dots$			
$\ln 2 = 0.693147\dots$			
$\lg e = 1/\ln 2 = 1.44269\dots$			

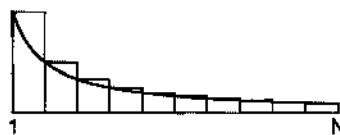


Figure 2.2
Harmonic numbers

The harmonic numbers are an approximation to the area under the curve $y = 1/x$. The constant γ accounts for the difference between H_N and $\ln N = \int_1^N dx/x$.

The natural logarithm $\ln N$ is the area under the curve $1/x$ between 1 and N ; the harmonic number H_N is the area under the step function that we define by evaluating $1/x$ at the integers between 1 and N . This relationship is illustrated in Figure 2.2. The formula

$$H_N \approx \ln N + \gamma + 1/(12N),$$

where $\gamma = 0.57721\dots$ (this constant is known as *Euler's constant*) gives an excellent approximation to H_N . By contrast with $\lfloor \lg N \rfloor$ and $\lceil \lg N \rceil$, it is better to use the library `log` function to compute H_N than to do so directly from the definition.

The sequence of numbers

$$0 \ 1 \ 1 \ 2 \ 3 \ 5 \ 8 \ 13 \ 21 \ 34 \ 55 \ 89 \ 144 \ 233 \ 377 \dots$$

that are defined by the formula

$$F_N = F_{N-1} + F_{N-2}, \quad \text{for } N \geq 2 \text{ with } F_0 = 0 \text{ and } F_1 = 1$$

are known as the *Fibonacci numbers*, and they have many interesting properties. For example, the ratio of two successive terms approaches the *golden ratio* $\phi = (1 + \sqrt{5})/2 \approx 1.61803\dots$. More detailed analysis shows that F_N is $\phi^N/\sqrt{5}$ rounded to the nearest integer.

We also have occasion to manipulate the familiar *factorial* function $N!$. Like the exponential function, the factorial arises in the brute-force solution to problems and grows much too fast for such solutions to be of practical interest. It also arises in the analysis of algorithms because it represents all the ways to arrange N objects. To approximate $N!$, we use *Stirling's formula*:

$$\lg N! \approx N \lg N - N \lg e + \lg \sqrt{2\pi N}.$$

For example, Stirling's formula tells us that the number of bits in the binary representation of $N!$ is about $N \lg N$.

Most of the formulas that we consider in this book are expressed in terms of the few functions that we have described in this section. Many other special functions can arise in the analysis of algorithms. For example, the classical *binomial distribution* and related *Poisson approximation* play an important role in the design and analysis of some of the fundamental search algorithms that we consider in Chapters 14 and 15. We discuss functions not listed here when we encounter them.

Exercises

- ▷ 2.5 For what values of N is $10N \lg N > 2N^2$?
- ▷ 2.6 For what values of N is $N^{3/2}$ between $N(\lg N)^2/2$ and $2N(\lg N)^2$?
- 2.7 For what values of N is $2NH_N - N < N \lg N + 10N$?
- 2.8 What is the smallest value of N for which $\log_{10} \log_{10} N > 8$?
- 2.9 Prove that $\lfloor \lg N \rfloor + 1$ is the number of bits required to represent N in binary.
- 2.10 Add columns to Table 2.1 for $N(\lg N)^2$ and $N^{3/2}$.
- 2.11 Add rows to Table 2.1 for 10^7 and 10^8 instructions per second.
- 2.12 Write a C function that computes H_N , using the `log` function from the standard math library.

- 2.13 Write an efficient C function that computes $\lceil \lg \lg N \rceil$. Do not use a library function.
- 2.14 How many digits are there in the decimal representation of 1 million factorial?
- 2.15 How many bits are there in the binary representation of $\lg(N!)$?
- 2.16 How many bits are there in the binary representation of H_N ?
- 2.17 Give a simple expression for $\lfloor \lg F_N \rfloor$.
- o 2.18 Give the smallest values of N for which $\lfloor H_N \rfloor = i$ for $1 \leq i \leq 10$.
- 2.19 Give the largest value of N for which you can solve a problem that requires at least $f(N)$ instructions on a machine that can execute 10^9 instructions per second, for the following functions $f(N)$: $N^{3/2}$, $N^{5/4}$, $2NH_N$, $N \lg N \lg \lg N$, and $N^2 \lg N$.

2.4 Big-Oh Notation

The mathematical artifact that allows us to suppress detail when we are analyzing algorithms is called the *O-notation*, or “big-Oh notation,” which is defined as follows.

Definition 2.1 A function $g(N)$ is said to be $O(f(N))$ if there exist constants c_0 and N_0 such that $g(N) < c_0 f(N)$ for all $N > N_0$.

We use the *O-notation* for three distinct purposes:

- To bound the error that we make when we ignore small terms in mathematical formulas
- To bound the error that we make when we ignore parts of a program that contribute a small amount to the total being analyzed
- To allow us to classify algorithms according to upper bounds on their total running times

We consider the third use in Section 2.7, and discuss briefly the other two here.

The constants c_0 and N_0 implicit in the *O-notation* often hide implementation details that are important in practice. Obviously, saying that an algorithm has running time $O(f(N))$ says nothing about the running time if N happens to be less than N_0 , and c_0 might be hiding a large amount of overhead designed to avoid a bad worst case. We would prefer an algorithm using N^2 nanoseconds over one using $\log N$ centuries, but we could not make this choice on the basis of the *O-notation*.

Often, the results of a mathematical analysis are not exact, but rather are approximate in a precise technical sense: The result might be an expression consisting of a sequence of decreasing terms. Just as we are most concerned with the inner loop of a program, we are most concerned with the *leading terms* (the largest terms) of a mathematical expression. The O -notation allows us to keep track of the leading terms while ignoring smaller terms when manipulating approximate mathematical expressions, and ultimately allows us to make concise statements that give accurate approximations to the quantities that we analyze.

Some of the basic manipulations that we use when working with expressions containing the O -notation are the subject of Exercises 2.20 through 2.25. Many of these manipulations are intuitive, but mathematically inclined readers may be interested in working Exercise 2.21 to prove the validity of the basic operations from the definition. Essentially, these exercises say that we can expand algebraic expressions using the O -notation as though the O were not there, then can drop all but the largest term. For example, if we expand the expression

$$(N + O(1))(N + O(\log N) + O(1)),$$

we get six terms

$$N^2 + O(N) + O(N \log N) + O(\log N) + O(N) + O(1),$$

but can drop all but the largest O -term, leaving the approximation

$$N^2 + O(N \log N).$$

That is, N^2 is a good approximation to this expression when N is large. These manipulations are intuitive, but the O -notation allows us to express them mathematically with rigor and precision. We refer to a formula with one O -term as an *asymptotic expression*.

For a more relevant example, suppose that (after some mathematical analysis) we determine that a particular algorithm has an inner loop that is iterated $2NH_N$ times on the average, an outer section that is iterated N times, and some initialization code that is executed once. Suppose further that we determine (after careful scrutiny of the implementation) that each iteration of the inner loop requires a_0 nanoseconds, the outer section requires a_1 nanoseconds, and the initialization part a_2 nanoseconds. Then we know that the average running time of

the program (in nanoseconds) is

$$2a_0NH_N + a_1N + a_2.$$

But it is also true that the running time is

$$2a_0NH_N - O(N).$$

This simpler form is significant because it says that, for large N , we may not need to find the values of a_1 or a_2 to approximate the running time. In general, there could well be many other terms in the mathematical expression for the exact running time, some of which may be difficult to analyze. The O -notation provides us with a way to get an approximate answer for large N without bothering with such terms.

Continuing this example, we also can use the O -notation to express running time in terms of a familiar function, $\ln N$. In terms of the O -notation, the approximation in Table 2.3 is expressed as $H_N = \ln N + O(1)$. Thus, $2a_0N\ln N + O(N)$ is an asymptotic expression for the total running time of our algorithm. We expect the running time to be close to the easily computed value $2a_0N\ln N$ for large N . The constant factor a_0 depends on the time taken by the instructions in the inner loop.

Furthermore, we do not need to know the value of a_0 to predict that the running time for input of size $2N$ will be about twice the running time for input of size N for huge N because

$$\frac{2a_0(2N)\ln(2N) + O(2N)}{2a_0N\ln N + O(N)} = \frac{2\ln(2N) + O(1)}{\ln N + O(1)} = 2 + O\left(\frac{1}{\log N}\right).$$

That is, the asymptotic formula allows us to make accurate predictions without concerning ourselves with details of either the implementation or the analysis. Note that such a prediction would *not* be possible if we were to have only an O -approximation for the leading term.

The kind of reasoning just outlined allows us to focus on the leading term when comparing or trying to predict the running times of algorithms. We are so often in the position of counting the number of times that fixed-cost operations are performed and wanting to use the leading term to estimate the result that we normally keep track of *only* the leading term, assuming implicitly that a precise analysis like the one just given could be performed, if necessary.

When a function $f(N)$ is asymptotically large compared to another function $g(N)$ (that is, $g(N)/f(N) \rightarrow 0$ as $N \rightarrow \infty$), we some-

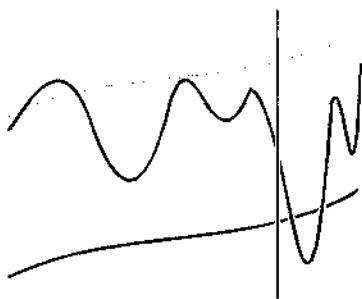


Figure 2.3
Bounding a function with an O -approximation

In this schematic diagram, the oscillating curve represents a function, $g(N)$, which we are trying to approximate; the black smooth curve represents another function, $f(N)$, which we are trying to use for the approximation; and the gray smooth curve represents $cf(N)$ for some unspecified constant c . The vertical line represents a value N_0 , indicating that the approximation is to hold for $N > N_0$. When we say that $g(N) = O(f(N))$, we expect only that the value of $g(N)$ falls below some curve the shape of $f(N)$ to the right of some vertical line. The behavior of $f(N)$ could otherwise be erratic (for example, it need not even be continuous).

times use in this book the (decidedly nontechnical) terminology *about* $f(N)$ to mean $f(N) + O(g(N))$. What we seem to lose in mathematical precision we gain in clarity, for we are more interested in the performance of algorithms than in mathematical details. In such cases, we can rest assured that, for large N (if not for all N), the quantity in question will be close to $f(N)$. For example, even if we know that a quantity is $N(N - 1)/2$, we may refer to it as being about $N^2/2$. This way of expressing the result is more quickly understood than the more detailed exact result, and, for example, deviates from the truth only by 0.1 percent for $N = 1000$. The precision lost in such cases pales by comparison with the precision lost in the more common usage $O(f(N))$. Our goal is to be both precise and concise when describing the performance of algorithms.

In a similar vein, we sometimes say that the running time of an algorithm *is proportional to* $f(N)$ when we can prove that it is equal to $cf(N) + g(N)$ with $g(N)$ asymptotically smaller than $f(N)$. When this kind of bound holds, we can project the running time for, say, $2N$ from our observed running time for N , as in the example just discussed. Figure 2.5 gives the factors that we can use for such projection for functions that commonly arise in the analysis of algorithms. Coupled with empirical studies (see Section 2.1), this approach frees us from the task of determining implementation-dependent constants in detail. Or, working backward, we often can easily develop an hypothesis about the functional growth of the running time of a program by determining the effect of doubling N on running time.

The distinctions among O -bounds, *is proportional to*, and *about* are illustrated in Figures 2.3 and 2.4. We use O -notation primarily to learn the fundamental asymptotic behavior of an algorithm; *is proportional to* when we want to predict performance by extrapolation from empirical studies; and *about* when we want to compare performance or to make absolute performance predictions.

Exercises

- ▷ 2.20 Prove that $O(1)$ is the same as $O(2)$.

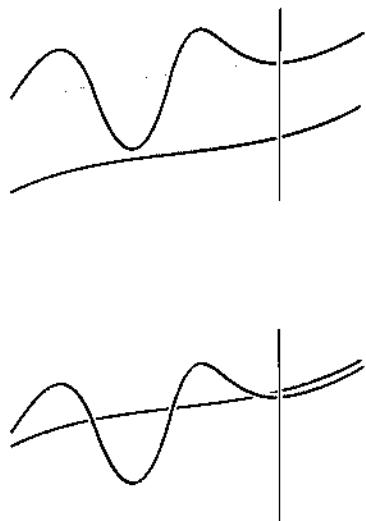


Figure 2.4
Functional approximations

When we say that $g(N)$ is proportional to $f(N)$ (top), we expect that it eventually grows like $f(N)$ does, but perhaps offset by an unknown constant. Given some value of $g(N)$, this knowledge allows us to estimate it for larger N . When we say that $g(N)$ is about $f(N)$ (bottom), we expect that we can eventually use f to estimate the value of g accurately.

2.21 Prove that we can make any of the following transformations in an expression that uses the O -notation:

$$\begin{aligned} f(N) &\rightarrow O(f(N)), \\ cO(f(N)) &\rightarrow O(f(N)), \\ O(cf(N)) &\rightarrow O(f(N)), \\ f(N) - g(N) = O(h(N)) &\rightarrow f(N) = g(N) + O(h(N)), \\ O(f(N))O(g(N)) &\rightarrow O(f(N)g(N)), \\ O(f(N)) + O(g(N)) &\rightarrow O(g(N)) \quad \text{if } f(N) = O(g(N)). \end{aligned}$$

○ **2.22** Show that $(N+1)(H_N + O(1)) = N \ln N + O(N)$.

2.23 Show that $N \ln N = O(N^{3/2})$.

• **2.24** Show that $N^M = O(\alpha^N)$ for any M and any constant $\alpha > 1$.

• **2.25** Prove that

$$\frac{N}{N + O(1)} = 1 + O\left(\frac{1}{N}\right).$$

1	none
$\lg N$	slight increase
N	double
$N \lg N$	slightly more than double
$N^{3/2}$	factor of $2\sqrt{2}$
N^2	factor of 4
N^3	factor of 8
2^N	square

Figure 2.5
Effect of doubling problem size on running time

Predicting the effect of doubling the problem size on the running time is a simple task when the running time is proportional to certain simple functions, as indicated in this table. In theory, we cannot depend on this effect unless N is huge, but this method is surprisingly effective. Conversely, a quick method for determining the functional growth of the running time of a program is to run that program empirically, doubling the input size for N as large as possible, then work backward from this table.

2.26 Suppose that $H_k = N$. Give an approximate formula that expresses k as a function of N .

• **2.27** Suppose that $\lg(k!) = N$. Give an approximate formula that expresses k as a function of N .

○ **2.28** You are given the information that the running time of one algorithm is $O(N \log N)$ and that the running time of another algorithm is $O(N^3)$. What does this statement imply about the relative performance of the algorithms?

○ **2.29** You are given the information that the running time of one algorithm is always about $N \log N$ and that the running time of another algorithm is $O(N^3)$. What does this statement imply about the relative performance of the algorithms?

○ **2.30** You are given the information that the running time of one algorithm is always about $N \log N$ and that the running time of another algorithm is always about N^3 . What does this statement imply about the relative performance of the algorithms?

○ **2.31** You are given the information that the running time of one algorithm is always proportional to $N \log N$ and that the running time of another algorithm is always proportional to N^3 . What does this statement imply about the relative performance of the algorithms?

○ **2.32** Derive the factors given in Figure 2.5: For each function $f(N)$ that appears on the left, find an asymptotic formula for $f(2N)/f(N)$.

2.5 Basic Recurrences

As we shall see throughout the book, a great many algorithms are based on the principle of recursively decomposing a large problem into one or more smaller ones, using solutions to the subproblems to solve the original problem. We discuss this topic in detail in Chapter 5, primarily from a practical point of view, concentrating on implementations and applications. We also consider an example in detail in Section 2.6. In this section, we look at basic methods for analyzing such algorithms and derive solutions to a few standard formulas that arise in the analysis of many of the algorithms that we will be studying. Understanding the mathematical properties of the formulas in this section will give us insight into the performance properties of algorithms throughout the book.

Formula 2.1 This formula arises for a program that loops through the input to eliminate one item:

$$C_N = C_{N-1} + N, \quad \text{for } N \geq 2 \text{ with } C_1 = 1.$$

Solution: C_N is about $N^2/2$. To find the value of C_N , we *telescope* the equation by applying it to itself, as follows:

$$\begin{aligned} C_N &= C_{N-1} + N \\ &= C_{N-2} + (N-1) + N \\ &= C_{N-3} + (N-2) + (N-1) + N \\ &\vdots \\ &= C_1 + 2 + \cdots + (N-2) + (N-1) + N \\ &= 1 + 2 + \cdots + (N-2) + (N-1) + N \\ &= \frac{N(N+1)}{2}. \end{aligned}$$

Evaluating the sum $1 + 2 + \cdots + (N-2) + (N-1) + N$ is elementary: The given result follows when we add the sum to itself, but in reverse order, term by term. This result—twice the value sought—consists of N terms, each of which sums to $N+1$.

This simple example illustrates the basic scheme that we use in this section as we consider a number of formulas, which are all based on the principle that recursive decomposition in an algorithm is directly reflected in its analysis. For example, the running time of such

algorithms is determined by the size and number of the subproblems and the time required for the decomposition. Mathematically, the dependence of the running time of an algorithm for an input of size N on its running time for smaller inputs is captured easily with formulas called *recurrence relations*. Such formulas describe precisely the performance of the corresponding algorithms: To derive the running time, we solve the recurrences. More rigorous arguments related to specific algorithms will come up when we get to the algorithms—here, we concentrate on the formulas themselves.

Formula 2.2 This recurrence arises for a recursive program that halves the input in one step:

N	$(N)_2$	$\lfloor \lg N \rfloor + 1$
1	1	1
2	10	2
3	11	2
4	100	3
5	101	3
6	110	3
7	111	3
8	1000	4
9	1001	4
10	1010	4
11	1011	4
12	1100	4
13	1101	4
14	1110	4
15	1111	4

Figure 2.6
Integer functions and binary representations

Given the binary representation of a number N (center), we obtain $\lfloor N/2 \rfloor$ by removing the rightmost bit. That is, the number of bits in the binary representation of N is 1 greater than the number of bits in the binary representation of $\lfloor N/2 \rfloor$. Therefore, $\lfloor \lg N \rfloor + 1$, the number of bits in the binary representation of N , is the solution to Formula 2.2 for the case that $N/2$ is interpreted as $\lfloor N/2 \rfloor$.

Solution: C_N is about $\lg N$. As written, this equation is meaningless unless N is even or we assume that $N/2$ is an integer division. For the moment, we assume that $N = 2^n$, so the recurrence is always well-defined. (Note that $n = \lg N$.) But then the recurrence telescopes even more easily than our first recurrence:

$$\begin{aligned} C_{2^n} &= C_{2^{n-1}} + 1 \\ &= C_{2^{n-2}} + 1 + 1 \\ &= C_{2^{n-3}} + 3 \\ &\vdots \\ &= C_2 + n \\ &= n + 1. \end{aligned}$$

The precise solution for general N depends on the interpretation of $N/2$. In the case that $N/2$ represents $\lfloor N/2 \rfloor$, we have a simple solution: C_N is the number of bits in the binary representation of N , and that number is $\lfloor \lg N \rfloor + 1$, by definition. This conclusion follows immediately from the fact that the operation of eliminating the rightmost bit of the binary representation of any integer $N > 0$ converts it into $\lfloor N/2 \rfloor$ (see Figure 2.6).

Formula 2.3 This recurrence arises for a recursive program that halves the input, but perhaps must examine every item in the input.

$$C_N = C_{N/2} + N. \quad \text{for } N \geq 2 \text{ with } C_1 = 0.$$

Solution: C_N is about $2N$. The recurrence telescopes to the sum $N + N/2 + N/4 + N/8 + \dots$ (Like Formula 2.2, the recurrence is precisely defined only when N is a power of 2). If the sequence is infinite, this simple geometric sum evaluates to exactly $2N$. Because we use integer division and stop at 1, this value is an approximation to the exact answer. The precise solution involves properties of the binary representation of N .

Formula 2.4 This recurrence arises for a recursive program that has to make a linear pass through the input, before, during, or after splitting that input into two halves:

$$C_N = 2C_{N/2} + N, \quad \text{for } N \geq 2 \text{ with } C_1 = 0.$$

Solution: C_N is about $N \lg N$. This solution is the most widely cited of those we are considering here, because the recurrence applies to a family of standard divide-and-conquer algorithms.

$$\begin{aligned} C_{2^n} &= 2C_{2^{n-1}} + 2^n \\ \frac{C_{2^n}}{2^n} &= \frac{C_{2^{n-1}}}{2^{n-1}} + 1 \\ &= \frac{C_{2^{n-2}}}{2^{n-2}} + 1 + 1 \\ &\vdots \\ &= n. \end{aligned}$$

We develop the solution very much as we did in Formula 2.2, but with the additional trick of dividing both sides of the recurrence by 2^n at the second step to make the recurrence telescope.

Formula 2.5 This recurrence arises for a recursive program that splits the input into two halves and then does a constant amount of other work (see Chapter 5).

$$C_N = 2C_{N/2} + 1, \quad \text{for } N \geq 2 \text{ with } C_1 = 1.$$

Solution: C_N is about $2N$. We can derive this solution in the same manner as we did the solution to Formula 2.4.

We can solve minor variants of these formulas, involving different initial conditions or slight differences in the additive term, using the same solution techniques, although we need to be aware that some recurrences that seem similar to these may actually be rather difficult

to solve. There is a variety of advanced general techniques for dealing with such equations with mathematical rigor (*see reference section*). We will encounter a few more complicated recurrences in later chapters, but we defer discussion of their solution until they arise.

Exercises

▷ 2.33 Give a table of the values of C_N in Formula 2.2 for $1 \leq N \leq 32$, interpreting $N/2$ to mean $[N/2]$.

▷ 2.34 Answer Exercise 2.33, but interpret $N/2$ to mean $[N/2]$.

▷ 2.35 Answer Exercise 2.34 for Formula 2.3.

○ 2.36 Suppose that f_N is proportional to a constant and that

$$C_N = C_{N/2} + f_N, \quad \text{for } N \geq t \text{ with } 0 \leq C_N < c \text{ for } N < t,$$

where c and t are both constants. Show that C_N is proportional to $\lg N$.

● 2.37 State and prove generalized versions of Formulas 2.3 through 2.5 that are analogous to the generalized version of Formula 2.2 in Exercise 2.36.

2.38 Give a table of the values of C_N in Formula 2.4 for $1 \leq N \leq 32$, for the following three cases: (i) interpret $N/2$ to mean $[N/2]$; (ii) interpret $N/2$ to mean $[N/2]$; (iii) interpret $2C_{N/2}$ to mean $C_{\lceil N/2 \rceil} + C_{\lceil N/2 \rceil}$.

2.39 Solve Formula 2.4 for the case when $N/2$ is interpreted as $[N/2]$, by using a correspondence to the binary representation of N , as in the proof of Formula 2.2. *Hint:* Consider all the numbers less than N .

2.40 Solve the recurrence

$$C_N = C_{N/2} + N^2, \quad \text{for } N \geq 2 \text{ with } C_1 = 0,$$

when N is a power of 2.

2.41 Solve the recurrence

$$C_N = C_{N^\alpha} + 1, \quad \text{for } N \geq 2 \text{ with } C_1 = 0,$$

when N is a power of α .

○ 2.42 Solve the recurrence

$$C_N = \alpha C_{N/2}, \quad \text{for } N \geq 2 \text{ with } C_1 = 1,$$

when N is a power of 2.

○ 2.43 Solve the recurrence

$$C_N = (C_{N/2})^2, \quad \text{for } N \geq 2 \text{ with } C_1 = 1,$$

when N is a power of 2.

• 2.44 Solve the recurrence

$$C_N = \left(2 + \frac{1}{\lg N}\right)C_{N/2}, \quad \text{for } N \geq 2 \text{ with } C_1 = 1,$$

when N is a power of 2.

• 2.45 Consider the family of recurrences like Formula 2.1, where we allow $N/2$ to be interpreted as $\lfloor N/2 \rfloor$ or $\lceil N/2 \rceil$, and we require only that the recurrence hold for $N > c_0$ with $C_N = O(1)$ for $N \leq c_0$. Prove that $\lg N + O(1)$ is the solution to all such recurrences.

•• 2.46 Develop generalized recurrences and solutions similar to Exercise 2.45 for Formulas 2.2 through 2.5.

2.6 Examples of Algorithm Analysis

Armed with the tools outlined in the previous three sections, we now consider the analysis of *sequential search* and *binary search*, two basic algorithms for determining whether or not any of a sequence of objects appears among a set of previously stored objects. Our purpose is to illustrate the manner in which we will compare algorithms, rather than to describe these particular algorithms in detail. For simplicity, we assume here that the objects in question are integers. We will consider more general applications in great detail in Chapters 12 through 16. The simple versions of the algorithms that we consider here not only expose many aspects of the algorithm design and analysis problem, but also have many direct applications.

For example, we might imagine a credit-card company that has N credit risks or stolen credit cards, and that wants to check whether any of M given transactions involves any one of the N bad numbers. To be concrete, we might think of N being large (say on the order of 10^3 to 10^6) and M being huge (say on the order of 10^6 to 10^9) for this application. The goal of the analysis is to be able to estimate the running times of the algorithms when the values of the parameters fall within these ranges.

Program 2.1 implements a straightforward solution to the search problem. It is packaged as a C function that operates on an array (see Chapter 3) for better compatibility with other code that we will examine for the same problem in Part 4, but it is not necessary to understand the details of the packaging to understand the algorithm: We store all the objects in an array; then, for each transaction, we look

through the array sequentially, from beginning to end, checking each to see whether it is the one that we seek.

To analyze the algorithm, we note immediately that the running time depends on whether or not the object sought is in the array. We can determine that the search is unsuccessful only by examining each of the N objects, but a search could end successfully at the first, second, or any one of the objects.

Therefore, the running time depends on the data. If all the searches are for the number that happens to be in the first position in the array, then the algorithm will be fast; if they are for the number that happens to be in the last position in the array, it will be slow. We discuss in Section 2.7 the distinction between being able to *guarantee* performance and being able to *predict* performance. In this case, the best guarantee that we can provide is that no more than N numbers will be examined.

To make a prediction, however, we need to make an assumption about the data. In this case, we might choose to assume that all the numbers are randomly chosen. This assumption implies, for example, that each number in the table is equally likely to be the object of a search. On reflection, we realize that it is that property of the search that is critical, because with randomly chosen numbers we would be unlikely to have a successful search at all (see Exercise 2.48). For some applications, the number of transactions that involve a successful search might be high; for other applications, it might be low. To avoid confusing the model with properties of the application, we separate the two cases (successful and unsuccessful) and analyze them independently. This example illustrates that a critical part of an effective analysis is the development of a reasonable model for the application at hand. Our analytic results will depend on the proportion of searches that are successful; indeed, it will give us information that we might need if we are to choose different algorithms for different applications based on this parameter.

Property 2.1 *Sequential search examines N numbers for each unsuccessful search and about $N/2$ numbers for each successful search on the average.*

If each number in the table is equally likely to be the object of a search, then

$$(1 + 2 + \dots + N)/N = (N + 1)/2$$

Program 2.1 Sequential search

This function checks whether the number v is among a previously stored set of numbers in $a[1]$, $a[1+1]$, ..., $a[r]$, by comparing against each number sequentially, starting at the beginning. If we reach the end without finding the number sought, then we return the value -1 . Otherwise, we return the index of the array position containing the number.

```
int search(int a[], int v, int l, int r)
{ int i;
  for (i = l; i <= r; i++)
    if (v == a[i]) return i;
  return -1;
}
```

is the average cost of a search. ■

Property 2.1 implies that the running time of Program 2.1 is proportional to N , subject to the implicit assumption that the average cost of comparing two numbers is constant. Thus, for example, we can expect that, if we double the number of objects, we double the amount of time required for a search.

We can speed up sequential search for unsuccessful search by putting the numbers in the table in order. Sorting the numbers in the table is the subject of Chapters 6 through 11. A number of the algorithms that we will consider get that task done in time proportional to $N \log N$, which is insignificant by comparison to the search costs when M is huge. In an ordered table, we can terminate the search immediately on reaching a number that is larger than the one that we seek. This change reduces the cost of sequential search to about $N/2$ numbers examined for unsuccessful search, the same as for successful search.

Property 2.2 *Sequential search in an ordered table examines N numbers for each search in the worst case and about $N/2$ numbers for each search on the average.*

We still need to specify a model for unsuccessful search. This result follows from assuming that the search is equally likely to terminate at any one of the $N + 1$ intervals defined by the N numbers in the table,

Program 2.2 Binary search

This program has the same functionality as Program 2.1, but it is much more efficient.

```
int search(int a[], int v, int l, int r)
{
    while (r >= l)
        { int m = (l+r)/2;
          if (v == a[m]) return m;
          if (v < a[m]) r = m-1; else l = m+1;
        }
    return -1;
}
```

which leads immediately to the expression

$$(1 + 2 + \dots + N + N)/N = (N + 3)/2.$$

The cost of an unsuccessful search ending before or after the N th entry in the table is the same: N . ■

Another way to state the result of Property 2.2 is to say that the running time of sequential search is proportional to MN for M transactions, on the average and in the worst case. If we double either the number of transactions or the number of objects in the table, we can expect the running time to double; if we double both, we can expect the running time to go up by a factor of 4. The result also tells us that the method is not suitable for huge tables. If it takes c microseconds to examine a single number, then, for $M = 10^9$ and $N = 10^6$, the running time for all the transactions would be at least $(c/2)10^9$ seconds, or, by Figure 2.1, about 16c years, which is prohibitive.

Program 2.2 is a classical solution to the search problem that is much more efficient than sequential search. It is based on the idea that, if the numbers in the table are in order, we can eliminate half of them from consideration by comparing the one that we seek with the one at the middle position in the table. If it is equal, we have a successful search. If it is less, we apply the same method to the left half of the table. If it is greater, we apply the same method to the right half of the

table. Figure 2.7 is an example of the operation of this method on a sample set of numbers.

Property 2.3 *Binary search never examines more than $\lfloor \lg N \rfloor + 1$ numbers.*

The proof of this property illustrates the use of recurrence relations in the analysis of algorithms. If we let T_N represent the number of comparisons required for binary search in the worst case, then the way in which the algorithm reduces search in a table of size N to search in a table half the size immediately implies that

$$T_N \leq T_{\lfloor N/2 \rfloor} + 1, \quad \text{for } N \geq 2 \text{ with } T_1 = 1.$$

To search in a table of size N , we examine the middle number, then search in a table of size no larger than $\lfloor N/2 \rfloor$. The actual cost could be less than this value because the comparison might cause us to terminate a successful search, or because the table to be searched might be of size $\lfloor N/2 \rfloor - 1$ (if N is even). As we did in the solution of Formula 2.2, we can prove immediately that $T_N \leq n + 1$ if $N = 2^n$ and then verify the general result by induction. ■

Property 2.3 allows us to solve a huge search problem with up to 1 million numbers with at most 20 comparisons per transaction, and that is likely to be less than the time it takes to read or write the number on many computers. The search problem is so important that several methods have been developed that are even faster than this one, as we shall see in Chapters 12 through 16.

Note that we express Property 2.1 and Property 2.2 in terms of the operations that we perform most often on the data. As we noted in the commentary following Property 2.1, we expect that each operation should take a constant amount of time, and we can conclude that the running time of binary search is proportional to $\lg N$ as compared to N for sequential search. As we double N , the running time of binary search hardly changes, but the running time of sequential search doubles. As N grows, the gap between the two methods becomes a chasm.

We can verify the analytic evidence of Properties 2.1 and 2.2 by implementing and testing the algorithms. For example, Table 2.4 shows running times for binary search and sequential search for M searches in a table of size N (including, for binary search, the cost of

1488	1488
1578	1578
1973	1973
3665	3665
4426	4426
4548	4548
5435	5435
5446	5446
6333	6333
6385	6385
6455	6455
6504	
6937	
6965	
7104	
7230	
8340	
8958	
9208	
9364	
9550	
9645	
9686	

Figure 2.7
Binary search

To see whether or not 5025 is in the table of numbers in the left column, we first compare it with 6504; that leads us to consider the first half of the array. Then we compare against 4548 (the middle of the first half); that leads us to the second half of the first half. We continue, always working on a subarray that would contain the number being sought, if it is in the table. Eventually, we get a subarray with just 1 element, which is not equal to 5025, so 5025 is not in the table.

Table 2.4 Empirical study of sequential and binary search

These relative timings validate our analytic results that sequential search takes time proportional to MN and binary search takes time proportional to $M \lg N$ for M searches in a table of N objects. When we increase N by a factor of 2, the time for sequential search increases by a factor of 2 as well, but the time for binary search hardly changes. Sequential search is infeasible for huge M as N increases, but binary search is fast even for huge tables.

N	$M = 1000$		$M = 10000$		$M = 100000$	
	S	B	S	B	S	B
125	1	1	13	2	130	20
250	3	0	25	2	251	22
500	5	0	49	3	492	23
1250	13	0	128	3	1276	25
2500	26	1	267	3		28
5000	53	0	533	3		30
12500	134	1	1337	3		33
25000	268	1		3		35
50000	537	0		4		39
100000	1269	1		5		47

Key:

S sequential search (Program 2.1)

B binary search (Program 2.2)

sorting the table) for various values of M and N . We will not consider the implementation of the program to run these experiments in detail here because it is similar to those that we consider in full detail in Chapters 6 and 11, and because we consider the use of library and external functions and other details of putting together programs from constituent pieces, including the `sort` function, in Chapter 3. For the moment, we simply stress that doing empirical testing is an integral part of evaluating the efficiency of an algorithm.

Table 2.4 validates our observation that the functional growth of the running time allows us to predict performance for huge cases

on the basis of empirical studies for small cases. The combination of mathematical analysis and empirical studies provides persuasive evidence that binary search is the preferred algorithm, by far.

This example is a prototype of our general approach to comparing algorithms. We use mathematical analysis of the frequency with which algorithms perform critical abstract operations, then use those results to deduce the functional form of the running time, which allows us to verify and extend empirical studies. As we develop algorithmic solutions to computational problems that are more and more refined, and as we develop mathematical analyses to learn their performance characteristics that are more and more refined, we call on mathematical studies from the literature, so as to keep our attention on the algorithms themselves in this book. We cannot do thorough mathematical and empirical studies of every algorithm that we encounter, but we strive to identify essential performance characteristics, knowing that, in principle, we can develop a scientific basis for making informed choices among algorithms in critical applications.

Exercises

- ▷ 2.47 Give the average number of comparisons used by Program 2.1 in the case that αN of the searches are successful, for $0 \leq \alpha \leq 1$.
- 2.48 Estimate the probability that at least one of M random 10-digit numbers matches one of a set of N given values, for $M = 10, 100$, and 1000 and $N = 10^3, 10^4, 10^5$, and 10^6 .
- 2.49 Write a driver program that generates M random integers and puts them in an array, then counts the number of N random integers that matches one of the numbers in the array, using sequential search. Run your program for $M = 10, 100$, and 1000 and $N = 10, 100$, and 1000 .
- 2.50 State and prove a property analogous to Property 2.3 for binary search.

2.7 Guarantees, Predictions, and Limitations

The running time of most algorithms depends on their input data. Typically, our goal in the analysis of algorithms is somehow to eliminate that dependence: We want to be able to say something about the performance of our programs that depends on the input data to as little an extent as possible, because we generally do not know what the input data will be each time the program is invoked. The examples

in Section 2.6 illustrate the two major approaches that we use toward this end: worst-case analysis and average-case analysis.

Studying the *worst-case* performance of algorithms is attractive because it allows us to make *guarantees* about the running time of programs. We say that the **number** of times certain abstract operations are executed is less than a certain function of the number of inputs, no matter what the input values are. For example, Property 2.3 is an example of such a guarantee for binary search, as is Property 1.3 for weighted quick union. If the guarantees are low, as is the case with binary search, then we are in a favorable situation, because we have eliminated cases for which our program might run slowly. Programs with good worst-case performance characteristics are a basic goal in algorithm design.

There are several difficulties with worst-case analysis, however. For a given algorithm, there might be a significant gap between the time required for it to solve a worst-case instance of the input and the time required for it to solve the data that it might encounter in practice. For example, quick union requires time proportional to N in the worst case, but only $\log N$ for typical data. More important, we cannot always prove that there is an input for which the running time of an algorithm achieves a certain bound; we can prove only that it is guaranteed to be lower than the bound. Moreover, for some problems, algorithms with good worst-case performance are significantly more complicated than are other algorithms. We often find ourselves in the position of having an algorithm with good worst-case performance that is slower than simpler algorithms for the data that occur in practice, or that is not sufficiently faster that the extra effort required to achieve good worst-case performance is justified. For many applications, other considerations—such as portability or reliability—are more important than improved worst-case performance guarantees. For example, as we saw in Chapter 1, weighted quick union with path compression provides provably better performance guarantees than weighted quick union, but the algorithms have about the same running time for typical practical data.

Studying the *average-case* performance of algorithms is attractive because it allows us to make *predictions* about the running time of programs. In the simplest situation, we can characterize precisely the inputs to the algorithm; for example, a sorting algorithm might

operate on an array of N random integers, or a geometric algorithm might process a set of N random points in the plane with coordinates between 0 and 1. Then, we calculate the average number of times that each instruction is executed, and calculate the average running time of the program by multiplying each instruction frequency by the time required for the instruction and adding them all together.

There are also several difficulties with average-case analysis, however. First, the input model may not accurately characterize the inputs encountered in practice, or there may be no natural input model at all. Few people would argue against the use of input models such as “randomly ordered file” for a sorting algorithm, or “random point set” for a geometric algorithm, and for such models it is possible to derive mathematical results that can predict accurately the performance of programs running on actual applications. But how should one characterize the input to a program that processes English-language text? Even for sorting algorithms, models other than randomly ordered inputs are of interest in certain applications. Second, the analysis might require deep mathematical reasoning. For example, the average-case analysis of union-find algorithms is difficult. Although the derivation of such results is normally beyond the scope of this book, we will illustrate their nature with a number of classical examples, and we will cite relevant results when appropriate (fortunately, many of our best algorithms have been analyzed in the research literature). Third, knowing the average value of the running time might not be sufficient: we may need to know the standard deviation or other facts about the distribution of the running time, which may be even more difficult to derive. In particular, we are often interested in knowing the chance that the algorithm could be dramatically slower than expected.

In many cases, we can answer the first objection listed in the previous paragraph by turning randomness to our advantage. For example, if we randomly scramble an array before attempting to sort it, then the assumption that the elements in the array are in random order is accurate. For such algorithms, which are called *randomized algorithms*, the average-case analysis leads to predictions of the expected running time in a strict probabilistic sense. Moreover, we are often able to prove that the probability that such an algorithm will be slow is negligibly small. Examples of such algorithms include quicksort

(see Chapter 9), randomized BSTs (see Chapter 13), and hashing (see Chapter 14).

The field of *computational complexity* is the branch of analysis of algorithms that helps us to understand the fundamental *limitations* that we can expect to encounter when designing algorithms. The overall goal is to determine the worst-case running time of the *best* algorithm to solve a given problem, to within a constant factor. This function is called the *complexity* of the problem.

Worst-case analysis using the O -notation frees the analyst from considering the details of particular machine characteristics. The statement that the running time of an algorithm is $O(f(N))$ is independent of the input and is a useful way to categorize algorithms in a way that is independent of both inputs and implementation details, separating the analysis of an algorithm from any particular implementation. We ignore constant factors in the analysis; in most cases, if we want to know whether the running time of an algorithm is proportional to N or proportional to $\log N$, it does not matter whether the algorithm is to be run on a nanocomputer or on a supercomputer, and it does not matter whether the inner loop has been implemented carefully with only a few instructions or badly implemented with many instructions.

When we can prove that the worst-case running time of an algorithm to solve a certain problem is $O(f(N))$, we say that $f(N)$ is an *upper bound* on the complexity of the problem. In other words, the running time of the best algorithm to solve a problem is no higher than the running time of any particular algorithm to solve the problem.

We constantly strive to improve our algorithms, but we eventually reach a point where no change seems to improve the running time. For every given problem, we are interested in knowing when to stop trying to find improved algorithms, so we seek *lower bounds* on the complexity. For many problems, we can prove that *any* algorithm to solve the problem must use a certain number of fundamental operations. Proving lower bounds is a difficult matter of carefully constructing a machine model and then developing intricate theoretical constructions of inputs that are difficult for any algorithm to solve. We rarely touch on the subject of proving lower bounds, but they represent computational barriers that guide us in the design of algorithms, so we maintain awareness of them when they are relevant.

When complexity studies show that the upper bound of an algorithm matches the lower bound, then we have some confidence that it is fruitless to try to design an algorithm that is fundamentally faster than the best known, and we can start to concentrate on the implementation. For example, binary search is optimal, in the sense that no algorithm that uses comparisons exclusively can use fewer comparisons in the worst case than binary search.

We also have matching upper and lower bounds for pointer-based union-find algorithms. Tarjan showed in 1975 that weighted quick union with path compression requires following less than $O(\lg^* V)$ pointers in the worst case, and that any pointer-based algorithm must follow more than a constant number of pointers in the worst case for some input. In other words, there is no point looking for some new improvement that will guarantee to solve the problem with a linear number of $i = a[i]$ operations. In practical terms, this difference is hardly significant, because $\lg^* V$ is so small; still, finding a simple linear algorithm for this problem was a research goal for many years, and Tarjan's lower bound has allowed researchers to move on to other problems. Moreover, the story shows that there is no avoiding functions like the rather complicated \log^* function, because such functions are intrinsic to this problem.

Many of the algorithms in this book have been subjected to detailed mathematical analyses and performance studies far too complex to be discussed here. Indeed, it is on the basis of such studies that we are able to recommend many of the algorithms that we discuss.

Not all algorithms are worthy of such intense scrutiny; indeed, during the design process, it is preferable to work with approximate performance indicators to guide the design process without extraneous detail. As the design becomes more refined, so must the analysis, and more sophisticated mathematical tools need to be applied. Often, the design process leads to detailed complexity studies that lead to theoretical algorithms that are rather far from any particular application. It is a common mistake to assume that rough analyses from complexity studies will translate immediately into efficient practical algorithms; such assumptions can lead to unpleasant surprises. On the other hand, computational complexity is a powerful tool that tells us when we have reached performance limits in our design work and that

can suggest departures in design in pursuit of closing the gap between upper and lower bounds.

In this book, we take the view that algorithm design, careful implementation, mathematical analysis, theoretical studies, and empirical analysis all contribute in important ways to the development of elegant and efficient programs. We want to gain information about the properties of our programs using any tools at our disposal, then to modify or develop new programs on the basis of that information. We will not be able to do exhaustive testing and analysis of every algorithm that we run in every programming environment on every machine, but we can use careful implementations of algorithms that we know to be efficient, then refine and compare them when peak performance is necessary. Throughout the book, when appropriate, we shall consider the most important methods in sufficient detail to appreciate why they perform well.

Exercise

- 2.51 You are given the information that the time complexity of one problem is $N \log N$ and that the time complexity of another problem is N^3 . What does this statement imply about the relative performance of specific algorithms that solve the problems?

References for Part One

There are a large number of introductory textbooks on programming. Still, the best source for specific facts about C and examples of C programs, in the same spirit as those found in this book, is Kernighan and Ritchie's book on the language.

The many variants on algorithms for the union-find problem of Chapter 1 are ably categorized and compared by van Leeuwen and Tarjan.

Bentley's books describe, again in the same spirit as much of the material here, a number of detailed case studies on evaluating various approaches to developing algorithms and implementations for solving numerous interesting problems.

The classic reference on the analysis of algorithms based on asymptotic worst-case performance measures is Aho, Hopcroft, and Ullman's book. Knuth's books cover average-case analysis more fully and are the authoritative source on specific properties of numerous algorithms. The books by Gonnet and Baeza-Yates and by Cormen, Leiserson, and Rivest are more recent works; both include extensive references to the research literature.

The book by Graham, Knuth and Patashnik covers the type of mathematics that commonly arises in the analysis of algorithms, and such material is also sprinkled liberally throughout Knuth's books. The book by Sedgewick and Flajolet is a thorough introduction to the subject.

- A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1975.
- J. L. Bentley, *Programming Pearls*, Addison-Wesley, Reading, MA, 1985; *More Programming Pearls*, Addison-Wesley, Reading, MA, 1988.
- R. Baeza-Yates and G. H. Gonnet, *Handbook of Algorithms and Data Structures*, second edition, Addison-Wesley, Reading, MA, 1984.
- T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press/McGraw-Hill, Cambridge, MA, 1990.
- R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, MA, 1988.

- B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, second edition, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, second edition, Addison-Wesley, Reading, MA, 1973; *Volume 2: Seminumerical Algorithms*, second edition, Addison-Wesley, Reading, MA, 1981; *Volume 3: Sorting and Searching*, second printing, Addison-Wesley, Reading, MA, 1975.
- R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1996.
- J. van Leeuwen and R. E. Tarjan, "Worst-case analysis of set-union algorithms," *Journal of the ACM*, 1984.

P A R T
T W O

Data Structures



CHAPTER THREE

Elementary Data Structures

ORGANIZING THE DATA for processing is an essential step in the development of a computer program. For many applications, the choice of the proper data structure is the only major decision involved in the implementation: once the choice has been made, the necessary algorithms are simple. For the same data, some data structures require more or less space than others; for the same operations on the data, some data structures lead to more or less efficient algorithms than others. The choices of algorithm and of data structure are closely intertwined, and we continually seek ways to save time or space by making the choice properly.

A data structure is not a passive object: We also must consider the operations to be performed on it (and the algorithms used for these operations). This concept is formalized in the notion of a *data type*. In this chapter, our primary interest is in concrete implementations of the fundamental approaches that we use to structure data. We consider basic methods of organization and methods for manipulating data, work through a number of specific examples that illustrate the benefits of each, and discuss related issues such as storage management. In Chapter 4, we discuss *abstract data types*, where we separate the definitions of data types from implementations.

We discuss properties of arrays, linked lists, and strings. These classical data structures have widespread applicability: with trees (see Chapter 5), they form the basis for virtually all the algorithms considered in this book. We consider various primitive operations for manipulating these data structures, to develop a basic set of tools that we can use to develop sophisticated algorithms for difficult problems.

The study of storing data as variable-sized objects and in linked data structures requires an understanding of how the system manages the storage that it allocates to programs for their data. We do not cover this subject exhaustively because many of the important considerations are system and machine dependent. However, we do discuss approaches to storage management and several basic underlying mechanisms. Also, we discuss the specific (stylized) manners in which we will be using C storage-allocation mechanisms in our programs.

At the end of the chapter, we consider several examples of *compound structures*, such as arrays of linked lists and arrays of arrays. The notion of building abstract mechanisms of increasing complexity from lower-level ones is a recurring theme throughout this book. We consider a number of examples that serve as the basis for more advanced algorithms later in the book.

The data structures that we consider in this chapter are important building blocks that we can use in a natural manner in C and many other programming languages. In Chapter 5, we consider another important data structure, the *tree*. Arrays, strings, linked lists, and trees are the basic elements underlying most of the algorithms that we consider in this book. In Chapter 4, we discuss the use of the concrete representations developed here in building basic abstract data types that can meet the needs of a variety of applications. In the rest of the book, we develop numerous variations of the basic tools discussed here, trees, and abstract data types, to create algorithms that can solve more difficult problems and that can serve us well as the basis for higher-level abstract data types in diverse applications.

3.1 Building Blocks

In this section, we review the primary low-level constructs that we use to store and process information in C. All the data that we process on a computer ultimately decompose into individual bits, but writing programs that exclusively process bits would be tiresome indeed. *Types* allow us to specify how we will use particular sets of bits and *functions* allow us to specify the operations that we will perform on the data. We use C *structures* to group together heterogeneous pieces of information, and we use *pointers* to refer to information indirectly. In this section, we consider these basic C mechanisms, in the context of presenting a

general approach to organizing our programs. Our primary goal is to lay the groundwork for the development, in the rest of the chapter and in Chapters 4 and 5, of the higher-level constructs that will serve as the basis for most of the algorithms that we consider in this book.

We write programs that process information derived from mathematical or natural-language descriptions of the world in which we live; accordingly, computing environments need to provide built-in support for the basic building blocks of such descriptions—numbers and characters. In C, our programs are all built from just a few basic types of data:

- Integers (`ints`).
- Floating-point numbers (`floats`).
- Characters (`chars`).

It is customary to refer to these basic types by their C names—`int`, `float`, and `char`—although we often use the generic terminology *integer*, *floating-point number*, and *character*, as well. Characters are most often used in higher-level abstractions—for example to make words and sentences—so we defer consideration of character data to Section 3.6 and look at numbers here.

We use a fixed number of bits to represent numbers, so `ints` are by necessity integers that fall within a specific range that depends on the number of bits that we use to represent them. Floating-point numbers approximate real numbers, and the number of bits that we use to represent them affects the precision with which we can approximate a real number. In C, we trade space for accuracy by choosing from among the types `int`, `long int`, or `short int` for integers and from among `float` or `double` for floating-point numbers. On most systems, these types correspond to underlying hardware representations. The number of bits used for the representation, and therefore the range of values (in the case of `ints`) or precision (in the case of `floats`), is machine-dependent (see Exercise 3.1), although C provides certain guarantees. In this book, for clarity, we normally use `int` and `float`, except in cases where we want to emphasize that we are working with problems where big numbers are needed.

In modern programming, we think of the type of the data more in terms of the needs of the program than the capabilities of the machine, primarily, in order to make programs portable. Thus, for example, we think of a `short int` as an object that can take on values between

–32,768 and 32,767, instead of as a 16-bit object. Moreover, our concept of an integer includes the operations that we perform on them: addition, multiplication, and so forth.

Definition 3.1 *A data type is a set of values and a collection of operations on those values.*

Operations are associated with types, not the other way around. When we perform an operation, we need to ensure that its operands and result are of the correct type. Neglecting this responsibility is a common programming error. In some situations, C performs implicit type conversions; in other situations, we use *casts*, or explicit type conversions. For example, if *x* and *N* are integers, the expression

`((float) x) / N`

includes both types of conversion: the `(float)` is a cast that converts the value of *x* to floating point; then an implicit conversion is performed for *N* to make both arguments of the divide operator floating point, according to C's rules for implicit type conversion.

Many of the operations associated with standard data types (for example, the arithmetic operations) are built into the C language. Other operations are found in the form of functions that are defined in standard function libraries; still others take form in the C functions that we define in our programs (see Program 3.1). That is, the concept of a data type is relevant not just to integer, floating point, and character built-in types. We often define our own data types, as an effective way of organizing our software. When we define a simple function in C, we are effectively creating a new data type, with the operation implemented by that function added to the operations defined for the types of data represented by its arguments. Indeed, in a sense, *each* C program is a data type—a list of sets of values (built-in or other types) and associated operations (functions). This point of view is perhaps too broad to be useful, but we shall see that narrowing our focus to understand our programs in terms of data types is valuable.

One goal that we have when writing programs is to organize them such that they apply to as broad a variety of situations as possible. The reason for adopting such a goal is that it might put us in the position of being able to reuse an old program to solve a new problem, perhaps completely unrelated to the problem that the program was originally intended to solve. First, by taking care to understand and to specify

Program 3.1 Function definition

The mechanism that we use in C to implement new operations on data is the *function definition*, illustrated here.

All functions have a list of *arguments* and possibly a *return value*. The function `lg` here has one argument and a return value, each of type `int`. The function `main` has neither arguments nor return value.

We *declare* the function by giving its name and the types of its return values. The first line of code here references a system file that contains declarations of system functions such as `printf`. The second line of code is a declaration for `lg`. The declaration is optional if the function is defined (see next paragraph) before it is used, as is the case with `main`. The declaration provides the information necessary for other functions to *call* or *invoke* the function, using arguments of the proper type. The calling function can use the function in an expression, in the same way as it uses variables of the return-value type.

We *define* functions with C code. All C programs include a definition of the function `main`, and this code also defines `lg`. In a function definition, we give names to the arguments (which we refer to as *parameters*) and express the computation in terms of those names, as if they were local variables. When the function is invoked, these variables are initialized with the values of the arguments and the function code is executed. The `return` statement is the instruction to end execution of the function and provide the return value to the calling function. In principle, the calling function is not to be otherwise affected, though we shall see many exceptions to this principle.

The separation of definition and declaration provides flexibility in organizing programs. For example, both could be in separate files (see *text*). Or, in a simple program like this one, we could put the definition of `lg` before the definition of `main` and omit its declaration.

```
#include <stdio.h>
int lg(int);
main()
{
    int i, N;
    for (i = 1, N = 10; i <= 6; i++, N *= 10)
        printf("%7d %2d %9d\n", N, lg(N), N*lg(N));
}
int lg(int N)
{
    int i;
    for (i = 0; N > 0; i++, N /= 2)
        return i;
}
```

precisely which operations a program uses, we can easily extend it to any type of data for which we can support those operations. Second, by taking care to understand and to specify precisely what a program does, we can add the abstract operation that it performs to the operations at our disposal in solving new problems.

Program 3.2 implements a simple computation on numbers using a simple data type defined with a `typedef` operation and a function (which itself is implemented with a library function). The main function refers to the data type, not the built-in type of the number. By not specifying the type of the numbers that the program processes, we extend its potential utility. For example, this practice is likely to extend the useful lifetime of a program. When some new circumstance (a new application, or perhaps a new compiler or computer), presents us with a new type of number with which we would like to work, we can update our program just by changing the data type.

This example does not represent a fully general solution to the problem of developing a type-independent program for computing averages and standard deviations—nor is it intended to do so. For example, the program depends on converting a number of type `Number` to a `float` to be included in the running average and variance, so we might add that conversion as an operation to the data type, rather than depend on the `(float)` cast, which only works for built-in types of numbers.

If we were to try to do operations other than arithmetic operations, we would soon find the need to add more operations to the data type. For example, we might want to print the numbers, which would require that we implement, say, a `printNum` function. Such a function would be less convenient than using the built-in format conversions in `printf`. Whenever we strive to develop a data type based on identifying the operations of importance in a program, we need to strike a balance between the level of generality that we choose and the ease of implementation and use that results.

It is worthwhile to consider in detail how we might change the data type to make Program 3.2 work with other types of numbers, say `floats`, rather than with `ints`. There are a number of different mechanisms available in C that we could use to take advantage of the fact that we have localized references to the type of the data. For such

Program 3.2 Types of numbers

This program computes the average μ and standard deviation σ of a sequence x_1, x_2, \dots, x_N of integers generated by the library procedure `rand`, following the mathematical definitions

$$\mu = \frac{1}{N} \sum_{1 \leq i \leq N} x_i \quad \text{and} \quad \sigma^2 = \frac{1}{N} \sum_{1 \leq i \leq N} (x_i - \mu)^2 = \frac{1}{N} \sum_{1 \leq i \leq N} x_i^2 - \mu^2.$$

Note that a direct implementation from the definition of σ^2 requires one pass to compute the average and another to compute the sums of the squares of the differences between the members of the sequence and the average, but rearranging the formula makes it possible for us to compute σ^2 in one pass through the data.

We use the `typedef` declaration to localize reference to the fact that the type of the data is `int`. For example, we could keep the `typedef` and the function `randNum` in a separate file (referenced by an `include` directive), and then we could use this program to test random numbers of a different type by changing that file (*see text*).

Whatever the type of the data, the program uses `ints` for indices and `floats` to compute the average and standard deviation, and will be effective only if conversion functions from the data to `float` perform in a reasonable manner.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
typedef int Number;
Number randNum()
{
    return rand();
}
main(int argc, char *argv[])
{
    int i, N = atoi(argv[1]);
    float m1 = 0.0, m2 = 0.0;
    Number x;
    for (i = 0; i < N; i++)
    {
        x = randNum();
        m1 += ((float) x)/N;
        m2 += ((float) x*x)/N;
    }
    printf("      Average: %f\n", m1);
    printf("Std. deviation: %f\n", sqrt(m2-m1*m1));
}
```

a small program, the simplest is to make a copy of the file, then to change the `typedef` to

```
typedef float Number
```

and the function `randNum` to

```
return 1.0*rand()/RAND_MAX;
```

(which will return random floating-point numbers between 0 and 1). Even for such a small program, this approach is inconvenient because it leaves us with two copies of the `main` program, and we will have to make sure that any later changes in that program are reflected in both copies. In C, an alternative approach is to put the `typedef` and `randNum` into a separate *header file*—called, say, `Num.h`—replacing them with the directive

```
#include "Num.h"
```

in the code in Program 3.2. Then, we can make a second header file with different `typedef` and `randNum`, and, by renaming one of these files or the other `Num.h`, use the main program in Program 3.2 with either, without modifying it at all.

A third alternative, which is recommended software engineering practice, is to split the program into *three* files:

- An *interface*, which defines the data structure and declares the functions to be used to manipulate the data structure
- An *implementation* of the functions declared in the interface
- A *client* program that uses the functions declared in the interface to work at a higher level of abstraction

With this arrangement, we can use the main program in Program 3.2 with integers or floats, or extend it to work with other data types, just by compiling it together with the specific code for the data type of interest. Next, we shall consider the precise change that we need to convert Program 3.2 into a more flexible implementation, using this approach.

We think of the interface as a definition of the data type. It is a contract between the client program and the implementation program. The client agrees to access the data only through the functions defined in the interface, and the implementation agrees to deliver the promised functions.

For the example in Program 3.2, the *interface* would consist of the declarations

```
typedef int Number;  
Number randNum();
```

The first line specifies the type of the data to be processed, and the second specifies an operation associated with the type. This code might be kept, for example, in a file named Num.h.

The *implementation* of the interface in Num.h is an implementation of the randNum function, which might consist of the code

```
#include <stdlib.h>  
#include "Num.h"  
Number randNum()  
{ return rand(); }
```

The first line refers to the system-supplied interface that describes the rand() function; the second line refers to the interface that we are implementing (we include it as a check that the function we are implementing is the same type as the one that we declared), and the final two lines give the code for the function. This code might be kept, for example, in a file named int.c. The actual code for the rand function is kept in the standard C run-time library.

A *client* program corresponding to Program 3.2 would begin with the include directives for interfaces that declare the functions that it uses, as follows:

```
#include <stdio.h>  
#include <math.h>  
#include "Num.h"
```

The function main from Program 3.2 then can follow these three lines. This code might be kept, for example, in a file named avg.c.

Compiled together, the programs avg.c and int.c described in the previous paragraphs have the same functionality as Program 3.2, but they represent a more flexible implementation both because the code associated with the data type is encapsulated and can be used by other client programs and because avg.c can be used with other data types without being changed.

There are many other ways to support data types besides the client-interface-implementation scenario just described, but we will not dwell on distinctions among various alternatives because such

distinctions are best drawn in a systems-programming context, rather than in an algorithm-design context (*see reference section*). However, we do often make use of this basic design paradigm because it provides us with a natural way to substitute improved implementations for old ones, and therefore to compare different algorithms for the same applications problem. Chapter 4 is devoted to this topic.

We often want to build data structures that allow us to handle collections of data. The data structures may be huge, or they may be used extensively, so we are interested in identifying the important operations that we will perform on the data and in knowing how to implement those operations efficiently. Doing these tasks is taking the first steps in the process of incrementally building lower-level abstractions into higher-level ones; that process allows us to conveniently develop ever more powerful programs. The simplest mechanisms for grouping data in an organized way in C are *arrays*, which we consider in Section 3.2, and *structures*, which we consider next.

Structures are aggregate types that we use to define collections of data such that we can manipulate an entire collection as a unit, but can still refer to individual components of a given datum by name. Structures are not at the same level as built-in types such as `int` or `float` in C, because the only operations that are defined for them (beyond referring to their components) are copy and assignment. Thus, we can use a structure to define a new type of data, and can use it to name variables, and can pass those variables as arguments to functions, but we have specifically to define as functions any operations that we want to perform.

For example, when processing geometric data we might want to work with the abstract notion of points in the plane. Accordingly, we can write

```
struct point { float x; float y; };
```

to indicate that we will use type `point` to refer to pairs of floating-point numbers. For example, the statement

```
struct point a, b;
```

declares two variables of this type. We can refer to individual members of a structure by name. For example, the statements

```
a.x = 1.0; a.y = 1.0; b.x = 4.0; b.y = 5.0;
```

set `a` to represent the point (1, 1) and `b` to represent the point (4, 5).

Program 3.3 Point data type interface

This interface defines a data type consisting of the set of values “pairs of floating-point numbers” and the operation consists of a function that computes the distance between two points.

```
typedef struct { float x; float y; } point;
float distance(point, point);
```

We can also pass structures as arguments to functions. For example, the code

```
float distance(struct point a, struct point b)
{ float dx = a.x - b.x, dy = a.y - b.y;
  return sqrt(dx*dx + dy*dy);
}
```

defines a function that computes the distance between two points in the plane. This example illustrates the natural way in which structures allow us to aggregate our data in typical applications.

Program 3.3 is an interface that embodies the definition of a data type for points in the plane, uses a structure to represent the points, and includes an operation to compute the distance between two points. Program 3.4 is a function that implements the operation. We use interface-implementation arrangements like this to define data types whenever possible, because they encapsulate the definition (in the interface) and the implementation in a clear and direct manner. We make use of the data type in a client program by including the interface and by compiling the implementation with the client program (or by using appropriate separate-compilation facilities). Program 3.4 uses a `typedef` to define the point data type so that client programs can declare points as `point` instead of `struct point`, and do not have to make any assumptions about how the data types are represented. In Chapter 4, we shall see how to carry this separation between client and implementation one step further.

We cannot use Program 3.2 to process items of type `point` because arithmetic and type conversion operations are not defined for points. Modern languages such as C++ and Java have basic constructs that make it possible to use previously defined high-level abstract operations, even for newly defined types. With a sufficiently general

Program 3.4 Point data type implementation

This implementation provides the definition for the distance function for points that is declared in Program 3.3. It makes use of a library function to compute the square root.

```
#include <math.h>
#include "Point.h"
float distance(point a, point b)
{ float dx = a.x - b.x, dy = a.y - b.y;
  return sqrt(dx*dx + dy*dy);
}
```

interface, we could make these arrangements, even in C. In this book, however, although we strive to develop interfaces of general utility, we resist obscuring our algorithms or sacrificing good performance for that reason. Our primary goal is to make clear the effectiveness of the algorithmic ideas that we will be considering. Although we often stop short of a fully general solution, we do pay careful attention to the process of precisely defining the abstract operations that we want to perform, as well as the data structures and algorithms that will support those operations, because doing so is at the heart of developing efficient and effective programs. We will return to this issue, in detail, in Chapter 4.

The point structure example just given is a simple one that comprises two items of the same type. In general, structures can mix different types of data. We shall be working extensively with such structures throughout the rest of this chapter.

Beyond giving us the specific basic types `int`, `float`, and `char`, and the ability to build them into compound types with `struct`, C provides us with the ability to manipulate our data indirectly. A *pointer* is a reference to an object in memory (usually implemented as a machine address). We declare a variable `a` to be a pointer to (for example) an integer by writing `int *a`, and we can refer to the integer itself as `*a`. We can declare pointers to any type of data. The unary operator `&` gives the machine address of an object, and is useful for initializing pointers. For example, `*&a` is the same as `a`. We restrict ourselves to using `&` for this purpose, as we prefer to work at a somewhat higher level of abstraction than machine addresses when possible.

objects in a fixed sequential fashion that is more suitable for access than for manipulation; or a *list*, where we organize objects in a logical sequential fashion that is more suitable for manipulation than for access.

Exercises

▷ 3.1 Find the largest and smallest numbers that you can represent with types `int`, `long int`, `short int`, `float`, and `double` in your programming environment.

3.2 Test the random-number generator on your system by generating N random integers between 0 and $r - 1$ with `rand() % r` and computing the average and standard deviation for $r = 10, 100$, and 1000 and $N = 10^3, 10^4, 10^5$, and 10^6 .

3.3 Test the random-number generator on your system by generating N random numbers of type `double` between 0 and 1, transforming them to integers between 0 and $r - 1$ by multiplying by r and truncating the result, and computing the average and standard deviation for $r = 10, 100$, and 1000 and $N = 10^3, 10^4, 10^5$, and 10^6 .

○ 3.4 Do Exercises 3.2 and 3.3 for $r = 2, 4$, and 16 .

3.5 Implement the necessary functions to allow Program 3.2 to be used for random *bits* (numbers that can take only the values 0 or 1).

3.6 Define a `struct` suitable for representing a playing card.

3.7 Write a client program that uses the data type in Programs 3.3 and 3.4 for the following task: Read a sequence of points (pairs of floating-point numbers) from standard input, and find the one that is closest to the first.

• 3.8 Add a function to the point data type (Programs 3.3 and 3.4) that determines whether or not three points are collinear, to within a numerical tolerance of 10^{-4} . Assume that the points are all in the unit square.

3.9 Define a data type for points in the plane that is based on using polar coordinates instead of Cartesian coordinates.

• 3.10 Define a data type for *triangles* in the unit square, including a function that computes the area of a triangle. Then write a client program that generates random triples of pairs of `floats` between 0 and 1 and computes empirically the average area of the triangles generated.

3.2 Arrays

Perhaps the most fundamental data structure is the *array*, which is defined as a primitive in C and in most other programming languages.

We have already seen the use of an array as the basis for the development of an efficient algorithm, in the examples in Chapter 1; we shall see many more examples in this section.

An array is a fixed collection of same-type data that are stored contiguously and that are accessible by an index. We refer to the i th element of an array a as $a[i]$. It is the responsibility of the programmer to store something meaningful in an array position $a[i]$ before referring to $a[i]$. In C, it is also the responsibility of the programmer to use indices that are nonnegative and smaller than the array size. Neglecting these responsibilities are two of the more common programming mistakes.

Arrays are fundamental data structures in that they have a direct correspondence with memory systems on virtually all computers. To retrieve the contents of a word from memory in machine language, we provide an address. Thus, we could think of the entire computer memory as an array, with the memory addresses corresponding to array indices. Most computer-language processors translate programs that involve arrays into efficient machine-language programs that access memory directly, and we are safe in assuming that an array access such as $a[i]$ translates to just a few machine instructions.

A simple example of the use of an array is given by Program 3.5, which prints out all prime numbers less than 10000. The method used, which dates back to the third century B.C., is called the *sieve of Eratosthenes* (see Figure 3.1). It is typical of algorithms that exploit the fact that we can access efficiently any item of an array, given that item's index. The implementation has four loops, three of which access the items of the array sequentially, from beginning to end; the fourth skips through the array, i items at a time. In some cases, sequential processing is essential; in other cases, sequential ordering is used because it is as good as any other. For example, we could change the first loop in Program 3.5 to

```
for (a[1] = 0, i = N-1; i > 1; i--) a[i] = 1;
```

without any effect on the computation. We could also reverse the order of the inner loop in a similar manner, or we could change the final loop to print out the primes in decreasing order, but we could not change the order of the outer loop in the main computation, because it depends on all the integers less than i being processed before $a[i]$ is tested for being prime.

i	2	3	5	a[i]
2	1			1
3	1			1
4	1	0		
5	1			1
6	1	0		
7	1			1
8	1	0		
9	1	0		
10	1	0		
11	1			1
12	1	0	0	
13	1			1
14	1	0		
15	1	0		
16	1	0		
17	1			1
18	1	0	0	
19	1			1
20	1	0		
21	1	0		
22	1	0		
23	1			1
24	1	0	0	
25	1		0	
26	1	0		
27	1	0		
28	1	0		
29	1			1
30	1	0	0	0
31	1			1

Figure 3.1
Sieve of Eratosthenes

To compute the prime numbers less than 32, we initialize all the array entries to 1 (second column), to indicate that no numbers are known to be nonprime ($a[0]$ and $a[1]$ are not used and are not shown). Then, we set array entries whose indices are multiples of 2, 3, and 5 to 0, since we know these multiples to be nonprime. Indices corresponding to array entries that remain 1 are prime (rightmost column).

Program 3.5 Sieve of Eratosthenes

The goal of this program is to set $a[i]$ to 1 if i is prime, and to 0 if i is not prime. First, it sets to 1 all array elements, to indicate that no numbers are known to be nonprime. Then it sets to 0 array elements corresponding to indices that are known to be nonprime (multiples of known primes). If $a[i]$ is still 1 after all multiples of smaller primes have been set to 0, then we know it to be prime.

Because the program uses an array consisting of the simplest type of elements, 0–1 values, it would be more space efficient if we explicitly used an array of bits, rather than one of integers. Also, some programming environments might require the array to be global if N is huge, or we could allocate it dynamically (see Program 3.6).

```
#define N 10000
main()
{ int i, j, a[N];
  for (i = 2; i < N; i++) a[i] = 1;
  for (i = 2; i < N; i++)
    if (a[i])
      for (j = i; i*j < N; j++) a[i*j] = 0;
  for (i = 2; i < N; i++)
    if (a[i]) printf("%d ", i);
  printf("\n");
}
```

We will not analyze the running time of Program 3.5 in detail because that would take us astray into number theory, but it is clear that the running time is proportional to

$$N + N/2 + N/3 + N/5 + N/7 + N/11 + \dots$$

which is less than $N + N/2 + N/3 + N/4 + \dots = NH_N \sim N \ln N$.

One of the distinctive features of C is that an array name generates a pointer to the first element of the array (the one with index 0). Moreover, simple *pointer arithmetic* is allowed: if p is a pointer to an object of a certain type, then we can write code that assumes that objects of that type are arranged sequentially, and can use $*p$ to refer to the first object, $*(p+1)$ to refer to the second object, $*(p+2)$ to refer to the third object, and so forth. In other words,

$*(a+i)$ and $a[i]$ are equivalent in C.

Program 3.6 Dynamic memory allocation for an array

To change the value of the maximum prime computed in Program 3.5, we need to recompile the program. Instead, we can take the maximum desired number from the command line, and use it to allocate space for the array at execution time, using the library function `malloc` from `stdlib.c`. For example, if we compile this program and use 1000000 as a command-line argument, then we get all the primes less than 1 million (as long as our computer is big and fast enough to make the computation feasible); we can also debug with 100 (without using much time or space). We will use this idiom frequently, though, for brevity, we will omit the insufficient-memory test.

```
#include <stdlib.h>
main(int argc, char *argv[])
{ long int i, j, N = atoi(argv[1]);
  int *a = malloc(N*sizeof(int));
  if (a == NULL)
    { printf("Insufficient memory.\n"); return; }
  ...
```

This equivalence provides an alternate mechanism for accessing objects in arrays that is sometimes more convenient than indexing. This mechanism is most often used for arrays of characters (strings); we discuss it again in Section 3.6.

Like structures, pointers to arrays are significant because they allow us to manipulate the arrays efficiently as higher-level objects. In particular, we can pass a pointer to an array as an argument to a function, thus enabling that function to access objects in the array without having to make a copy of the whole array. This capability is indispensable when we write programs to manipulate huge arrays. For example, the search functions that we examined in Section 2.6 use this feature. We shall see other examples in Section 3.7.

The implementation in Program 3.5 assumes that the size of the array must be known beforehand: to run the program for a different value of `N`, we must change the constant `N` and recompile the program before executing it. Program 3.6 shows an alternate approach, where a user of the program can type in the value of `N`, and it will respond with the primes less than `N`. It uses two basic C mechanisms, both of which involve passing arrays as arguments to functions. The first is the

mechanism by which command-line arguments are passed to the main program, in an array `argv` of size `argc`. The array `argv` is a compound array made up of objects that are arrays (strings) themselves, so we shall defer discussing it in further detail until Section 3.7, and shall take on faith for the moment that the variable `N` gets the number that the user types when executing the program.

The second basic mechanism that we use in Program 3.6 is `malloc`, a function that *allocates* the amount of memory that we need for our array at execution time, and returns, for our exclusive use, a pointer to the array. In some programming languages, it is difficult or impossible to allocate arrays dynamically; in some other programming languages, memory allocation is an automatic mechanism. Dynamic allocation is an essential tool in programs that manipulate multiple arrays, some of which might have to be huge. In this case, without memory allocation, we would have to predeclare an array as large as any value that the user is allowed to type. In a large program where we might use many arrays, it is not feasible to do so for each array. We will generally use code like Program 3.6 in this book because of the flexibility that it provides, although in specific applications when the array size is known, simpler versions like Program 3.5 are perfectly suitable. If the array size is fixed and huge, the array may need to be global in some systems. We discuss several of the mechanisms behind memory allocation in Section 3.5, and we look at a way to use `malloc` to support an abstract dynamic growth facility for arrays in Section 14.5. As we shall see, however, such mechanisms have associated costs, so we generally regard arrays as having the characteristic property that, once allocated, their sizes are fixed, and cannot be changed.

Not only do arrays closely reflect the low-level mechanisms for accessing data in memory on most computers, but also they find widespread use because they correspond directly to natural methods of organizing data for applications. For example, arrays also correspond directly to *vectors*, the mathematical term for indexed lists of objects.

Program 3.7 is an example of a simulation program that uses an array. It simulates a sequence of *Bernoulli trials*, a familiar abstract concept from probability theory. If we flip a coin N times, the probability that we see k heads is

$$\binom{N}{k} \frac{1}{2^N} \approx \frac{e^{-(k-N/2)^2/N}}{\sqrt{\pi N/2}}.$$

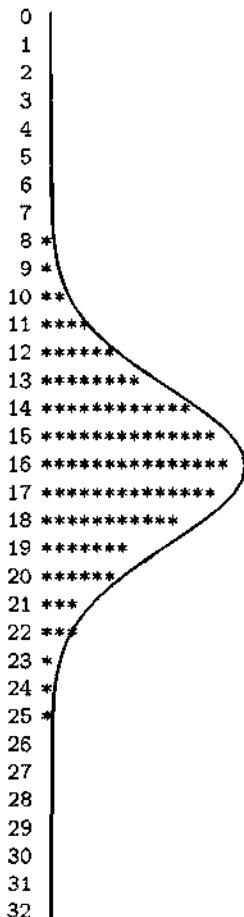


Figure 3.2
Coin-flipping simulation

This table shows the result of running Program 3.7 with $N = 32$ and $M = 1000$, simulating 1000 experiments of flipping a coin 32 times. The number of heads that we should see is approximated by the normal distribution function, which is drawn over the data.

Program 3.7 Coin-flipping simulation

If we flip a coin N times, we expect to get $N/2$ heads, but could get anywhere from 0 to N heads. This program runs the experiment M times, taking both N and M from the command line. It uses an array f to keep track of the frequency of occurrence of the outcome “ i heads” for $0 \leq i \leq N$, then prints out a histogram of the result of the experiments, with one asterisk for each 10 occurrences.

The operation on which this program is based—indexing an array with a computed value—is critical to the efficiency of many computational procedures.

```
#include <stdlib.h>
int heads()
{ return rand() < RAND_MAX/2; }
main(int argc, char *argv[])
{ int i, j, cnt;
  int N = atoi(argv[1]), M = atoi(argv[2]);
  int *f = malloc((N+1)*sizeof(int));
  for (j = 0; j <= N; j++) f[j] = 0;
  for (i = 0; i < M; i++, f[cnt]++)
    for (cnt = 0, j = 0; j <= N; j++)
      if (heads()) cnt++;
  for (j = 0; j <= N; j++)
  {
    printf("%2d ", j);
    for (i = 0; i < f[j]; i+=10) printf("*");
    printf("\n");
  }
}
```

The approximation is known as the *normal approximation*: the familiar bell-shaped curve. Figure 3.2 illustrates the output of Program 3.7 for 1000 trials of the experiment of flipping a coin 32 times. Many more details on the Bernoulli distribution and the normal approximation can be found in any text on probability, and we shall encounter these distributions again in Chapter 13. In the present context, our interest in the computation is that we use the numbers as indices into an array to count their frequency of occurrence. The ability of arrays to support this kind of operation is one of their prime virtues.

Program 3.8 Closest-point computation

This program illustrates the use of an array of structures, and is representative of the typical situation where we save items in an array to process them later, during some computation. It counts the number of pairs of N randomly generated points in the unit square that can be connected by a straight line of length less than d , using the data type for points described in Section 3.1. The running time is $O(N^2)$, so this program cannot be used for huge N . Program 3.20 provides a faster solution.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "Point.h"
float randFloat()
{ return 1.0*rand()/RAND_MAX; }
main(int argc, char *argv[])
{
    float d = atof(argv[2]);
    int i, j, cnt = 0, N = atoi(argv[1]);
    point *a = malloc(N*(sizeof(*a)));
    for (i = 0; i < N; i++)
        { a[i].x = randFloat(); a[i].y = randFloat(); }
    for (i = 0; i < N; i++)
        for (j = i+1; j < N; j++)
            if (distance(a[i], a[j]) < d) cnt++;
    printf("%d edges shorter than %f\n", cnt, d);
}
```

Programs 3.5 and 3.7 both compute array indices from the data at hand. In a sense, when we use a computed value to access an array of size N , we are taking N possibilities into account with just a single operation. This gain in efficiency is compelling when we can realize it, and we shall be encountering algorithms throughout the book that make use of arrays in this way.

We use arrays to organize all different manner of types of objects, not just integers. In C, we can declare arrays of any built-in or user-defined type (i.e., compound objects declared as structures). Program 3.8 illustrates the use of an array of structures for points in the plane using the structure definition that we considered in Section 3.1.

This program also illustrates a common use of arrays: to save data away so that they can be quickly accessed in an organized manner in some computation. Incidentally, Program 3.8 is also interesting as a prototypical quadratic algorithm, which checks all pairs of a set of N data items, and therefore takes time proportional to N^2 . In this book, we look for improvements whenever we see such an algorithm, because its use becomes infeasible as N grows. In this case, we shall see how to use a compound data structure to perform this computation in linear time, in Section 3.7.

We can create compound types of arbitrary complexity in a similar manner: We can have not just arrays of structs, but also arrays of arrays, or structs containing arrays. We will consider these different options in detail in Section 3.7. Before doing so, however, we will examine *linked lists*, which serve as the primary alternative to arrays for organizing collections of objects.

Exercises

- ▷ 3.11 Suppose that `a` is declared as `int a[99]`. Give the contents of the array after the following two statements are executed:

```
for (i = 0; i < 99; i++) a[i] = 98-i;  
for (i = 0; i < 99; i++) a[i] = a[a[i]];
```

- 3.12 Modify our implementation of the sieve of Eratosthenes (Program 3.5) to use an array of (i) chars; and (ii) bits. Determine the effects of these changes on the amount of space and time used by the program.

- ▷ 3.13 Use the sieve of Eratosthenes to determine the number of primes less than N , for $N = 10^3, 10^4, 10^5$, and 10^6 .

- 3.14 Use the sieve of Eratosthenes to draw a plot of N versus the number of primes less than N for N between 1 and 1000.

- 3.15 Empirically determine the effect of removing the test `if (a[i])` that guards the inner loop of Program 3.5, for $N = 10^3, 10^4, 10^5$, and 10^6 .

- 3.16 Analyze Program 3.5 to explain the effect that you observed in Exercise 3.15.

- ▷ 3.17 Write a program that counts the number of different integers less than 1000 that appear in an input stream.

- 3.18 Write a program that determines empirically the number of random positive integers less than 1000 that you can expect to generate before getting a repeated value.

- 3.19 Write a program that determines empirically the number of random positive integers less than 1000 that you can expect to generate before getting each value at least once.

3.20 Modify Program 3.7 to simulate a situation where the coin turns up heads with probability p . Run 1000 trials for an experiment with 32 flips with $p = 1/6$ to get output that you can compare with Figure 3.2.

3.21 Modify Program 3.7 to simulate a situation where the coin turns up heads with probability λ/N . Run 1000 trials for an experiment with 32 flips to get output that you can compare with Figure 3.2. This distribution is the classical *Poisson* distribution.

○ 3.22 Modify Program 3.8 to print out the coordinates of the closest pair of points.

• 3.23 Modify Program 3.8 to perform the same computation in d dimensions.

3.3 Linked Lists

When our primary interest is to go through a collection of items sequentially, one by one, we can organize the items as a *linked list*: a basic data structure where each item contains the information that we need to get to the next item. The primary advantage of linked lists over arrays is that the links provide us with the capability to rearrange the items efficiently. This flexibility is gained at the expense of quick access to any arbitrary item in the list, because the only way to get to an item in the list is to follow links, one node to the next. There are a number of ways to organize linked lists, all starting with the following basic definition.

Definition 3.2 A linked list is a set of items where each item is part of a node that also contains a link to a node.

We define nodes in terms of references to nodes, so linked lists are sometimes referred to as *self-referent* structures. Moreover, although a node's link usually refers to a different node, it could refer to the node itself, so linked lists can also be *cyclic* structures. The implications of these two facts will become apparent as we begin to consider concrete representations and applications of linked lists.

Normally, we think of linked lists as implementing a sequential arrangement of a set of items: Starting at a given node, we consider its item to be first in the sequence. Then, we follow its link to another node, which gives us an item that we consider to be second in the sequence, and so forth. In principle, the list could be cyclic and the sequence could seem infinite, but we most often work with lists that

correspond to a simple sequential arrangement of a finite set of items, adopting one of the following conventions for the link in the final node:

- It is a *null link* that points to no node.
- It refers to a *dummy node* that contains no item.
- It refers back to the first node, making the list a *circular list*.

In each case, following links from the first node to the final one defines a sequential arrangement of items. Arrays define a sequential ordering of items as well; in an array, however, the sequential organization is provided implicitly, by the position in the array. (Arrays also support arbitrary access by index, which lists do not.)

We first consider nodes with precisely one link, and, in most applications, we work with one-dimensional lists where all nodes except possibly the first one and the final one each have precisely one link referring to them. This corresponds to the simplest situation, which is also the one that interests us most, where linked lists correspond to finite sequences of items. We will consider more complicated situations in due course.

Linked lists are defined as a primitive in some programming environments, but not in C. However, the basic building blocks that we discussed in Section 3.1 are well suited to implementing linked lists. Specifically, we use pointers for links and structures for nodes. The `typedef` declaration gives us a way to refer to links and nodes, as follows:

```
typedef struct node *link;
struct node { Item item; link next; };
```

which is nothing more than C code for Definition 3.2. Links are pointers to nodes, and nodes consist of items and links. We assume that another part of the program uses `typedef` or some other mechanism to allow us to declare variables of type `Item`. We shall see more complicated representations in Chapter 4 that provide more flexibility and allow more efficient implementations of certain operations, but this simple representation will suffice for us to consider the fundamentals of list processing. We use similar conventions for linked structures throughout the book.

Memory allocation is a central consideration in the effective use of linked lists. Although we have defined a single structure (`struct node`), it is important to remember that we will have many instances of

this structure, one for each node that we want to use. Generally, we do not know the number of nodes that we will need until our program is executing, and various parts of our programs might have similar calls on the available memory, so we make use of system programs to keep track of our memory usage. To begin, whenever we want to use a new node, we need to create an instance of a node structure and to reserve memory for it—for example, we typically write code such as

```
link x = malloc(sizeof *x);
```

to direct the `malloc` function from `stdlib.h` and the `sizeof` operator to reserve enough memory for a node and to return a pointer to it in `x`. (This line of code does not refer directly to `node`, but a `link` can only refer to a `node`, so `sizeof` and `malloc` have the information that they need.) In Section 3.5, we shall consider the memory-allocation process in more detail. For the moment, for simplicity, we regard this line of code as a C idiom for creating new nodes. Indeed, our use of `malloc` is structured in this way throughout this book.

Now, once a list node is created, how do we refer to the information it comprises—its item and its link? We have already seen the basic operations that we need for this task: We simply dereference the pointer, then use the structure member names—the item in the node referenced by link `x` (which is of type `Item`) is `(*x).item` and the link (which is of type `link`) is `(*x).link`. These operations are so heavily used, however, that C provides the shorthand `x->item` and `x->link`, which are equivalent forms. Also, we so often need to use the phrase “the node referenced by link `x`” that we simply say “node `x`”—the link *does* name the node.

The correspondence between links and C pointers is essential, but we must bear in mind that the former is an abstraction and the latter a concrete representation. For example, we can also represent links with array indices, as we shall see at the end of this section.

Figures 3.3 and 3.4 show the two fundamental operations that we perform on linked lists. We can *delete* any item from a linked list, to make it shrink by 1 in length; and we can *insert* an item into a linked list at any point, to make it grow by 1 in length. For simplicity, we assume in these figures that the lists are circular and never become empty. We will consider null links, dummy nodes, and empty lists in Section 3.4. As shown in the figures, insertion and deletion each

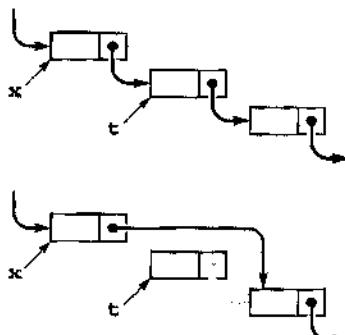


Figure 3.3
Linked-list deletion

To delete, or remove, the node following a given node `x` from a linked list, we set `t` to point to the node to be removed, then change `x`'s link to point to `t->next`. The pointer `t` can be used to refer to the removed node (to return it to a free list, for example). Although its link still points into the list, we generally do not use such a link after removing the node from the list, except perhaps to inform the system, via `free`, that its memory can be reclaimed.

require just two statements in C. To delete the node following node x , we use the statements

```
t = x->next; x->next = t->next;
```

or simply

```
x->next = x->next->next;
```

To insert node t into a list at a position following node x , we use the statements

```
t->next = x->next; x->next = t;
```

The simplicity of insertion and deletion is the *raison d'être* of linked lists. The corresponding operations are unnatural and inconvenient in arrays, because they require moving all of the array's contents following the affected item.

By contrast, linked lists are *not* well suited for the *find the k th item* (find an item given its index) operation that characterizes efficient access in arrays. In an array, we find the k th item simply by accessing $a[k]$; in a list, we have to traverse k links. Another operation that is unnatural on singly linked lists is “*find the item before a given item*.”

When we remove a node from a linked list using $x->next = x->next->next$, we may never be able to access it again. For small programs such as the examples we consider at first, this is no special concern, but we generally regard it as good programming practice to use the function `free`, which is the counterpart to `malloc`, for any node that we no longer wish to use. Specifically, the sequence of instructions

```
t = x->next; x->next = t->next; free(t);
```

not only removes t from our list but also informs the system that the memory it occupies may be used for some other purpose. We pay particular attention to `free` when we have large list objects, or large numbers of them, but we will ignore it until Section 3.5, so that we may focus on appreciating the benefits of linked structures.

We will see many examples of applications of these and other basic operations on linked lists in later chapters. Since the operations involve only a few statements, we often manipulate the lists directly rather than defining functions for inserting, deleting, and so forth. As an example, we consider next a program for solving the *Josephus problem* in the spirit of the sieve of Eratosthenes.

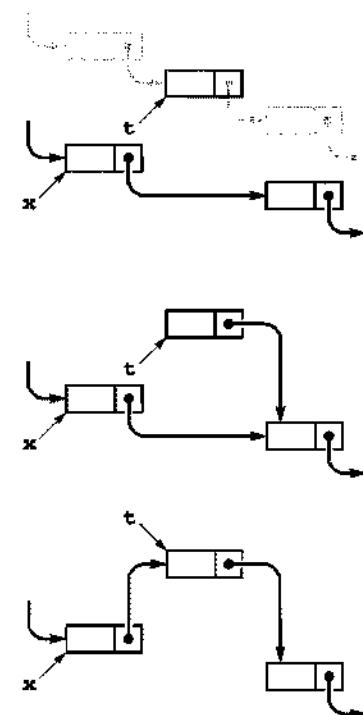


Figure 3.4
Linked-list insertion

To insert a given node t into a linked list at a position following another given node x (top), we set $t->next$ to $x->next$ (center), then set $x->next$ to t (bottom).

Program 3.9 Circular list example (Josephus problem)

To represent people arranged in a circle, we build a circular linked list, with a link from each person to the person on the left in the circle. The integer i represents the i th person in the circle. After building a one-node circular list for 1, we insert 2 through N after that node, resulting in a circle with 1 through N , leaving x pointing to N . Then, we skip $M - 1$ nodes, beginning with 1, and set the link of the $(M - 1)$ st to skip the M th, continuing until only one node is left.

```
#include <stdlib.h>
typedef struct node* link;
struct node { int item; link next; };
main(int argc, char *argv[])
{
    int i, N = atoi(argv[1]), M = atoi(argv[2]);
    link t = malloc(sizeof *t), x = t;
    t->item = 1; t->next = t;
    for (i = 2; i <= N; i++)
    {
        x = (x->next = malloc(sizeof *x));
        x->item = i; x->next = t;
    }
    while (x != x->next)
    {
        for (i = 1; i < M; i++) x = x->next;
        x->next = x->next->next; N--;
    }
    printf("%d\n", x->item);
}
```

We imagine that N people have decided to elect a leader by arranging themselves in a circle and eliminating every M th person around the circle, closing ranks as each person drops out. The problem is to find out which person will be the last one remaining (a mathematically inclined potential leader will figure out ahead of time which position in the circle to take). The identity of the elected leader is a function of N and M that we refer to as the *Josephus function*. More generally, we may wish to know the order in which the people are eliminated. For example, as shown in Figure 3.5, if $N = 9$ and $M = 5$, the people

are eliminated in the order 5 1 7 4 3 6 9 2, and 8 is the leader chosen. Program 3.9 reads in N and M and prints out this ordering.

Program 3.9 uses a *circular* linked list to simulate the election process directly. First, we build the list for 1 to N : We build a circular list consisting of a single node for person 1, then insert the nodes for people 2 through N , in that order, following that node in the list, using the insertion code illustrated in Figure 3.4. Then, we proceed through the list, counting through $M - 1$ items, deleting the next one using the code illustrated in Figure 3.3, and continuing until only one node is left (which then points to itself).

The sieve of Eratosthenes and the Josephus problem clearly illustrate the distinction between using arrays and using linked lists to represent a sequentially organized collection of objects. Using a linked list instead of an array for the sieve of Eratosthenes would be costly because the algorithm's efficiency depends on being able to access any array position quickly, and using an array instead of a linked list for the Josephus problem would be costly because the algorithm's efficiency depends on the ability to delete items quickly. When we choose a data structure, we *must* be aware of the effects of that choice upon the efficiency of the algorithms that will process the data. This interplay between data structures and algorithms is at the heart of the design process and is a recurring theme throughout this book.

In C, pointers provide a direct and convenient concrete realization of the abstract concept of a linked list, but the essential value of the abstraction does not depend on any particular implementation. For example, Figure 3.6 shows how we could use arrays of integers to implement the linked list for the Josephus problem. That is, we can implement linked lists using array indices, instead of pointers. Linked lists are thus useful even in the simplest of programming environments. Linked lists were useful well before pointer constructs were available in high-level languages such as C. Even in modern systems, simple array-based implementations are sometimes convenient.

Exercises

- ▷ 3.24 Write a function that returns the number of nodes on a circular list, given a pointer to one of the nodes on the list.
- 3.25 Write a code fragment that determines the number of nodes that are between the nodes referenced by two given pointers x and t to nodes on a circular list.

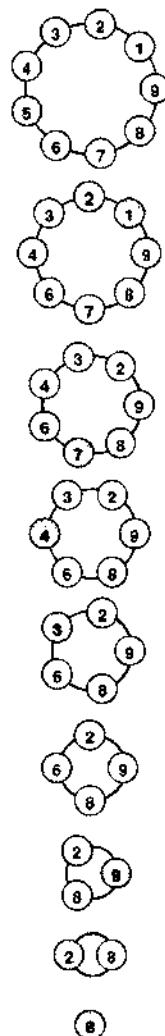


Figure 3.5
Example of Josephus election

This diagram shows the result of a Josephus-style election, where the group stands in a circle, then counts around the circle, eliminating every fifth person and closing the circle.

	0	1	2	3	4	5	6	7	8
item	1	2	3	4	5	6	7	8	9
next	1	2	3	4	5	6	7	8	0

5	1	2	3	4	5	6	7	8	9
	1	2	3	5	5	6	7	8	0

1	1	2	3	4	5	6	7	8	9
	1	2	3	5	5	6	7	8	1

7	1	2	3	4	5	6	7	8	9
	1	2	3	5	5	7	7	8	1

4	1	2	3	4	5	6	7	8	9
	1	2	5	5	6	7	7	8	1

3	1	2	3	4	5	6	7	8	9
	1	5	5	5	5	7	7	8	1

6	1	2	3	4	5	6	7	8	9
	1	7	5	5	5	7	7	8	1

9	1	2	3	4	5	6	7	8	9
	1	7	5	5	5	7	7	1	1

2	1	2	3	4	5	6	7	8	9
	1	7	5	5	5	7	7	7	1

Figure 3.6
Array representation of a linked list

This sequence shows the linked list for the Josephus problem (see Figure 3.5), implemented with array indices instead of pointers. The index of the item following the item with index 0 in the list is next[0], and so forth. Initially (top three rows), the item for person i has index $i-1$, and we form a circular list by setting next[i] to $i+1$ for i from 0 to 8 and next[8] to 0. To simulate the Josephus-election process, we change the links (next array entries) but do not move the items. Each pair of lines shows the result of moving through the list four times with $x = \text{next}[x]$, then deleting the fifth item (displayed at the left) by setting next[x] to next[next[x]].

3.26 Write a code fragment that, given pointers x and t to two disjoint circular lists, inserts the list pointed to by t into the list pointed to by x , at the point following x .

• 3.27 Given pointers x and t to nodes on a circular list, write a code fragment that moves the node following t to the position following the node following x on the list.

3.28 When building the list, Program 3.9 sets twice as many link values as it needs to because it maintains a circular list after each node is inserted. Modify the program to build the circular list without doing this extra work.

3.29 Give the running time of Program 3.9, within a constant factor, as a function of M and N .

3.30 Use Program 3.9 to determine the value of the Josephus function for $M = 2, 3, 5, 10$, and $N = 10^3, 10^4, 10^5$, and 10^6 .

3.31 Use Program 3.9 to plot the Josephus function versus N for $M = 10$ and N from 2 to 1000.

○ 3.32 Redo the table in Figure 3.6, beginning with item i initially at position $N-i$ in the array.

3.33 Develop a version of Program 3.9 that uses an array of indices to implement the linked list (see Figure 3.6).

3.4 Elementary List Processing

Linked lists bring us into a world of computing that is markedly different from that of arrays and structures. With arrays and structures, we save an item in memory and later refer to it by name (or by index) in much the same manner as we might put a piece of information in a file drawer or an address book; with linked lists, the manner in which we save information makes it more difficult to access but easier to rearrange. Working with data that are organized in linked lists is called *list processing*.

When we use arrays, we are susceptible to program bugs involving out-of-bounds array accesses. The most common bug that we encounter when using linked lists is a similar bug where we reference an undefined pointer. Another common mistake is to use a pointer that we have changed unknowingly. One reason that this problem arises is that we may have multiple pointers to the same node without necessarily realizing that that is the case. Program 3.9 avoids several such problems by using a circular list that is never empty, so that each

link always refers to a well-defined node, and each link can also be interpreted as referring to the list.

Developing correct and efficient code for list-processing applications is an acquired programming skill that requires practice and patience to develop. In this section, we consider examples and exercises that will increase our comfort with working with list-processing code. We shall see numerous other examples throughout the book, because linked structures are at the heart of some of our most successful algorithms.

As mentioned in Section 3.3, we use a number of different conventions for the first and final pointers in a list. We consider some of them in this section, even though we adopt the policy of reserving the term *linked list* to describe the simplest situation.

Definition 3.3 *A linked list is either a null link or a link to a node that contains an item and a link to a linked list.*

This definition is more restrictive than Definition 3.2, but it corresponds more closely to the mental model that we have when we write list-processing code. Rather than exclude all the other various conventions by using only this definition, and rather than provide specific definitions corresponding to each convention, we let both stand, with the understanding that it will be clear from the context which type of linked list we are using.

One of the most common operations that we perform on lists is to *traverse* them: We scan through the items on the list sequentially, performing some operation on each. For example, if *x* is a pointer to the first node of a list, the final node has a null pointer, and *visit* is a function that takes an item as an argument, then we might write

```
for (t = x; t != NULL; t = t->next) visit(t->item);
```

to traverse the list. This loop (or its equivalent while form) is as ubiquitous in list-processing programs as is the corresponding

```
for (i = 0; i < N; i++)
```

in array-processing programs.

Program 3.10 is an implementation of a simple list-processing task, reversing the order of the nodes on a list. It takes a linked list as an argument, and returns a linked list comprising the same nodes, but with the order reversed. Figure 3.7 shows the change that the

Program 3.10 List reversal

This function reverses the links in a list, returning a pointer to the final node, which then points to the next-to-final node, and so forth, with the link in the first node of the original list set to `NULL`. To accomplish this task, we need to maintain links to three consecutive nodes in the list.

```
link reverse(link x)
{ link t, y = x, r = NULL;
  while (y != NULL)
    { t = y->next; y->next = r; r = y; y = t; }
  return r;
}
```

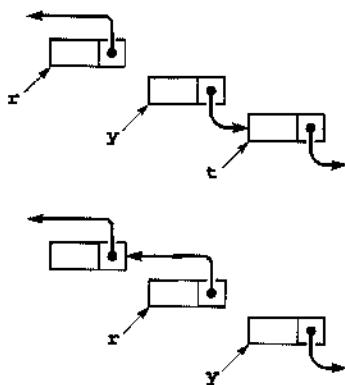


Figure 3.7
List reversal

To reverse the order of a list, we maintain a pointer `x` to the portion of the list already processed, and a pointer `y` to the portion of the list not yet seen. This diagram shows how the pointers change for each node in the list. We save a pointer to the node following `y` in `t`, change `y`'s link to point to `x`, and then move `x` to `y` and `y` to `t`.

function makes for each node in its main loop. Such a diagram makes it easier for us to check each statement of the program to be sure that the code changes the links as intended, and programmers typically use these diagrams to understand the operation of list-processing implementations.

Program 3.11 is an implementation of another list-processing task: rearranging the nodes of a list to put their items in sorted order. It generates N random integers, puts them into a list in the order that they were generated, rearranges the nodes to put their items in sorted order, and prints out the sorted sequence. As we discuss in Chapter 6, the expected running time of this program is proportional to N^2 , so the program is not useful for large N . Beyond this observation, we defer discussing the sort aspect of this program to Chapter 6, because we shall see a great many methods for sorting in Chapters 6 through 10. Our purpose now is to present the implementation as an example of a list-processing application.

The lists in Program 3.11 illustrate another commonly used convention: We maintain a dummy node called a *head node* at the beginning of each list. We ignore the item field in a list's head node, but maintain its link as the pointer to the node containing the first item in the list. The program uses two lists: one to collect the random input in the first loop, and the other to collect the sorted output in the second loop. Figure 3.8 diagrams the changes that Program 3.11 makes during one iteration of its main loop. We take the next node

Program 3.11 List insertion sort

This code generates N random integers between 0 and 999, builds a linked list with one number per node (first `for` loop), and then rearranges the nodes so that the numbers appear in order when we traverse the list (second `for` loop). To accomplish the sort, we maintain two lists, an input (unsorted) list and an output (sorted) list. On each iteration of the loop, we remove a node from the input and insert it into position in the output. The code is simplified by the use of head nodes for each list, that contain the links to the first nodes on the lists. For example, without the head node, the case where the node to be inserted into the output list goes at the beginning would involve extra code.

```
struct node heada, headb;
link t, u, x, a = &heada, b;
for (i = 0, t = a; i < N; i++)
{
    t->next = malloc(sizeof *t);
    t = t->next; t->next = NULL;
    t->item = rand() % 1000;
}
b = &headb; b->next = NULL;
for (t = a->next; t != NULL; t = u)
{
    u = t->next;
    for (x = b; x->next != NULL; x = x->next)
        if (x->next->item > t->item) break;
    t->next = x->next; x->next = t;
}
```

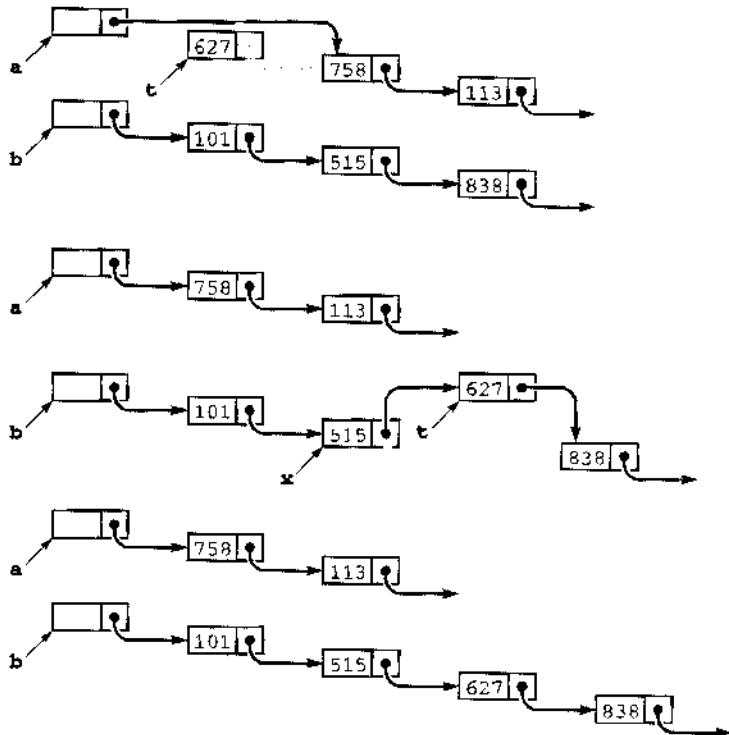
off the input list, find where it belongs in the output list, and link it into position.

The primary reason to use the head node at the beginning becomes clear when we consider the process of adding the *first* node to the sorted list. This node is the one in the input list with the smallest item, and it could be anywhere on the list. We have three options:

- Duplicate the `for` loop that finds the smallest item and set up a one-node list in the same manner as in Program 3.9.
- Test whether the output list is empty every time that we wish to insert a node.

Figure 3.8
Linked-list sort

This diagram depicts one step in transforming an unordered linked list (pointed to by *a*) into an ordered one (pointed to by *b*), using insertion sort. We take the first node of the unordered list, keeping a pointer to it in *t* (top). Then, we search through *b* to find the first node *x* with *x->next->item* $>$ *t->item* (or *x->next = NULL*), and insert *t* into the list following *x* (center). These operations reduce the length of *a* by one node, and increase the length of *b* by one node, keeping *b* in order (bottom). Iterating, we eventually exhaust *a* and have the nodes in order in *b*.



- Use a dummy head node whose link points to the first node on the list, as in the given implementation.

The first option is inelegant and requires extra code; the second is also inelegant and requires extra time.

The use of a head node does incur some cost (the extra node), and we can avoid the head node in many common applications. For example, we can also view Program 3.10 as having an input list (the original list) and an output list (the reversed list), but we do not need to use a head node in that program because all insertions into the output list are at the beginning. We shall see still other applications that are more simply coded when we use a dummy node, rather than a null link, at the *tail* of the list. There are no hard-and-fast rules about whether or not to use dummy nodes—the choice is a matter of style combined with an understanding of effects on performance. Good programmers

Table 3.1 Head and tail conventions in linked lists

This table gives implementations of basic list-processing operations with five commonly used conventions. This type of code is used in simple applications where the list-processing code is inline.

Circular, never empty

```
first insert: head->next = head;
insert t after x: t->next = x->next; x->next = t;
delete after x: x->next = x->next->next;
traversal loop: t = head;
    do { ... t = t->next; } while (t != head);
test if one item: if (head->next == head)
```

Head pointer, null tail

```
initialize: head = NULL;
insert t after x: if (x == NULL) { head = t; head->next = NULL; }
    else { t->next = x->next; x->next = t; }
delete after x: t = x->next; x->next = t->next;
traversal loop: for (t = head; t != NULL; t = t->next)
test if empty: if (head == NULL)
```

Dummy head node, null tail

```
initialize: head = malloc(sizeof *head);
    head->next = NULL;
insert t after x: t->next = x->next; x->next = t;
delete after x: t = x->next; x->next = t->next;
traversal loop: for (t = head->next; t != NULL; t = t->next)
test if empty: if (head->next == NULL)
```

Dummy head and tail nodes

```
initialize: head = malloc(sizeof *head);
    z = malloc(sizeof *z);
    head->next = z; z->next = z;
insert t after x: t->next = x->next; x->next = t;
delete after x: x->next = x->next->next;
traversal loop: for (t = head->next; t != z; t = t->next)
test if empty: if (head->next == z)
```

Program 3.12 List-processing interface

This code, which might be kept in an interface file `list.h`, specifies the types of nodes and links, and declares some of the operations that we might want to perform on them. We declare our own functions for allocating and freeing memory for list nodes. The function `initNodes` is for the convenience of the implementation. The `typedef` for `Node` and the functions `Next` and `Item` allow clients to use lists without dependence upon implementation details.

```
typedef struct node* link;
struct node { itemType item; link next; };
typedef link Node;
void initNodes(int);
link newNode(int);
void freeNode(link);
void insertNext(link, link);
link deleteNext(link);
link Next(link);
int Item(link);
```

enjoy the challenge of picking the convention that most simplifies the task at hand. We shall see several such tradeoffs throughout this book.

For reference, a number of options for linked-list conventions are laid out in Table 3.1; others are discussed in the exercises. In all the cases in Table 3.1, we use a pointer `head` to refer to the list, and we maintain a consistent stance that our program manages links to nodes, using the given code for various operations. Allocating and freeing memory for nodes and filling them with information is the same for all the conventions. Robust functions implementing the same operations would have extra code to check for error conditions. The purpose of the table is to expose similarities and differences among the various options.

Another important situation in which it is sometimes convenient to use head nodes occurs when we want to pass pointers to lists as arguments to functions that may modify the list, in the same way that we do for arrays. Using a head node allows the function to accept or return an empty list. If we do not have a head node, we need a mechanism for the function to inform the calling program when

Program 3.13 List allocation for the Josephus problem

This program for the Josephus problem is an example of a client program utilizing the list-processing primitives declared in Program 3.12 and implemented in Program 3.14.

```
#include "list.h"
main(int argc, char *argv[])
{ int i, N = atoi(argv[1]), M = atoi(argv[2]);
  Node t, x;
  initNodes(N);
  for (i = 2, x = newNode(i); i <= N; i++)
    { t = newNode(i); insertNext(x, t); x = t; }
  while (x != Next(x))
  {
    for (i = 1; i < M; i++) x = Next(x);
    freeNode(deleteNext(x));
  }
  printf("%d\n", Item(x));
}
```

it leaves an empty list. One such mechanism—the one used for the function in Program 3.10—is to have list-processing functions take pointers to input lists as arguments and return pointers to output lists. With this convention, we do not need to use head nodes. Furthermore, it is well suited to recursive list processing, which we use extensively throughout the book (see Section 5.1).

Program 3.12 illustrates declarations for a set of black-box functions that implement the basic list operations, in case we choose not to repeat the code inline. Program 3.13 is our Josephus-election program (Program 3.9) recast as a client program that uses this interface. Identifying the important operations that we use in a computation and defining them in an interface gives us the flexibility to consider different concrete implementations of critical operations and to test their effectiveness. We consider one implementation for the operations defined in Program 3.12 in Section 3.5 (see Program 3.14), but we could also try other alternatives without changing Program 3.13 at all (see Exercise 3.52). This theme will recur throughout the book,

and we will consider mechanisms to make it easier to develop such implementations in Chapter 4.

Some programmers prefer to encapsulate all operations on low-level data structures such as linked lists by defining functions for every low-level operation in interfaces like Program 3.12. Indeed, as we shall see in Chapter 4, the C class mechanism makes it easy to do so. However, that extra layer of abstraction sometimes masks the fact that just a few low-level operations are involved. In this book, when we are implementing higher-level interfaces, we usually write low-level operations on linked structures directly, to clearly expose the essential details of our algorithms and data structures. We shall see many examples in Chapter 4.

By adding more links, we can add the capability to move backward through a linked list. For example, we can support the operation “find the item *before* a given item” by using a *doubly linked list* in which we maintain two links for each node: one (*prev*) to the item before, and another (*next*) to the item after. With dummy nodes or a circular list, we can ensure that x , $x \rightarrow \text{next} \rightarrow \text{prev}$, and $x \rightarrow \text{prev} \rightarrow \text{next}$ are the same for every node in a doubly linked list. Figures 3.9 and 3.10 show the basic link manipulations required to implement *delete*, *insert after*, and *insert before*, in a doubly linked list. Note that, for *delete*, we do not need extra information about the node before it (or the node after it) in the list, as we did for singly linked lists—that information is contained in the node itself.

Indeed, the primary significance of doubly linked lists is that they allow us to delete a node when the *only* information that we have about that node is a link to it. Typical situations are when the link is passed as an argument in a function call, and when the node has other links and is also part of some other data structure. Providing this extra capability doubles the space needed for links in each node and doubles the number of link manipulations per basic operation, so doubly linked lists are not normally used unless specifically called for. We defer considering detailed implementations to a few specific situations where we have such a need—for example in Section 9.5.

We use linked lists throughout this book, first for basic ADT implementations (see Chapter 4), then as components in more complex data structures. Linked lists are many programmers’ first exposure to an abstract data structure that is under the programmers’ direct

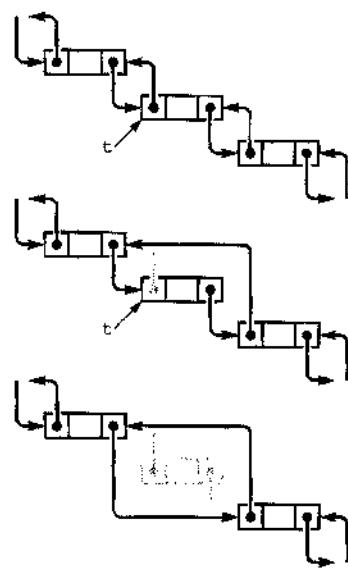


Figure 3.9
Deletion in a doubly-linked list

In a doubly-linked list, a pointer to a node is sufficient information for us to be able to remove it, as diagrammed here. Given t , we set $t \rightarrow \text{next} \rightarrow \text{prev}$ to $t \rightarrow \text{prev}$ (center), and $t \rightarrow \text{prev} \rightarrow \text{next}$ to $t \rightarrow \text{next}$ (bottom).

control. They represent an essential tool for our use in developing the high-level abstract data structures that we need for a host of important problems, as we shall see.

Exercises

▷ 3.34 Write a function that moves the largest item on a given list to be the final node on the list.

3.35 Write a function that moves the smallest item on a given list to be the first node on the list.

3.36 Write a function that rearranges a linked list to put the nodes in even positions after the nodes in odd positions in the list, preserving the relative order of both the evens and the odds.

3.37 Implement a code fragment for a linked list that exchanges the positions of the nodes after the nodes referenced by two given links *t* and *u*.

○ 3.38 Write a function that takes a link to a list as argument and returns a link to a copy of the list (a new list that contains the same items, in the same order).

3.39 Write a function that takes two arguments—a link to a list and a function that takes a link as argument—and removes all items on the given list for which the function returns a nonzero value.

3.40 Solve Exercise 3.39, but make copies of the nodes that pass the test and return a link to a list containing those nodes, in the order that they appear in the original list.

3.41 Implement a version of Program 3.10 that uses a head node.

3.42 Implement a version of Program 3.11 that does not use head nodes.

3.43 Implement a version of Program 3.9 that uses a head node.

3.44 Implement a function that exchanges two given nodes on a doubly-linked list.

○ 3.45 Give an entry for Table 3.1 for a list that is never empty, is referred to with a pointer to the first node, and for which the final node has a pointer to itself.

3.46 Give an entry for Table 3.1 for a circular list that has a dummy node, which serves as both head and tail.

3.5 Memory Allocation for Lists

An advantage of linked lists over arrays is that linked lists gracefully grow and shrink during their lifetime. In particular, their maximum

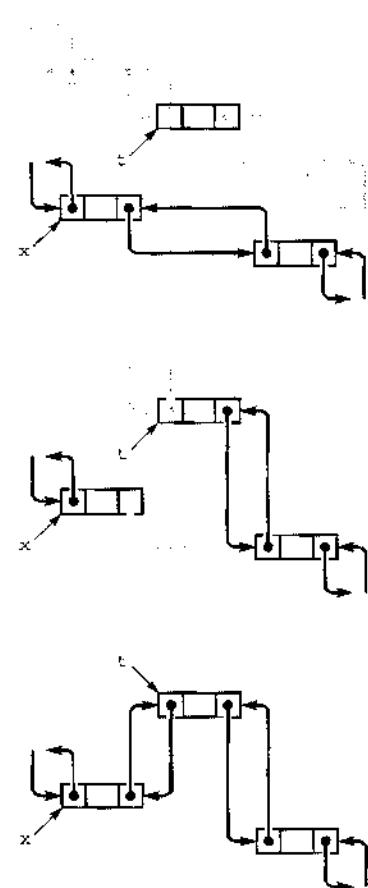


Figure 3.10
Insertion in a doubly-linked list

To insert a node into a doubly-linked list, we need to set four pointers. We can insert a new node after a given node (diagrammed here) or before a given node. We insert a given node *t* after another given node *x* by setting *t*->*next* to *x*->*next* and *x*->*next*->*prev* to *t* (center), and then setting *x*->*next* to *t* and *t*->*prev* to *x* (bottom).

size does not need to be known in advance. One important practical ramification of this observation is that we can have several data structures share the same space, without paying particular attention to their relative size at any time.

The crux of the matter is to consider how the system function `malloc` might be implemented. For example, when we delete a node from a list, it is one thing for us to rearrange the links so that the node is no longer hooked into the list, but what does the system do with the space that the node occupied? And how does the system recycle space such that it can always find space for a node when `malloc` is called and more space is needed? The mechanisms behind these questions provide another example of the utility of elementary list processing.

The system function `free` is the counterpart to `malloc`. When we are done using a chunk of allocated memory, we call `free` to inform the system that the chunk is available for later use. *Dynamic memory allocation* is the process of managing memory and responding to calls on `malloc` and `free` from client programs.

When we are calling `malloc` directly in applications such as Program 3.9 or Program 3.11, all the calls request memory blocks of the same size. This case is typical, and an alternate method of keeping track of memory available for allocation immediately suggests itself: Simply use a linked list! All nodes that are not on any list that is in use can be kept together on a single linked list. We refer to this list as the *free list*. When we need to allocate space for a node, we get it by *deleting* it from the free list; when we remove a node from any of our lists, we dispose of it by *inserting* it onto the free list.

Program 3.14 is an implementation of the interface defined in Program 3.12, including the memory-allocation functions. When compiled with Program 3.13, it produces the same result as the direct implementation with which we began in Program 3.9. Maintaining the free list for fixed-size nodes is a trivial task, given the basic operations for inserting nodes onto and deleting nodes from a list.

Figure 3.11 illustrates how the free list grows as nodes are freed, for Program 3.13. For simplicity, the figure assumes a linked-list implementation (no head node) based on array indices.

Implementing a general-purpose memory allocator in a C environment is much more complex than is suggested by our simple examples, and the implementation of `malloc` in the standard library

	0	1	2	3	4	5	6	7	8
item	1	2	3	4	5	6	7	8	9
next	1	2	3	4	5	6	7	8	0
<hr/>									
	1	2	3	4	5	6	7	8	9
4	1	2	3	5	.	6	7	8	0
0	1	2	3	4	5	6	7	8	1
6	1	2	3	4	5	6	7	8	9
5	4	2	3	5	.	7	0	8	1
3	1	2	3	4	5	6	7	8	9
2	4	2	5	6	.	7	0	8	1
7	1	2	3	4	5	6	7	8	9
1	4	5	3	6	.	7	0	8	1
5	1	2	3	4	5	6	7	8	9
4	4	7	3	6	.	2	0	8	1
8	1	2	3	4	5	6	7	8	9
0	4	7	3	6	.	1	2	0	1
<hr/>									
	1	2	3	4	5	6	7	8	9
1	4	8	3	6	.	2	0	7	5

Figure 3.11
Array representation of a linked list, with free list

This version of Figure 3.6 shows the result of maintaining a free list with the nodes deleted from the circular list, with the index of first node on the free list given at the left. At the end of the process, the free list is a linked list containing all the items that were deleted. Following the links, starting at 1, we see the items in the order 2 9 6 3 4 7 1 5, which is the reverse of the order in which they were deleted.

Program 3.14 Implementation of list-processing interface

This program gives implementations of the functions declared in Program 3.12, and illustrates a standard approach to allocating memory for fixed-size nodes. We build a free list that is initialized to the maximum number of nodes that our program will use, all linked together. Then, when a client program allocates a node, we remove that node from the free list; when a client program frees a node, we link that node in to the free list.

By convention, client programs do not refer to list nodes except through function calls, and nodes returned to client programs have self-links. These conventions provide some measure of protection against referencing undefined pointers.

```
#include <stdlib.h>
#include "list.h"
link freelist;
void initNodes(int N)
{ int i;
  freelist = malloc((N+1)*(sizeof *freelist));
  for (i = 0; i < N+1; i++)
    freelist[i].next = &freelist[i+1];
  freelist[N].next = NULL;
}
link newNode(int i)
{ link x = deleteNext(freelist);
  x->item = i; x->next = x;
  return x;
}
void freeNode(link x)
{ insertNext(freelist, x); }
void insertNext(link x, link t)
{ t->next = x->next; x->next = t; }
link deleteNext(link x)
{ link t = x->next; x->next = t->next; return t; }
link Next(link x)
{ return x->next; }
int Item(link x)
{ return x->item; }
```

is certainly not as simple as is indicated by Program 3.14. One primary difference between the two is that `malloc` has to handle storage-allocation requests for nodes of varying sizes, ranging from tiny to huge. Several clever algorithms have been developed for this purpose. Another approach that is used by some modern systems is to relieve the user of the need to `free` nodes explicitly by using *garbage-collection* algorithms to remove automatically any nodes not referenced by any link. Several clever storage management algorithms have also been developed along these lines. We will not consider them in further detail because their performance characteristics are dependent on properties of specific systems and machines.

Programs that can take advantage of specialized knowledge about an application often are more efficient than general-purpose programs for the same task. Memory allocation is no exception to this maxim. An algorithm that has to handle storage requests of varying sizes cannot know that we are always going to be making requests for blocks of one fixed size, and therefore cannot take advantage of that fact. Paradoxically, another reason to avoid general-purpose library functions is that doing so makes programs more portable—we can protect ourselves against unexpected performance changes when the library changes or when we move to a different system. Many programmers have found that using a simple memory allocator like the one illustrated in Program 3.14 is an effective way to develop efficient and portable programs that use linked lists. This approach applies to a number of the algorithms that we will consider throughout this book, which make similar kinds of demands on the memory-management system.

Exercises

- 3.47 Write a program that frees (calls `free` with a pointer to) all the nodes on a given linked list.
- 3.48 Write a program that frees the nodes in positions that are divisible by 5 in a linked list (the fifth, tenth, fifteenth, and so forth).
- 3.49 Write a program that frees the nodes in even positions in a linked list (the second, fourth, sixth, and so forth).
- 3.50 Implement the interface in Program 3.12 using `malloc` and `free` directly in `allocNode` and `freeNode`, respectively.
- 3.51 Run empirical studies comparing the running times of the memory-allocation functions in Program 3.14 with `malloc` and `free` (see Exer-

cise 3.50) for Program 3.13 with $M = 2$ and $N = 10^3, 10^4, 10^5$, and 10^6 .

3.52 Implement the interface in Program 3.12 using array indices (and no head node) rather than pointers, in such a way that Figure 3.11 is a trace of the operation of your program.

o 3.53 Suppose that you have a set of nodes with no null pointers (each node points to itself or to some other node in the set). Prove that you ultimately get into a cycle if you start at any given node and follow links.

• 3.54 Under the conditions of Exercise 3.53, write a code fragment that, given a pointer to a node, finds the number of different nodes that it ultimately reaches by following links from that node, *without* modifying any nodes. Do not use more than a constant amount of extra memory space.

•• 3.55 Under the conditions of Exercise 3.54, write a function that determines whether or not two given links, if followed, eventually end up on the same cycle.

3.6 Strings

We use the term *string* to refer to a variable-length array of characters, defined by a starting point and by a string-termination character marking the end. Strings are valuable as low-level data structures, for two basic reasons. First, many computing applications involve processing textual data, which can be represented directly with strings. Second, many computer systems provide direct and efficient access to *bytes* of memory, which correspond directly to characters in strings. That is, in a great many situations, the string abstraction matches needs of the application to the capabilities of the machine.

The abstract notion of a sequence of characters ending with a string-termination character could be implemented in many ways. For example, we could use a linked list, although that choice would exact a cost of one pointer per character. The concrete array-based implementation that we consider in this section is the one that is built into C. We shall also examine other implementations in Chapter 4.

The difference between a string and an array of characters revolves around *length*. Both represent contiguous areas of memory, but the length of an array is set at the time that the array is created, whereas the length of a string may change during the execution of a program. This difference has interesting implications, which we shall explore shortly.

We need to reserve memory for a string, either at compile time, by declaring a fixed-length array of characters, or at execution time, by calling `malloc`. Once the array is allocated, we can fill it with characters, starting at the beginning, and ending with the string-termination character. Without a string-termination character, a string is no more or no less than an array of characters; with the string-termination character, we can work at a higher level of abstraction, and consider only the portion of the array from the beginning to the string-termination character to contain meaningful information. In C, the termination character is the one with value 0, also known as '`\0`'.

For example, to find the length of a string, we count the number of characters between the beginning and the string-termination character. Table 3.2 gives simple operations that we commonly perform on strings. They all involve processing the strings by scanning through them from beginning to end. Many of these functions are available as library functions declared in `<string.h>`, although many programmers use slightly modified versions in inline code for simple applications. Robust functions implementing the same operations would have extra code to check for error conditions. We include the code here not just to highlight its simplicity, but also to expose its performance characteristics plainly.

One of the most important operations that we perform on strings is the *compare* operation, which tells us which of two strings would appear first in the dictionary. For purposes of discussion, we assume an idealized dictionary (since the actual rules for strings that contain punctuation, uppercase and lowercase letters, numbers, and so forth are rather complex), and compare strings character-by-character, from beginning to end. This ordering is called *lexicographic order*. We also use the compare function to tell whether strings are equal—by convention, the compare function returns a negative number if the first argument string appears before the second in the dictionary, returns 0 if they are equal, and returns 1 if the first appears after the second in lexicographic order. It is critical to take note that doing equality testing is *not* the same as determining whether two string *pointers* are equal—if two string pointers are equal, then so are the referenced strings (they are the *same* string), but we also could have different string pointers that point to equal strings (identical sequences of characters). Numerous applications involve storing information as strings, then processing

Table 3.2 Elementary string-processing operations

This table gives implementations of basic string-processing operations, using two different C language primitives. The pointer approach leads to more compact code, but the indexed-array approach is a more natural way to express the algorithms and leads to code that is easier to understand. The pointer version of the concatenate operation is the same as the indexed array version, and the pointer version of prefixed compare is obtained from the normal compare in the same way as for the indexed array version and is omitted. The implementations all take time proportional to string lengths.

Indexed array versions

```
Compute string length (strlen(a))
    for (i = 0; a[i] != 0; i++) ; return i;

Copy (strcpy(a, b))
    for (i = 0; (a[i] = b[i]) != 0; i++) ;

Compare (strcmp(a, b))
    for (i = 0; a[i] == b[i]; i++)
        if (a[i] == 0) return 0;
    return a[i] - b[i];

Compare (prefix) (strncmp(a, b, strlen(a)))
    for (i = 0; a[i] == b[i]; i++)
        if (a[i] == 0) return 0;
    if (a[i] == 0) return 0;
    return a[i] - b[i];

Append (strcat(a, b))
    strcpy(a+strlen(a), b)
```

Equivalent pointer versions

```
Compute string length (strlen(a))
    b = a; while (*b++) ; return b-a-1;

Copy (strcpy(a, b))
    while (*a++ = *b++) ;

Compare (strcmp(a, b))
    while (*a++ == *b++)
        if (*(a-1) == 0) return 0;
    return *(a-1) - *(b-1);
```

Program 3.15 String search

This program discovers all occurrences of a word from the command line in a (presumably much larger) text string. We declare the text string as a fixed-size character array (we could also use `malloc`, as in Program 3.6) and read it from standard input, using `getchar()`. Memory for the word from the command line-argument is allocated by the system before this program is invoked, and we find the string pointer in `argv[1]`. For each starting position `i` in `a`, we try matching the substring starting at that position with `p`, testing for equality character by character. Whenever we reach the end of `p` successfully, we print out the starting position `i` of the occurrence of the word in the text.

```
#include <stdio.h>
#define N 10000
main(int argc, char *argv[])
{
    int i, j, t;
    char a[N], *p = argv[1];
    for (i = 0; i < N-1; a[i] = t, i++)
        if ((t = getchar()) == EOF) break;
    a[i] = 0;
    for (i = 0; a[i] != 0; i++)
    {
        for (j = 0; p[j] != 0; j++)
            if (a[i+j] != p[j]) break;
        if (p[j] == 0) printf("%d ", i);
    }
    printf("\n");
}
```

or accessing that information by comparing the strings, so the compare operation is a particularly critical one. We shall see a specific example in Section 3.7 and in numerous other places throughout the book.

Program 3.15 is an implementation of a simple string-processing task, which prints out the places where a short pattern string appears within a long text string. Several sophisticated algorithms have been developed for this task, but this simple one illustrates several of the conventions that we use when processing strings in C.

String processing provides a convincing example of the need to be knowledgeable about the performance of library functions. The

problem is that a library function might take more time than we expect, intuitively. For example, *determining the length of a string takes time proportional to the length of the string*. Ignoring this fact can lead to severe performance problems. For example, after a quick look at the library, we might implement the pattern match in Program 3.15 as follows:

```
for (i = 0; i < strlen(a); i++)
    if (strncmp(&a[i], p, strlen(p)) == 0)
        printf("%d ", i);
```

Unfortunately, this code fragment takes time proportional to at least the *square* of the length of *a*, no matter what code is in the body of the loop, because it goes all the way through *a* to determine its length each time through the loop. This cost is considerable, even prohibitive: Running this program to check whether this book (which has more than 1 million characters) contains a certain word would require trillions of instructions. Problems such as this one are difficult to detect because the program might work fine when we are debugging it for small strings, but then slow down or even never finish when it goes into production. Moreover, we can avoid such problems only if we know about them!

This kind of error is called a *performance bug*, because the code can be verified to be correct, but it does not perform as efficiently as we (implicitly) expect. Before we can even begin the study of efficient algorithms, we must be certain to have eliminated performance bugs of this type. Although standard libraries have many virtues, we must be wary of the dangers of using them for simple functions of this kind.

One of the essential concepts that we return to time and again in this book is that different implementations of the same abstract notion can lead to widely different performance characteristics. For example, if we keep track of the length of the string, we can support a function that can return the length of a string in constant time, but for which other operations run more slowly. One implementation might be appropriate for one application; another implementation might be appropriate for another application.

Library functions, all too often, cannot guarantee to provide the best performance for all applications. Even if (as in the case of *strlen*) the performance of a library function is well documented, we have no assurance that some future implementation might not involve

performance changes that will have adverse effects on our programs. This issue is critical in the design of algorithms and data structures, and thus is one that we must always bear in mind. We shall discuss other examples and further ramifications in Chapter 4.

Strings are actually pointers to chars. In some cases, this realization can lead to compact code for string-processing functions. For example, to copy one string to another, we could write

```
while (*a++ = *b++) ;
```

instead of

```
for (i = 0; a[i] != 0; i++) a[i] = b[i];
```

or the third option given in Table 3.2. These two ways of referring to strings are equivalent, but may lead to code with different performance properties on different machines. We generally use the array version for clarity and the pointer version for economy, reserving detailed study of which is best for particular pieces of frequently executed code in particular applications.

Memory allocation for strings is more difficult than for linked lists because strings vary in size. Indeed, a fully general mechanism to reserve space for strings is neither more nor less than the system-provided `malloc` and `free` functions. As mentioned in Section 3.6, various algorithms have been developed for this problem, whose performance characteristics are system and machine dependent. Often, memory allocation is a less severe problem when we are working with strings than it might first appear, because we work with *pointers* to the strings, rather than with the characters themselves. Indeed, we *do not* normally assume in C code that all strings sit in individually allocated chunks of memory. We tend to assume that each string sits in memory of indeterminate allocation, just big enough for the string and its termination character. We must be very careful to ensure adequate allocation when we are performing operations that build or lengthen strings. As an example, we shall consider a program that reads strings and manipulates them in Section 3.7.

Exercises

- ▷ 3.56 Write a program that takes a string as argument, and that prints out a table giving, for each character that occurs in the string, the character and its frequency of occurrence.

▷ 3.57 Write a program that checks whether a given string is a palindrome (reads the same backward or forward), ignoring blanks. For example, your program should report success for the string if i had a hifi.

3.58 Suppose that memory for strings is individually allocated. Write versions of `strcpy` and `strcat` that allocate memory and return a pointer to the new string for the result.

3.59 Write a program that takes a string as argument and reads a sequence of words (sequences of characters separated by blank space) from standard input, printing out those that appear as substrings somewhere in the argument string.

3.60 Write a program that replaces substrings of more than one blank in a given string by exactly one blank.

3.61 Implement a pointer version of Program 3.15.

○ 3.62 Write an efficient program that finds the length of the longest sequence of blanks in a given string, examining as few characters in the string as possible. *Hint:* Your program should become faster as the length of the sequence of blanks increases.

3.7 Compound Data Structures

Arrays, linked lists, and strings all provide simple ways to structure data sequentially. They provide a first level of abstraction that we can use to group objects in ways amenable to processing the objects efficiently. Having settled on these abstractions, we can use them in a hierarchical fashion to build up more complex structures. We can contemplate arrays of arrays, arrays of lists, arrays of strings, and so forth. In this section, we consider examples of such structures.

In the same way that one-dimensional arrays correspond to vectors, *two-dimensional* arrays, with two indices, correspond to *matrices*, and are widely used in mathematical computations. For example, we might use the following code to multiply two matrices `a` and `b`, leaving the result in a third matrix `c`.

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0, c[i][j] = 0.0; k < N; k++)
            c[i][j] += a[i][k]*b[k][j];
```

We frequently encounter mathematical computations that are naturally expressed in terms of multidimensional arrays.

Program 3.16 Two-dimensional array allocation

This function dynamically allocates the memory for a two-dimensional array, as an array of arrays. We first allocate an array of pointers, then allocate memory for each row. With this function, the statement

```
int **a = malloc2d(M, N);
```

allocates an M -by- N array of integers.

```
int **malloc2d(int r, int c)
{ int i;
  int **t = malloc(r * sizeof(int *));
  for (i = 0; i < r; i++)
    t[i] = malloc(c * sizeof(int));
  return t;
}
```

Beyond mathematical applications, a familiar way to structure information is to use a table of numbers organized into rows and columns. A table of students' grades in a course might have one row for each student, and one column for each assignment. In C, such a table would be represented as a two-dimensional array with one index for the row and one for the column. If we were to have 100 students and 10 assignments, we would write `grades[100][10]` to declare the array, and then refer to the i th student's grade on the j th assignment as `grade[i][j]`. To compute the average grade on an assignment, we sum together the elements in a column and divide by the number of rows; to compute a particular student's average grade in the course, we sum together the elements in a row and divide by the number of columns, and so forth. Two-dimensional arrays are widely used in applications of this type. On a computer, it is often convenient and straightforward to use more than two dimensions: An instructor might use a third index to keep student-grade tables for a sequence of years.

Two-dimensional arrays are a notational convenience, as the numbers are ultimately stored in the computer memory, which is essentially a one-dimensional array. In many programming environments, two-dimensional arrays are stored in *row-major order* in a one-dimensional array: In an array `a[M][N]`, the first N positions would be occupied by the first row (elements `a[0][0]` through `a[0][N-1]`),

Program 3.17 Sorting an array of strings

This program illustrates an important string-processing function: rearranging a set of strings into sorted order. We read strings into a buffer large enough to hold them all, maintaining a pointer to each string in an array, then rearrange the pointers to put the pointer to the smallest string in the first position in the array, the pointer to the second smallest string in the second position in the array, and so forth.

The `qsort` library function that actually does the sort takes four arguments: a pointer to the beginning of the array, the number of objects, the size of each object, and a comparison function. It achieves independence from the type of object being sorted by blindly rearranging the blocks of data that represent objects (in this case string pointers) and by using a comparison function that takes pointers to void as argument. This code casts these back to type pointer to pointer to char for `strcmp`. To actually access the first character in a string for a comparison, we dereference three pointers: one to get the index (which is a pointer) into our array, one to get the pointer to the string (using the index), and one to get the character (using the pointer).

We use a different method to achieve type independence for our sorting and searching functions (see Chapters 4 and 6).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define Nmax 1000
#define Mmax 10000
char buf[Mmax]; int M = 0;
int compare(void *i, void *j)
{ return strcmp(*(char **)i, *(char **)j); }
main()
{ int i, N;
  char* a[Nmax];
  for (N = 0; N < Nmax; N++)
  {
    a[N] = &buf[M];
    if (scanf("%s", a[N]) == EOF) break;
    M += strlen(a[N])+1;
  }
  qsort(a, N, sizeof(char*), compare);
  for (i = 0; i < N; i++) printf("%s\n", a[i]);
}
```

the second N positions by the second row (elements $a[1][0]$ through $a[1][N-1]$), and so forth. With row-major order, the final line in the matrix-multiplication code in the beginning of this section is precisely equivalent to

$$c[N*i+j] = a[N*i+k]*b[N*k+j]$$

The same scheme generalizes to provide a facility for arrays with more dimensions. In C, multidimensional arrays may be implemented in a more general manner: we can define them to be compound data structures (arrays of arrays). This provides the flexibility, for example, to have an array of arrays that differ in size.

We saw a method in Program 3.6 for dynamic allocation of arrays that allows us to use our programs for varying problem sizes without recompiling them, and would like to have a similar method for multidimensional arrays. How do we allocate memory for multidimensional arrays whose size we do not know at compile time? That is, we want to be able to refer to an array element such as $a[i][j]$ in a program, but cannot declare it as `int a[M][N]` (for example) because we do not know the values of M and N . For row-major order, a statement like

```
int* a = malloc(M*N*sizeof(int));
```

would be an effective way to allocate an M -by- N array of integers, but this solution will not work in all C environments, because not all implementations use row-major order. Program 3.16 gives a solution for two-dimensional arrays, based on their definition as arrays of arrays.

Program 3.17 illustrates the use of a similar compound structure: an array of strings. At first blush, since our abstract notion of a string is an array of characters, we might represent arrays of strings as arrays of arrays. However, the concrete representation that we use for a string in C is a *pointer* to the beginning of an array of characters, so an array of strings can also be an array of pointers. As illustrated in Figure 3.12, we then can get the effect of rearranging strings simply by rearranging the pointers in the array. Program 3.17 uses the `qsort` library function—implementing such functions is the subject of Chapters 6 through 9 in general and of Chapter 7 in particular. This example illustrates a typical scenario for processing strings: we read the characters themselves into a huge one-dimensional array, save

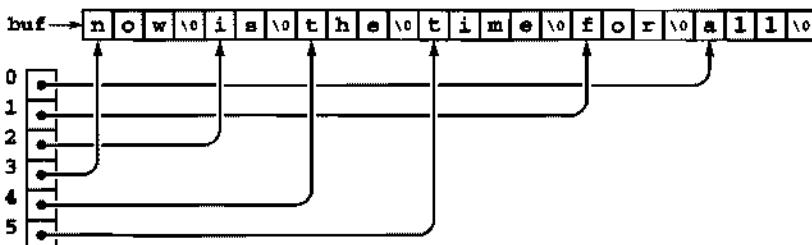
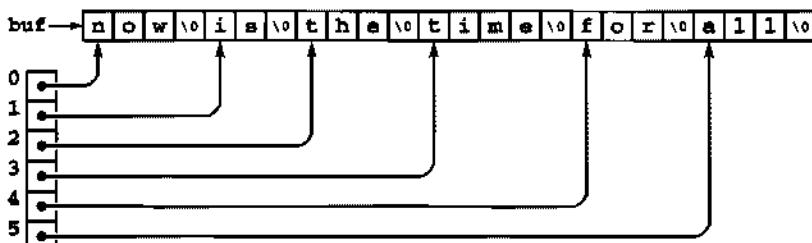


Figure 3.12
String sort

When processing strings, we normally work with pointers into a buffer that contains the strings (top), because the pointers are easier to manipulate than the strings themselves, which vary in length. For example, the result of a sort is to rearrange the pointers such that accessing them in order gives the strings in alphabetical (lexicographic) order.

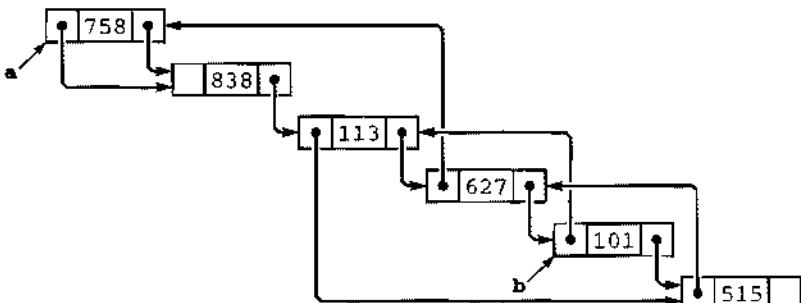
pointers to individual strings (delimiting them with string-termination characters), then manipulate the pointers.

We have already encountered another use of arrays of strings: the `argv` array that is used to pass argument strings to `main` in C programs. The system stores in a string buffer the command line typed by the user and passes to `main` a pointer to an array of pointers to strings in that buffer. We use conversion functions to calculate numbers corresponding to some arguments; we use other arguments as strings, directly.

We can build compound data structures exclusively with links, as well. Figure 3.13 shows an example of a *multilist*, where nodes have multiple link fields and belong to independently maintained linked lists. In algorithm design, we often use more than one link to build up complex data structures, but in such a way that they are used to allow us to process them efficiently. For example, a doubly linked list is a multilist that satisfies the constraint that $x \rightarrow l \rightarrow x$ and $x \rightarrow r \rightarrow l$ are both equal to x . We shall examine a much more important data structure with two links per node in Chapter 5.

Figure 3.13
A multilist

We can link together nodes with two link fields in two independent lists, one using one link field, the other using the other link field. Here, the right link field links together nodes in one order (for example, this order could be the order in which the nodes were created) and the left link field links together nodes in a different order (for example, in this case, sorted order, perhaps the result of insertion sort using the left link field only). Following right links from a, we visit the nodes in the order created; following left links from b, we visit the nodes in sorted order.



If a multidimensional matrix is *sparse* (relatively few of the entries are *nonzero*), then we might use a *multilist* rather than a multidimensional array to represent it. We could use one node for each value in the matrix and one link for each dimension, with the link pointing to the next item in that dimension. This arrangement reduces the storage required from the product of the maximum indices in the dimensions to be proportional to the number of nonzero entries, but increases the time required for many algorithms, because they have to traverse links to access individual elements.

To see more examples of compound data structures and to highlight the distinction between indexed and linked data structures, we next consider data structures for representing graphs. A *graph* is a fundamental combinatorial object that is defined simply as a set of objects (called *vertices*) and a set of connections among the vertices (called *edges*). We have already encountered graphs, in the connectivity problem of Chapter 1.

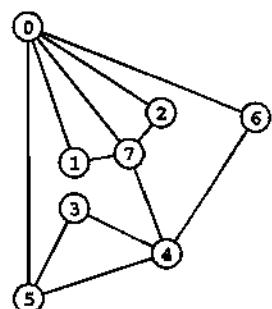
We assume that a graph with V vertices and E edges is defined by a set of E pairs of integers between 0 and $V-1$. That is, we assume that the vertices are labeled with the integers 0, 1, ..., $V-1$, and that the edges are specified as pairs of vertices. As in Chapter 1 we take the pair $i-j$ as defining a connection between i and j and thus having the same meaning as the pair $j-i$. Graphs that comprise such edges are called *undirected graphs*. We shall consider other types of graphs in Part 7.

One straightforward method for representing a graph is to use a two-dimensional array, called an *adjacency matrix*. With an adjacency matrix, we can determine immediately whether or not there is an edge from vertex i to vertex j , just by checking whether row i and column

Program 3.18 Adjacency-matrix graph representation

This program reads a set of edges that define an undirected graph and builds an adjacency-matrix representation for the graph, setting $a[i][j]$ and $a[j][i]$ to 1 if there is an edge from i to j or j to i in the graph, or to 0 if there is no such edge. The program assumes that the number of vertices V is a compile-time constant. Otherwise, it would need to dynamically allocate the array that represents the adjacency matrix (see Exercise 3.72).

```
#include <stdio.h>
#include <stdlib.h>
main()
{ int i, j, adj[V][V];
  for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
      adj[i][j] = 0;
  for (i = 0; i < V; i++) adj[i][i] = 1;
  while (scanf("%d %d\n", &i, &j) == 2)
    { adj[i][j] = 1; adj[j][i] = 1; }
}
```



	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	1	1	1	0	0	0
4	0	0	0	1	1	1	1	0
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

j of the matrix is nonzero. For the undirected graphs that we are considering, if there is an entry in row i and column j , then there also must be an entry in row j and column i , so the matrix is symmetric. Figure 3.14 shows an example of an adjacency matrix for an undirected graph; Program 3.18 shows how we can create an adjacency matrix, given a sequence of edges as input.

Another straightforward method for representing a graph is to use an array of linked lists, called *adjacency lists*. We keep a linked list for each vertex, with a node for each vertex connected to that vertex. For the undirected graphs that we are considering, if there is a node for j in i 's list, then there must be a node for i in j 's list. Figure 3.15 shows an example of the adjacency-lists representation of an undirected graph; Program 3.19 shows how we can create an adjacency-lists representation of a graph, given a sequence of edges as input.

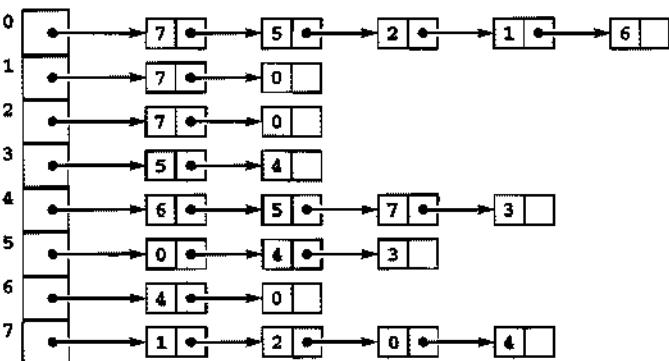
Both graph representations are arrays of simpler data structures—one for each vertex describing the edges incident on that vertex. For

Figure 3.14
Graph with adjacency matrix representation

A graph is a set of vertices and a set of edges connecting the vertices. For simplicity, we assign indices (nonnegative integers, consecutively, starting at 0) to the vertices. An adjacency matrix is a two-dimensional array where we represent a graph by putting a 1 bit in row i and column j if and only if there is an edge from vertex i to vertex j . The array is symmetric about the diagonal. By convention, we assign 1 bits on the diagonal (each vertex is connected to itself). For example, the sixth row (and the sixth column) says that vertex 6 is connected to vertices 0, 4, and 6.

Figure 3.15
Adjacency-lists representation
of a graph

This representation of the graph in Figure 3.14 uses an array of lists. The space required is proportional to the number of nodes plus the number of edges. To find the indices of the vertices connected to a given vertex i , we look at the i th position in an array, which contains a pointer to a linked list containing one node for each vertex connected to i .



an adjacency matrix, the simpler data structure is implemented as an indexed array; for an adjacency list, it is implemented as a linked list.

Thus, we face straightforward space tradeoffs when we represent a graph. The adjacency matrix uses space proportional to V^2 ; the adjacency lists use space proportional to $V + E$. If there are few edges (such a graph is said to be *sparse*), then the adjacency-lists representation uses far less space; if most pairs of vertices are connected by edges (such a graph is said to be *dense*), the adjacency-matrix representation might be preferable, because it involves no links. Some algorithms will be more efficient with the adjacency-matrix representation, because it allows the question “is there an edge between vertex i and vertex j ? ” to be answered in constant time; other algorithms will be more efficient with the adjacency-lists representation, because it allows us to process all the edges in a graph in time proportional to $V + E$, rather than to V^2 . We see a specific example of this tradeoff in Section 5.8.

Both the adjacency-matrix and the adjacency-lists graph representations can be extended straightforwardly to handle other types of graphs (see, for example, Exercise 3.71). They serve as the basis for most of the graph-processing algorithms that we shall consider in Part 7.

To conclude this chapter, we consider an example that shows the use of compound data structures to provide an efficient solution to the simple geometric problem that we considered in Section 3.2. Given d , we want to know how many pairs from a set of N points in the unit square can be connected by a straight line of length less than d .

Program 3.19 Adjacency-lists graph representation

This program reads a set of edges that define a graph and builds an adjacency-matrix representation for the graph. An adjacency list for a graph is an array of lists, one for each vertex, where the j th list contains a linked list of the nodes connected to the j th vertex.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node *link;
struct node
{ int v; link next; };
link NEW(int v, link next)
{ link x = malloc(sizeof *x);
  x->v = v; x->next = next;
  return x;
}
main()
{ int i, j; link adj[V];
  for (i = 0; i < V; i++) adj[i] = NULL;
  while (scanf("%d %d\n", &i, &j) == 2)
  {
    adj[j] = NEW(i, adj[j]);
    adj[i] = NEW(j, adj[i]);
  }
}
```

Program 3.20 uses a two-dimensional array of linked lists to improve the running time of Program 3.8 by a factor of about $1/d^2$ when N is sufficiently large. It divides the unit square up into a grid of equal-sized smaller squares. Then, for each square, it builds a linked list of all the points that fall into that square. The two-dimensional array provides the capability to access immediately the set of points close to a given point; the linked lists provide the flexibility to store the points where they may fall without our having to know ahead of time how many points fall into each grid square.

The space used by Program 3.20 is proportional to $1/d^2 + N$, but the running time is $O(d^2N^2)$, which is a substantial improvement over the brute-force algorithm of Program 3.8 for small d . For exam-

Program 3.20 A two-dimensional array of lists

This program illustrates the effectiveness of proper data-structure choice, for the geometric computation of Program 3.8. It divides the unit square into a grid, and maintains a two-dimensional array of linked lists, with one list corresponding to each grid square. The grid is chosen to be sufficiently fine that all points within distance d of any given point are either in the same grid square or an adjacent one. The function `malloc2d` is like the one in Program 3.16, but for objects of type `link` instead of `int`.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "Point.h"
typedef struct node* link;
struct node { point p; link next; };
link **grid; int G; float d; int cnt = 0;
gridinsert(float x, float y)
{ int i, j; link s;
  int X = x*G +1; int Y = y*G+1;
  link t = malloc(sizeof *t);
  t->p.x = x; t->p.y = y;
  for (i = X-1; i <= X+1; i++)
    for (j = Y-1; j <= Y+1; j++)
      for (s = grid[i][j]; s != NULL; s = s->next)
        if (distance(s->p, t->p) < d) cnt++;
  t->next = grid[X][Y]; grid[X][Y] = t;
}
main(int argc, char *argv[])
{ int i, j, N = atoi(argv[1]);
  d = atof(argv[2]); G = 1/d;
  grid = malloc2d(G+2, G+2);
  for (i = 0; i < G+2; i++)
    for (j = 0; j < G+2; j++)
      grid[i][j] = NULL;
  for (i = 0; i < N; i++)
    gridinsert(randFloat(), randFloat());
  printf("%d edges shorter than %f\n", cnt, d);
}
```

ple, with $N = 10^6$ and $d = 0.001$, we can solve the problem in time and space that is effectively linear, whereas the brute-force algorithm would require a prohibitive amount of time. We can use this data structure as the basis for solving many other geometric problems, as well. For example, combined with a union-find algorithm from Chapter 1, it gives a near-linear algorithm for determining whether a set of N random points in the plane can be connected together with lines of length d —a fundamental problem of interest in networking and circuit design.

As suggested by the examples that we have seen in this section, there is no end to the level of complexity that we can build up from the basic abstract constructs that we can use to structure data of differing types into objects and sequence the objects into compound objects, either implicitly or with explicit links. These examples still leave us one step away from full generality in structuring data, as we shall see in Chapter 5. Before taking that step, however, we shall consider the important abstract data structures that we can build with linked lists and arrays—basic tools that will help us in developing the next level of generality.

Exercises

- 3.63 Write a version of Program 3.16 that handles *three*-dimensional arrays.
- 3.64 Modify Program 3.17 to process input strings individually (allocate memory for each string after reading it from the input). You can assume that all strings have less than 100 characters.
- 3.65 Write a program to fill in a two-dimensional array of 0–1 values by setting $a[i][j]$ to 1 if the greatest common divisor of i and j is 1, and to 0 otherwise.
- 3.66 Use Program 3.20 in conjunction with Program 1.4 to develop an efficient program that can determine whether a set of N points can be connected with edges of length less than d .
- 3.67 Write a program to convert a sparse matrix from a two-dimensional array to a multilist with nodes for only nonzero values.
- 3.68 Implement matrix multiplication for matrices represented with multilists.
- ▷ 3.69 Show the adjacency matrix that is built by Program 3.18 given the input pairs 0–2, 1–4, 2–5, 3–6, 0–4, 6–0, and 1–3.
- ▷ 3.70 Show the adjacency lists that are built by Program 3.19 given the input pairs 0–2, 1–4, 2–5, 3–6, 0–4, 6–0, and 1–3.

- 3.71 A *directed graph* is one where vertex connections have orientations; edges go *from* one vertex *to* another. Do Exercises 3.69 and 3.70 under the assumption that the input pairs represent a directed graph, with $i-j$ signifying that there is an edge from i to j . Also, draw the graph, using arrows to indicate edge orientations.
- 3.72 Modify Program 3.18 to take the number of vertices as a command-line argument, then dynamically allocate the adjacency matrix.
- 3.73 Modify Program 3.19 to take the number of vertices as a command-line argument, then dynamically allocate the array of lists.
- 3.74 Write a function that uses the adjacency matrix of a graph to calculate, given vertices a and b , the number of vertices c with the property that there is an edge from a to c and from c to b .
- 3.75 Answer Exercise 3.74, but use adjacency lists.

CHAPTER FOUR

Abstract Data Types

DEVELOPING ABSTRACT MODELS for our data and for the ways in which our programs process those data is an essential ingredient in the process of solving problems with a computer. We see examples of this principle at a low level in everyday programming (for example when we use arrays and linked lists, as discussed in Chapter 3) and at a high level in problem-solving (as we saw in Chapter 1, when we used union–find forests to solve the connectivity problem). In this chapter, we consider *abstract data types* (*ADTs*), which allow us to build programs that use high-level abstractions. With abstract data types, we can separate the conceptual transformations that our programs perform on our data from any particular data-structure representation and algorithm implementation.

All computer systems are based on *layers of abstraction*: We adopt the abstract model of a bit that can take on a binary 0–1 value from certain physical properties of silicon and other materials; then, we adopt the abstract model of a machine from dynamic properties of the values of a certain set of bits; then, we adopt the abstract model of a programming language that we realize by controlling the machine with a machine-language program; then, we adopt the abstract notion of an algorithm implemented as a C language program. Abstract data types allow us to take this process further, to develop abstract mechanisms for certain computational tasks at a higher level than provided by the C system, to develop application-specific abstract mechanisms that are suitable for solving problems in numerous applications areas, and to build higher-level abstract mechanisms that use these basic

mechanisms. Abstract data types give us an ever-expanding set of tools that we can use to attack new problems.

On the one hand, our use of abstract mechanisms frees us from detailed concern about how they are implemented; on the other hand, when performance matters in a program, we need to be cognizant of the costs of basic operations. We use many basic abstractions that are built into the computer hardware and provide the basis for machine instructions; we implement others in software; and we use still others that are provided in previously written systems software. Often, we build higher-level abstract mechanisms in terms of more primitive ones. The same basic principle holds at all levels: We want to identify the critical operations in our programs and the critical characteristics of our data, to define both precisely at an abstract level, and to develop efficient concrete mechanisms to support them. We consider many examples of this principle in this chapter.

To develop a new layer of abstraction, we need to *define* the abstract objects that we want to manipulate and the operations that we perform on them; we need to *represent* the data in some data structure and to *implement* the operations; and (the point of the exercise) we want to ensure that the objects are convenient to *use* to solve an applications problem. These comments apply to simple data types as well, and the basic mechanisms that we discussed in Chapter 3 to support data types will serve our purposes, with one significant extension.

Definition 4.1 *An abstract data type (ADT) is a data type (a set of values and a collection of operations on those values) that is accessed only through an interface. We refer to a program that uses an ADT as a client, and a program that specifies the data type as an implementation.*

The key distinction that makes a data type abstract is drawn by the word *only*: with an ADT, client programs do not access any data values except through the operations provided in the interface. The representation of the data and the functions that implement the operations are in the implementation, and are completely separated from the client, by the interface. We say that the interface is *opaque*: the client cannot see the implementation through the interface.

For example, the interface for the data type for points (Program 3.3) in Section 3.1 explicitly declares that points are represented

as structures with pairs of floats, with members named *x* and *y*. Indeed, this use of data types is common in large software systems: we develop a set of conventions for how data is to be represented (and define a number of associated operations) and make those conventions available in an interface for use by client programs that comprise a large system. The data type ensures that all parts of the system are in agreement on the representation of core system-wide data structures. While valuable, this strategy has a flaw: if we need to *change* the data representation, then we need to change all the client programs. Program 3.3 again provides a simple example: one reason for developing the data type is to make it convenient for client programs to manipulate points, and we expect that clients will access the individual coordinates when needed. But we cannot change to a different representation (polar coordinates, say, or three dimensions, or even different data types for the individual coordinates) without changing all the client programs.

Our implementation of a simple list-processing interface in Section 3.4 (Program 3.12) is an example of a first step towards an ADT. In the client program that we considered (Program 3.13), we adopted the convention that we would access the data only through the operations defined in the interface, and were therefore able to consider changing the representation without changing the client (see Exercise 3.52). Adopting such a convention amounts to using the data type as though it was abstract, but leaves us exposed to subtle bugs, because the data representation remains available to clients, in the interface, and we would have to be vigilant to ensure that they do not depend upon it, even if accidentally. With true ADTs, we provide no information to clients about data representation, and are thus free to change it.

Definition 4.1 does not specify what an interface is or how the data type and the operations are to be described. This imprecision is necessary because specifying such information in full generality requires a formal mathematical language and eventually leads to difficult mathematical questions. This question is central in programming language design. We shall discuss the specification problem further after we consider examples of ADTs.

ADTs have emerged as an effective mechanism for organizing large modern software systems. They provide a way to limit the size and complexity of the interface between (potentially complicated) al-

gorithms and associated data structures and (a potentially large number of) programs that use the algorithms and data structures. This arrangement makes it easier to understand a large applications program as a whole. Moreover, unlike simple data types, ADTs provide the flexibility necessary to make it convenient to change or improve the fundamental data structures and algorithms in the system. Most important, the ADT interface defines a contract between users and implementors that provides a precise means of communicating what each can expect of the other.

We examine ADTs in detail in this chapter because they also play an important role in the study of data structures and algorithms. Indeed, the essential motivation behind the development of nearly all the algorithms that we consider in this book is to provide efficient implementations of the basic operations for certain fundamental ADTs that play a critical role in many computational tasks. Designing an ADT is only the first step in meeting the needs of applications programs—we also need to develop viable implementations of the associated operations and underlying data structures that enable them. Those tasks are the topic of this book. Moreover, we use abstract models directly to develop and to compare the performance characteristics of algorithms and data structures, as in the example in Chapter 1: Typically, we develop an applications program that uses an ADT to solve a problem, then develop multiple implementations of the ADT and compare their effectiveness. In this chapter, we consider this general process in detail, with numerous examples.

C programmers use data types and ADTs regularly. At a low level, when we process integers using only the operations provided by C for integers, we are essentially using a system-defined abstraction for integers. The integers could be represented and the operations implemented some other way on some new machine, but a program that uses only the operations specified for integers will work properly on the new machine. In this case, the various C operations for integers constitute the interface, our programs are the clients, and the system hardware and software provide the implementation. Often, the data types are sufficiently abstract that we can move to a new machine with, say, different representations for integers or floating point numbers, without having to change programs (though this ideal is not achieved as often as we would like).

At a higher level, as we have seen, C programmers often define interfaces in the form of .h files that describe a set of operations on some data structure, with implementations in some independent .c file. This arrangement provides a contract between user and implementor, and is the basis for the standard libraries that are found in C programming environments. However, many such libraries comprise operations on a particular data structure, and therefore constitute data types, but not *abstract* data types. For example, the C string library is not an ADT because programs that use strings know how strings are represented (arrays of characters) and typically access them directly via array indexing or pointer arithmetic. We could not switch, for example, to a linked-list representation of strings without changing the client programs. The memory-allocation interface and implementation for linked lists that we considered in Sections 3.4 and 3.5 has this same property. By contrast, ADTs allow us to develop implementations that not only use different implementations of the operations, but also involve different underlying data structures. Again, the key distinction that characterizes ADTs is the requirement that the data type be accessed *only* through the interface.

We shall see many examples of data types that *are* abstract throughout this chapter. After we have developed a feel for the concept, we shall return to a discussion of philosophical and practical implications, at the end of the chapter.

4.1 Abstract Objects and Collections of Objects

The data structures that we use in applications often contain a great deal of information of various types, and certain pieces of information may belong to multiple independent data structures. For example, a file of personnel data may contain records with names, addresses, and various other pieces of information about employees; and each record may need to belong to one data structure for searching for particular employees, to another data structure for answering statistical queries, and so forth.

Despite this diversity and complexity, a large class of computing applications involve generic manipulation of data objects, and need access to the information associated with them for a limited number of specific reasons. Many of the manipulations that are required are

a natural outgrowth of basic computational procedures, so they are needed in a broad variety of applications. Many of the fundamental algorithms that we consider in this book can be applied effectively to the task of building a layer of abstraction that can provide client programs with the ability to perform such manipulations efficiently. Thus, we shall consider in detail numerous ADTs that are associated with such manipulations. They define various operations on collections of abstract objects, independent of the type of the object.

We have discussed the use of simple data types in order to write code that does not depend on object types, in Chapter 3, where we used `typedef` to specify the type of our data items. This approach allows us to use the same code for, say, integers and floating-point numbers, just by changing the `typedef`. With pointers, the object types can be arbitrarily complex. When we use this approach, we are making implicit assumptions about the operations that we perform on the objects, and we are not hiding the data representation from our client programs. ADTs provide a way for us to make explicit any assumptions about the operations that we perform on data objects.

We will consider a general mechanism for the purpose of building ADTs for generic data objects in detail in Section 4.8. It is based on having the interface defined in a file named `Item.h`, which provides us with the ability to declare variables of type `Item`, and to use these variables in assignment statements, as function arguments, and as function return values. In the interface, we explicitly define any operations that our algorithms need to perform on generic objects. The mechanism that we shall consider allows us to do all this without providing any information about the data representation to client programs, thus giving us a true ADT.

For many applications, however, the different types of generic objects that we want to consider are simple and similar, and it is essential that the implementations be as efficient as possible, so we often use simple data types, not true ADTs. Specifically, we often use `Item.h` files that describe the objects themselves, not an interface. Most often, this description consists of a `typedef` to define the data type and a few macros to define the operations. For example, for an application where the only operation that we perform on the data (beyond the generic ones enabled by the `typedef`) is `eq` (test whether

two items are the same), we would use an `Item.h` file comprising the two lines of code:

```
typedef int Item  
#define eq(A, B) (A == B) .
```

Any client program with the line `#include Item.h` can use `eq` to test whether two items are equal (as well as using items in declarations, assignment statements, and function arguments and return values) in the code implementing some algorithm. Then we could use that same client program for strings, for example, by changing `Item.h` to

```
typedef char* Item;  
#define eq(A, B) (strcmp(A, B) == 0) .
```

This arrangement does not constitute the use of an ADT because the particular data representation is freely available to any program that includes `Item.h`. We typically would add macros or function calls for other simple operations on items (for example to print them, read them, or set them to random values). We adopt the convention in our client programs that we use items *as though* they were defined in an ADT, to allow us to leave the types of our basic objects unspecified in our code without any performance penalty. To use a true ADT for such a purpose would be overkill for many applications, but we shall discuss the possibility of doing so in Section 4.8, after we have seen many other examples. In principle, we can apply the technique for arbitrarily complicated data types, although the more complicated the type, the more likely we are to consider the use of a true ADT.

Having settled on some method for implementing data types for generic objects, we can move on to consider *collections* of objects. Many of the data structures and algorithms that we consider in this book are used to implement fundamental ADTs comprising collections of abstract objects, built up from the following two operations:

- *insert* a new object into the collection.
- *delete* an object from the collection.

We refer to such ADTs as *generalized queues*. For convenience, we also typically include explicit operations to *initialize* the data structure and to *count* the number of items in the data structure (or just to test whether it is empty). Alternatively, we could encompass these operations within *insert* and *delete* by defining appropriate return values.

We also might wish to *destroy* the data structure or to *copy* it; we shall discuss such operations in Section 4.8.

When we *insert* an object, our intent is clear, but which object do we get when we *delete* an object from the collection? Different ADTs for collections of objects are characterized by different criteria for deciding which object to remove for the *delete* operation and by different conventions associated with the various criteria. Moreover, we shall encounter a number of other natural operations beyond *insert* and *delete*. Many of the algorithms and data structures that we consider in this book were designed to support efficient implementation of various subsets of these operations, for various different *delete* criteria and other conventions. These ADTs are conceptually simple, used widely, and lie at the core of a great many computational tasks, so they deserve the careful attention that we pay them.

We consider several of these fundamental data structures, their properties, and examples of their application while at the same time using them as examples to illustrate the basic mechanisms that we use to develop ADTs. In Section 4.2, we consider the *pushdown stack*, where the rule for removing an object is to remove the one that was most recently inserted. We consider applications of stacks in Section 4.3, and implementations in Section 4.4, including a specific approach to keeping the applications and implementations separate. Following our discussion of stacks, we step back to consider the process of creating a new ADT, in the context of the union–find abstraction for the connectivity problem that we considered in Chapter 1. Following that, we return to collections of abstract objects, to consider FIFO queues and generalized queues (which differ from stacks on the abstract level only in that they involve using a different rule to remove items) and generalized queues where we disallow duplicate items.

As we saw in Chapter 3, arrays and linked lists provide basic mechanisms that allow us to *insert* and *delete* specified items. Indeed, linked lists and arrays are the underlying data structures for several of the implementations of generalized queues that we consider. As we know, the cost of insertion and deletion is dependent on the specific structure that we use and the specific item being inserted or deleted. For a given ADT, our challenge is to choose a data structure that allows us to perform the required operations efficiently. In this chapter, we examine in detail several examples of ADTs for which linked lists and

arrays provide appropriate solutions. ADTs that support more powerful operations require more sophisticated implementations, which are the prime impetus for many of the algorithms that we consider in this book.

Data types comprising collections of abstract objects (generalized queues) are a central object of study in computer science because they directly support a fundamental paradigm of computation. For a great many computations, we find ourselves in the position of having many objects with which to work, but being able to process only one object at a time. Therefore, we need to save the others while processing that one. This processing might involve examining some of the objects already saved away or adding more to the collection, but operations of saving the objects away and retrieving them according to some criterion are the basis of the computation. Many classical data structures and algorithms fit this mold, as we shall see.

Exercises

► 4.1 Give a definition for `Item` and `eq` that might be used for floating-point numbers, where two floating-point numbers are considered to be equal if the absolute value of their difference divided by the larger (in absolute value) of the two numbers is less than 10^{-6} .

► 4.2 Give a definition for `Item` and `eq` that might be used for points in the plane (see Section 3.1).

4.3 Add a macro `ITEMshow` to the generic object type definitions for integers and strings described in the text. Your macro should print the value of the item on standard output.

► 4.4 Give definitions for `Item` and `ITEMshow` (see Exercise 4.3) that might be used in programs that process playing cards.

4.5 Rewrite Program 3.1 to use a generic object type in a file `Item.h`. Your object type should include `ITEMshow` (see Exercise 4.3) and `ITEMrand`, so that the program can be used for any type of number for which `+` and `/` are defined.

4.2 Pushdown Stack ADT

Of the data types that support *insert* and *delete* for collections of objects, the most important is called the *pushdown stack*.

A stack operates somewhat like a busy professor's "in" box: work piles up in a stack, and whenever the professor has a chance to get some work done, it comes off the top. A student's paper might

well get stuck at the bottom of the stack for a day or two, but a conscientious professor might manage to get the stack emptied at the end of the week. As we shall see, computer programs are naturally organized in this way. They frequently postpone some tasks while doing others; moreover, they frequently need to return to the most recently postponed task first. Thus, pushdown stacks appear as the fundamental data structure for many algorithms.

L	L
A	LA
*	A L
S	LS
T	LST
I	LBSTI
*	LSI
N	LSI N
*	LSI
F	LSTF
I	LSTFI
R	LSTFIR
*	LSFI
S	LSFIS
T	LSFIST
*	LSFIS
*	LSFI
O	LSFI O
U	LSFI OU
*	LSFI O
T	LSFI OT
*	LSFI O
*	LSFI
I	LSF
*	LS
*	S
*	L

Figure 4.1
Pushdown stack (LIFO queue)
example

This list shows the result of the sequence of operations in the left column (top to bottom), where a letter denotes push and an asterisk denotes pop. Each line displays the operation, the letter popped for pop operations, and the contents of the stack after the operation, in order from least recently inserted to most recently inserted, left to right.

Definition 4.2 A pushdown stack is an ADT that comprises two basic operations: insert (push) a new item, and delete (pop) the item that was most recently inserted.

That is, when we speak of a *pushdown stack ADT*, we are referring to a description of the *push* and *pop* operations that is sufficiently well specified that a client program can make use of them, and to some implementation of the operations enforcing the rule that characterizes a pushdown stack: items are removed according to a *last-in, first-out (LIFO)* discipline. In the simplest case, which we use most often, both client and implementation refer to just a single stack (that is, the “set of values” in the data type is just that one stack); in Section 4.8, we shall see how to build an ADT that supports multiple stacks.

Figure 4.1 shows how a sample stack evolves through a series of *push* and *pop* operations. Each *push* increases the size of the stack by 1 and each *pop* decreases the size of the stack by 1. In the figure, the items in the stack are listed in the order that they are put on the stack, so that it is clear that the rightmost item in the list is the one at the top of the stack—the item that is to be returned if the next operation is *pop*. In an implementation, we are free to organize the items any way that we want, as long as we allow clients to maintain the illusion that the items are organized in this way.

To write programs that use the pushdown stack abstraction, we need first to define the interface. In C, one way to do so is to declare the four operations that client programs may use, as illustrated in Program 4.1. We keep these declarations in a file STACK.h that is referenced as an include file in client programs and implementations.

Furthermore, we expect that there is no other connection between client programs and implementations. We have already seen, in Chapter 1, the value of identifying the abstract operations on which a computation is based. We are now considering a mechanism that

Program 4.1 Pushdown-stack ADT interface

This interface defines the basic operations that define a pushdown stack. We assume that the four declarations here are in a file `STACK.h`, which is referenced as an include file by client programs that use these functions and implementations that provide their code; and that both clients and implementations define `Item`, perhaps by including an `Item.h` file (which may have a `typedef` or which may define a more general interface). The argument to `STACKinit` specifies the maximum number of elements expected on the stack.

```
void STACKinit(int);
int STACKempty();
void STACKpush(Item);
Item STACKpop();
```

allows us to write programs that use these abstract operations. To enforce the abstraction, we hide the data structure and the implementation from the client. In Section 4.3, we consider examples of client programs that use the stack abstraction; in Section 4.4, we consider implementations.

In an ADT, the purpose of the interface is to serve as a contract between client and implementation. The function declarations ensure that the calls in the client program and the function definitions in the implementation match, but the interface otherwise contains no information about how the functions are to be implemented, or even how they are to behave. How can we explain what a stack is to a client program? For simple structures like stacks, one possibility is to exhibit the code, but this solution is clearly not effective in general. Most often, programmers resort to English-language descriptions, in documentation that accompanies the code.

A rigorous treatment of this situation requires a full description, in some formal mathematical notation, of how the functions are supposed to behave. Such a description is sometimes called a *specification*. Developing a specification is generally a challenging task. It has to describe *any* program that implements the functions in a mathematical metalanguage, whereas we are used to specifying the behavior of functions with code written in a programming language. In practice, we describe behavior in English-language descriptions. Before getting

drawn further into epistemological issues, we move on. In this book, we give detailed examples, English-language descriptions, and multiple implementations for most of the ADTs that we consider.

To emphasize that our specification of the pushdown stack ADT is sufficient information for us to write meaningful client programs, we consider, in Section 4.3, two client programs that use pushdown stacks, before considering any implementation.

Exercises

► 4.6 A letter means *push* and an asterisk means *pop* in the sequence

E A S * Y * Q U E * * * S T * * * I O * N * * *.

Give the sequence of values returned by the *pop* operations.

4.7 Using the conventions of Exercise 4.6, give a way to insert asterisks in the sequence E A S Y so that the sequence of values returned by the *pop* operations is (i) E A S Y ; (ii) Y S A E ; (iii) A S Y E ; (iv) A Y E S ; or, in each instance, prove that no such sequence exists.

•• 4.8 Given two sequences, give an algorithm for determining whether or not asterisks can be added to make the first produce the second, when interpreted as a sequence of stack operations in the sense of Exercise 4.7.

4.3 Examples of Stack ADT Clients

We shall see a great many applications of stacks in the chapters that follow. As an introductory example, we now consider the use of stacks for evaluating arithmetic expressions. For example, suppose that we need to find the value of a simple arithmetic expression involving multiplication and addition of integers, such as

$$5 * (((9 + 8) * (4 * 6)) + 7)$$

The calculation involves saving intermediate results: For example, if we calculate $9 + 8$ first, then we have to save the result 17 while, say, we compute $4 * 6$. A pushdown stack is the ideal mechanism for saving intermediate results in such a calculation.

We begin by considering a simpler problem, where the expression that we need to evaluate is in a form where each operator appears *after* its two arguments, rather than between them. As we shall see, any arithmetic expression can be arranged in this form, which is called

postfix, by contrast with *infix*, the customary way of writing arithmetic expressions. The postfix representation of the expression in the previous paragraph is

$$5 \ 9 \ 8 \ + \ 4 \ 6 \ * \ * \ 7 \ + \ *$$

The reverse of postfix is called *prefix*, or *Polish notation* (because it was invented by the Polish logician Lukasiewicz).

In infix, we need parentheses to distinguish, for example,

$$5 * ((9 + 8) * (4 * 6)) + 7)$$

from

$$((5 * 9) + 8) * ((4 * 6) + 7)$$

but parentheses are unnecessary in postfix (or prefix). To see why, we can consider the following process for converting a postfix expression to an infix expression: We replace all occurrences of two operands followed by an operator by their infix equivalent, with parentheses, to indicate that the result can be considered to be an operand. That is, we replace any occurrence of $a \ b \ *$ and $a \ b \ +$ by $(a * b)$ and $(a + b)$, respectively. Then, we perform the same transformation on the resulting expression, continuing until all the operators have been processed. For our example, the transformation happens as follows:

$$\begin{aligned} & 5 \ 9 \ 8 \ + \ 4 \ 6 \ * \ * \ 7 \ + \ * \\ & 5 (9 + 8) (4 * 6) * 7 + * \\ & 5 ((9 + 8) * (4 * 6)) 7 + * \\ & 5 (((9 + 8) * (4 * 6)) + 7) * \\ & (5 * (((9 + 8) * (4 * 6)) + 7)) \end{aligned}$$

We can determine the operands associated with any operator in the postfix expression in this way, so no parentheses are necessary.

Alternatively, with the aid of a stack, we can actually perform the operations and evaluate any postfix expression, as illustrated in Figure 4.2. Moving from left to right, we interpret each operand as the command to “push the operand onto the stack,” and each operator as the commands to “pop the two operands from the stack, perform the operation, and push the result.” Program 4.2 is a C implementation of this process.

Postfix notation and an associated pushdown stack give us a natural way to organize a series of computational procedures. Some calculators and some computing languages explicitly base their method

5	5
9	5 9
8	5 9 8
+	5 17
4	5 17 4
6	5 17 4 6
*	5 17 24
*	5 408
7	5 408 7
+	5 415
*	2075

Figure 4.2
Evaluation of a postfix expression

This sequence shows the use of a stack to evaluate the postfix expression $5 \ 9 \ 8 \ + \ 4 \ 6 \ * \ * \ 7 \ + \ *$. Proceeding from left to right through the expression, if we encounter a number, we push it on the stack; and if we encounter an operator, we push the result of applying the operator to the top two numbers on the stack.

Program 4.2 Postfix-expression evaluation

This pushdown-stack client reads any postfix expression involving multiplication and addition of integers, then evaluates the expression and prints the computed result.

When we encounter operands, we push them on the stack; when we encounter operators, we pop the top two entries from the stack and push the result of applying the operator to them. The order in which the two `STACKpop()` operations are performed in the expressions in this code is unspecified in C, so the code for noncommutative operators such as subtraction or division would be slightly more complicated.

The program assumes that at least one blank follows each integer, but otherwise does not check the legality of the input at all. The final `if` statement and the `while` loop perform a calculation similar to the C `atoi` function, which converts integers from ASCII strings to integers for calculation. When we encounter a new digit, we multiply the accumulated result by 10 and add the digit.

The stack contains integers—that is, we assume that `Item` is defined to be `int` in `Item.h`, and that `Item.h` is also included in the stack implementation (see, for example, Program 4.4).

```
#include <stdio.h>
#include <string.h>
#include "Item.h"
#include "STACK.h"
main(int argc, char *argv[])
{
    char *a = argv[1]; int i, N = strlen(a);
    STACKinit(N);
    for (i = 0; i < N; i++)
    {
        if (a[i] == '+')
            STACKpush(STACKpop() + STACKpop());
        if (a[i] == '*')
            STACKpush(STACKpop() * STACKpop());
        if ((a[i] >= '0') && (a[i] <= '9'))
            STACKpush(0);
        while ((a[i] >= '0') && (a[i] <= '9'))
            STACKpush(10 * STACKpop() + (a[i++]-'0'));
    }
    printf("%d \n", STACKpop());
}
```

of calculation on postfix and stack operations—every operation pops its arguments from the stack and returns its results to the stack.

One example of such a language is the PostScript language, which is used to print this book. It is a complete programming language where programs are written in postfix and are interpreted with the aid of an internal stack, precisely as in Program 4.2. Although we cannot cover all the aspects of the language here (*see reference section*), it is sufficiently simple that we can study some actual programs, to appreciate the utility of the postfix notation and the pushdown-stack abstraction. For example, the string

```
5 9 8 add 4 6 mul mul 7 add mul
```

is a PostScript program! Programs in PostScript consist of operators (such as `add` and `mul`) and operands (such as integers). As we did in Program 4.2 we interpret a program by reading it from left to right: If we encounter an operand, we push it onto the stack; if we encounter an operator, we pop its operands (if any) from the stack and push the result (if any). Thus, the execution of this program is fully described by Figure 4.2: The program leaves the value 2075 on the stack.

PostScript has a number of primitive functions that serve as instructions to an abstract plotting device; we can also define our own functions. These functions are invoked with arguments on the stack in the same way as any other function. For example, the PostScript code

```
0 0 moveto 144 hill 0 72 moveto 72 hill stroke
```

corresponds to the sequence of actions “call `moveto` with arguments 0 and 0, then call `hill` with argument 144,” and so forth. Some operators refer directly to the stack itself. For example the operator `dup` duplicates the entry at the top of the stack so, for example, the PostScript code

```
144 dup 0 rlineto 60 rotate dup 0 rlineto
```

corresponds to the sequence of actions “call `rlineto` with arguments 144 and 0, then call `rotate` with argument 60, then call `rlineto` with arguments 144 and 0,” and so forth. The PostScript program in Figure 4.3 defines and uses the function `hill`. Functions in PostScript are like macros: The sequence `/hill { A } def` makes the name `hill` equivalent to the operator sequence inside the braces. Figure 4.3 is an example of a PostScript program that defines a function and draws a simple diagram.



```
/hill {
    dup 0 rlineto
    60 rotate
    dup 0 rlineto
    -120 rotate
    dup 0 rlineto
    60 rotate
    dup 0 rlineto
    pop
} def
0 0 moveto
144 hill
0 72 moveto
72 hill
stroke
```

Figure 4.3
Sample PostScript program

The diagram at the top was drawn by the PostScript program below it. The program is a postfix expression that uses the built-in functions `moveto`, `rlineto`, `rotate`, `stroke` and `dup`; and the user-defined function `hill` (see text). The graphics commands are instructions to a plotting device: `moveto` instructs that device to go to the specified position on the page (coordinates are in points, which are 1/72 inch); `rlineto` instructs it to move to the specified position in coordinates relative to its current position, adding the line it makes to its current path; `rotate` instructs it to turn left the specified number of degrees; and `stroke` instructs it to draw the path that it has traced.

In the present context, our interest in PostScript is that this widely used programming language is based on the pushdown-stack abstraction. Indeed, many computers implement basic stack operations in hardware because they naturally implement a function-call mechanism: Save the current environment on entry to a function by pushing information onto a stack; restore the environment on exit by using information popped from the stack. As we see in Chapter 5, this connection between pushdown stacks and programs organized as functions that call functions is an essential paradigm of computation.

Returning to our original problem, we can also use a pushdown stack to convert fully parenthesized arithmetic expressions from infix to postfix, as illustrated in Figure 4.4. For this computation, we push the *operators* onto a stack, and simply pass the *operands* through to the output. Then, each right parenthesis indicates that both arguments for the last operator have been output, so the operator itself can be popped and output.

Program 4.3 is an implementation of this process. Note that arguments appear in the postfix expression in the same order as in the infix expression. It is also amusing to note that the left parentheses are not needed in the infix expression. The left parentheses would be required, however, if we could have operators that take differing numbers of operands (see Exercise 4.11).

In addition to providing two different examples of the use of the pushdown-stack abstraction, the entire algorithm that we have developed in this section for evaluating infix expressions is itself an exercise in abstraction. First, we convert the input to an intermediate representation (postfix). Second, we simulate the operation of an abstract stack-based machine to interpret and evaluate the expression. This same schema is followed by many modern programming-language translators, for efficiency and portability: The problem of compiling a C program for a particular computer is broken into two tasks centered around an intermediate representation, so that the problem of translating the program is separated from the problem of executing that program, just as we have done in this section. We shall see a related, but different, intermediate representation in Section 5.7.

This application also illustrates that ADTs do have their limitations. For example, the conventions that we have discussed do not provide an easy way to combine Programs 4.2 and 4.3 into a single

(
5	5
*	*
(
(
(
9	9
+	*
8	8
)	+
*	*
(
4	4
*	*
6	6
)	*
)	*
+	*
7	7
)	+
)	*

Figure 4.4
Conversion of an infix expression to postfix

This sequence shows the use of a stack to convert the infix expression $(5*((9+8)*(4*6))+7)$ to its postfix form 5 9 8 + 4 6 * * 7 + *. We proceed from left to right through the expression: If we encounter a number, we write it to the output; if we encounter a left parenthesis, we ignore it; if we encounter an operator, we push it on the stack; and if we encounter a right parenthesis, we write the operator at the top of the stack to the output.

Program 4.3 Infix-to-postfix conversion

This program is another example of a pushdown-stack client. In this case, the stack contains characters—we assume that `Item` is defined to be `char` (that is, we use a different `Item.h` file than for Program 4.2). To convert $(A+B)$ to the postfix form $AB+$, we ignore the left parenthesis, convert A to postfix, save the $+$ on the stack, convert B to postfix, then, on encountering the right parenthesis, pop the stack and output the $+$.

```
#include <stdio.h>
#include <string.h>
#include "Item.h"
#include "STACK.h"
main(int argc, char *argv[])
{ char *a = argv[1]; int i, N = strlen(a);
  STACKinit(N);
  for (i = 0; i < N; i++)
  {
    if (a[i] == ')')
      printf("%c ", STACKpop());
    if ((a[i] == '+') || (a[i] == '*'))
      STACKpush(a[i]);
    if ((a[i] >= '0') && (a[i] <= '9'))
      printf("%c ", a[i]);
  }
  printf("\n");
}
```

program, using the same pushdown-stack ADT for both. Not only do we need two different stacks, but also one of the stacks holds single characters (operators), whereas the other holds numbers. To better appreciate the problem, suppose that the numbers are, say, floating-point numbers, rather than integers. Using a general mechanism to allow sharing the same implementation between both stacks (an extension of the approach that we consider in Section 4.8) is likely to be more trouble than simply using two different stacks (see Exercise 4.16). In fact, as we shall see, this solution might be the approach of choice, because different implementations may have different performance characteristics, so we might not wish to decide a priori that one ADT will serve

both purposes. Indeed, our main focus is on the implementations and their performance, and we turn now to those topics for pushdown stacks.

Exercises

► 4.9 Convert to postfix the expression

$(5 * ((9 * 8) + (7 * (4 + 6))))$.

► 4.10 Give, in the same manner as Figure 4.2, the contents of the stack as the following expression is evaluated by Program 4.2

$5\ 9\ *\ 8\ 7\ 4\ 6\ +\ *\ 2\ 1\ 3\ *\ +\ *\ +\ *$.

► 4.11 Extend Programs 4.2 and 4.3 to include the $-$ (subtract) and $/$ (divide) operations.

► 4.12 Extend your solution to Exercise 4.11 to include the unary operators $-$ (negation) and $\$$ (square root). Also, modify the abstract stack machine in Program 4.2 to use floating point. For example, given the expression

$(-(-1) + \$((-1) * (-1)-(4 * (-1))))/2$

your program should print the value 1.618034.

► 4.13 Write a PostScript program that draws this figure:



• 4.14 Prove by induction that Program 4.2 correctly evaluates any postfix expression.

○ 4.15 Write a program that converts a postfix expression to infix, using a pushdown stack.

• 4.16 Combine Program 4.2 and Program 4.3 into a single module that uses two different stack ADTs: a stack of integers and a stack of operators.

● 4.17 Implement a compiler and interpreter for a programming language where each program consists of a single arithmetic expression preceded by a sequence of assignment statements with arithmetic expressions involving integers and variables named with single lower-case characters. For example, given the input

```
(x = 1)
(y = (x + 1))
(((x + y) * 3) + (4 * x))
```

your program should print the value 13.

4.4 Stack ADT Implementations

In this section, we consider two implementations of the stack ADT: one using arrays and one using linked lists. The implementations are

both straightforward applications of the basic tools that we covered in Chapter 3. They differ only, we expect, in their performance characteristics.

If we use an array to represent the stack, each of the functions declared in Program 4.1 is trivial to implement, as shown in Program 4.4. We put the items in the array precisely as diagrammed in Figure 4.1, keeping track of the index of the top of the stack. Doing the *push* operation amounts to storing the item in the array position indicated by the top-of-stack index, then incrementing the index; doing the *pop* operation amounts to decrementing the index, then returning the item that it designates. The *initialize* operation involves allocating an array of the indicated size, and the *test if empty* operation involves checking whether the index is 0. Compiled together with a client program such as Program 4.2 or Program 4.3, this implementation provides an efficient and effective pushdown stack.

We know one potential drawback to using an array representation: As is usual with data structures based on arrays, we need to know the maximum size of the array before using it, so that we can allocate memory for it. In this implementation, we make that information an argument to the function that implements *initialize*. This constraint is an artifact of our choice to use an array implementation; it is not an essential part of the stack ADT. We may have no easy way to estimate the maximum number of elements that our program will be putting on the stack: If we choose an arbitrarily high value, this implementation will make inefficient use of space, and that may be undesirable in an application where space is a precious resource. If we choose too small a value, our program might not work at all. By using an ADT, we make it possible to consider other alternatives, in other implementations, without changing any client program.

For example, to allow the stack to grow and shrink gracefully, we may wish to consider using a linked list, as in the implementation in Program 4.5. In this program, we keep the stack in reverse order from the array implementation, from most recently inserted element to least recently inserted element, to make the basic stack operations easier to implement, as illustrated in Figure 4.5. To *pop*, we remove the node from the front of the list and return its item; to *push*, we create a new node and add it to the front of the list. Because all linked-list operations are at the beginning of the list, we do not need to use a

Program 4.4 Array implementation of a pushdown stack

When there are N items in the stack, this implementation keeps them in $s[0], \dots, s[N-1]$; in order from least recently inserted to most recently inserted. The top of the stack (the position where the next item to be pushed will go) is $s[N]$. The client program passes the maximum number of items expected on the stack as the argument to `STACKinit`, which allocates an array of that size, but this code does not check for errors such as pushing onto a full stack (or popping an empty one).

```
#include <stdlib.h>
#include "Item.h"
#include "STACK.h"
static Item *s;
static int N;
void STACKinit(int maxN)
{ s = malloc(maxN*sizeof(Item)); N = 0; }
int STACKempty()
{ return N == 0; }
void STACKpush(Item item)
{ s[N++] = item; }
Item STACKpop()
{ return s[--N]; }
```

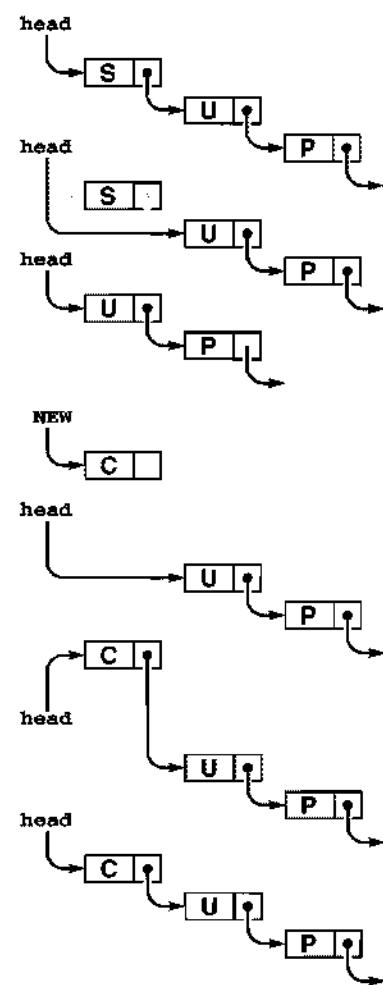
head node. This implementation does not need to use the argument to `STACKinit`.

Programs 4.4 and 4.5 are two different implementations for the same ADT. We can substitute one for the other without making *any* changes in client programs such as the ones that we examined in Section 4.3. They differ in only their performance characteristics—the time and space that they use. For example, the list implementation uses more time for push and pop operations, to allocate memory for each *push* and deallocate memory for each *pop*. If we have an application where we perform these operations a huge number of times, we might prefer the array implementation. On the other hand, the array implementation uses the amount of space necessary to hold the maximum number of items expected throughout the computation, while the list implementation uses space proportional to the number of items,

Program 4.5 Linked-list implementation of a pushdown stack

This code implements the stack ADT as illustrated in Figure 4.5. It uses an auxiliary function `NEW` to allocate the memory for a node, set its fields from the function arguments, and return a link to the node.

```
#include <stdlib.h>
#include "Item.h"
typedef struct STACKnode* link;
struct STACKnode { Item item; link next; };
static link head;
link NEW(Item item, link next)
{ link x = malloc(sizeof *x);
  x->item = item; x->next = next;
  return x;
}
void STACKinit(int maxN)
{ head = NULL; }
int STACKempty()
{ return head == NULL; }
STACKpush(Item item)
{ head = NEW(item, head); }
Item STACKpop()
{ Item item = head->item;
  link t = head->next;
  free(head); head = t;
  return item;
}
```



but always uses extra space for one link per item. If we need a huge stack that is usually nearly full, we might prefer the array implementation; if we have a stack whose size varies dramatically and other data structures that could make use of the space not being used when the stack has only a few items in it, we might prefer the list implementation.

These same considerations about space usage hold for many ADT implementations, as we shall see throughout the book. We often are in the position of choosing between the ability to access any item quickly but having to predict the maximum number of items needed ahead of time (in an array implementation) and the flexibility of always using

Figure 4.5
Linked-list pushdown stack

The stack is represented by a pointer `head`, which points to the first (most recently inserted) item. To pop the stack (top), we remove the item at the front of the list, by setting `head` from its link. To push a new item onto the stack (bottom), we link it in at the beginning by setting its link field to `head`, then setting `head` to point to it.

space proportional to the number of items in use while giving up the ability to access every item quickly (in a linked-list implementation).

Beyond basic space-usage considerations, we normally are most interested in performance differences among ADT implementations that relate to running time. In this case, there is little difference between the two implementations that we have considered.

Property 4.1 *We can implement the push and pop operations for the pushdown stack ADT in constant time, using either arrays or linked lists.*

This fact follows immediately from inspection of Programs 4.4 and 4.5.

■ That the stack items are kept in different orders in the array and the linked-list implementations is of no concern to the client program. The implementations are free to use any data structure whatever, as long as they maintain the illusion of an abstract pushdown stack. In both cases, the implementations are able to create the illusion of an efficient abstract entity that can perform the requisite operations with just a few machine instructions. Throughout this book, our goal is to find data structures and efficient implementations for other important ADTs.

The linked-list implementation supports the illusion of a stack that can grow without bound. Such a stack is impossible in practical terms: at some point, `malloc` will return `NULL` when the request for more memory cannot be satisfied. It is also possible to arrange for an array-based stack to grow dynamically, by doubling the size of the array when the stack becomes half full, and halving the size of the array when the stack becomes half empty. We leave the details of this implementation as an exercise in Chapter 14, where we consider the process in detail for a more advanced application.

Exercises

▷ 4.18 Give the contents of `s[0], …, s[4]` after the execution of the operations illustrated in Figure 4.1, using Program 4.4.

○ 4.19 Suppose that you change the pushdown-stack interface to replace `test if empty` by `count`, which should return the number of items currently in the data structure. Provide implementations for `count` for the array representation (Program 4.4) and the linked-list representation (Program 4.5).

4.20 Modify the array-based pushdown-stack implementation in the text (Program 4.4) to call a function `STACKerror` if the client attempts to `pop` when the stack is empty or to `push` when the stack is full.

- 4.21 Modify the linked-list-based pushdown-stack implementation in the text (Program 4.5) to call a function `STACKerror` if the client attempts to `pop` when the stack is empty or if there is no memory available from `malloc` for a `push`.
- 4.22 Modify the linked-list-based pushdown-stack implementation in the text (Program 4.5) to use an array of indices to implement the list (see Figure 3.4).
- 4.23 Write a linked-list-based pushdown-stack implementation that keeps items on the list in order from least recently inserted to most recently inserted. You will need to use a doubly linked list.
- 4.24 Develop an ADT that provides clients with two different pushdown stacks. Use an array implementation. Keep one stack at the beginning of the array and the other at the end. (If the client program is such that one stack grows while the other one shrinks, this implementation uses less space than other alternatives.)
 - 4.25 Implement an infix-expression-evaluation function for integers that includes Programs 4.2 and 4.3, using your ADT from Exercise 4.24.

4.5 Creation of a New ADT

Sections 4.2 through 4.4 present a complete example of C code that captures one of our most important abstractions: the pushdown stack. The *interface* in Section 4.2 defines the basic operations; *client programs* such as those in Section 4.3 can use those operations without dependence on how the operations are implemented; and *implementations* such as those in Section 4.4 provide the necessary concrete representation and program code to realize the abstraction.

To design a new ADT, we often enter into the following process. Starting with the task of developing a client program to solve an applications problem, we identify operations that seem crucial: What would we *like* to be able to do with our data? Then, we define an interface and write client code to test the hypothesis that the existence of the ADT would make it easier for us to implement the client program. Next, we consider the idea of whether or not we *can* implement the operations in the ADT with reasonable efficiency. If we cannot, we perhaps can seek to understand the source of the inefficiency and to modify the interface to include operations that are better suited to efficient implementation. These modifications affect the client program, and we modify it accordingly. After a few iterations, we have a

Program 4.6 Equivalence-relations ADT interface

The ADT interface mechanism makes it convenient for us to encode precisely our decision to consider the connectivity algorithm in terms of three abstract operations: *initialize*, *find* whether two nodes are connected, and perform a *union* operation to consider them connected henceforth.

```
void UFinit(int);
int UFFind(int, int);
void UFFunion(int, int);
```

working client program and a working implementation, so we freeze the interface: We adopt a policy of not changing it. At this moment, the development of client programs and the development of implementations are separable: We can write other client programs that use the same ADT (perhaps we write some driver programs that allow us to test the ADT), we can write other implementations, and we can compare the performance of multiple implementations.

In other situations, we might define the ADT first. This approach might involve asking questions such as these: What basic operations would client programs want to perform on the data at hand? Which operations do we know how to implement efficiently? After we develop an implementation, we might test its efficacy on client programs. We might modify the interface and do more tests, before eventually freezing the interface.

In Chapter 1, we considered a detailed example where thinking on an abstract level helped us to find an efficient algorithm for solving a complex problem. We consider next the use of the general approach that we are discussing in this chapter to encapsulate the specific abstract operations that we exploited in Chapter 1.

Program 4.6 defines the interface, in terms of two operations (in addition to *initialize*) that seem to characterize the algorithms that we considered in Chapter 1 for connectivity, at a high abstract level. Whatever the underlying algorithms and data structures, we want to be able to check whether or not two nodes are known to be connected, and to declare that two nodes are connected.

Program 4.7 is a client program that uses the ADT defined in the interface of Program 4.6 to solve the connectivity problem. One benefit

Program 4.7 Equivalence-relations ADT client

The ADT of Program 4.6 separates the connectivity algorithm from the union–find implementation, making that algorithm more accessible.

```
#include <stdio.h>
#include "UF.h"
main(int argc, char *argv[])
{ int p, q, N = atoi(argv[1]);
  UFinit(N);
  while (scanf("%d %d", &p, &q) == 2)
    if (!UFFind(p, q))
      { UFUnion(p, q); printf(" %d %d\n", p, q); }
```

of using the ADT is that this program is easy to understand, because it is written in terms of abstractions that allow the computation to be expressed in a natural way.

Program 4.8 is an implementation of the union–find interface defined in Program 4.6 that uses a forest of trees represented by two arrays as the underlying representation of the known connectivity information, as described in Section 1.3. The different algorithms that we considered in Chapter 1 represent different implementations of this ADT, and we can test them as such without changing the client program at all.

This ADT leads to programs that are slightly less efficient than those in Chapter 1 for the connectivity application, because it does not take advantage of the property of that client that every *union* operation is immediately preceded by a *find* operation. We sometimes incur extra costs of this kind as the price of moving to a more abstract representation. In this case, there are numerous ways to remove the inefficiency, perhaps at the cost of making the interface or the implementation more complicated (see Exercise 4.27). In practice, the paths are extremely short (particularly if we use path compression), so the extra cost is likely to be negligible in this case.

The combination of Programs 4.6 through 4.8 is operationally equivalent to Program 1.3, but splitting the program into three parts is a more effective approach because it

Program 4.8 Equivalence-relations ADT implementation

This implementation of the weighted-quick-union code from Chapter 1, together with the interface of Program 4.6, packages the code in a form that makes it convenient for use in other applications. The implementation uses a local function `find`.

```
#include <stdlib.h>
#include "UF.h"
static int *id, *sz;
void UFinit(int N)
{
    int i;
    id = malloc(N*sizeof(int));
    sz = malloc(N*sizeof(int));
    for (i = 0; i < N; i++)
        { id[i] = i; sz[i] = 1; }
}
static int find(int x)
{
    int i = x;
    while (i != id[i]) i = id[i]; return i;
}
int UFind(int p, int q)
{
    return (find(p) == find(q));
}
void UFunion(int p, int q)
{
    int i = find(p), j = find(q);
    if (i == j) return;
    if (sz[i] < sz[j])
        { id[i] = j; sz[j] += sz[i]; }
    else { id[j] = i; sz[i] += sz[j]; }
}
```

- Separates the task of solving the high-level (connectivity) problem from the task of solving the low-level (union-find) problem, allowing us to work on the two problems independently
- Gives us a natural way to compare different algorithms and data structures for solving the problem
- Gives us an abstraction that we can use to build other algorithms
- Defines, through the interface, a way to check that the software is operating as expected

- Provides a mechanism that allows us to upgrade to new representations (new data structures or new algorithms) without changing the client program at all

These benefits are widely applicable to many tasks that we face when developing computer programs, so the basic tenets underlying ADTs are widely used.

Exercises

- 4.26 Modify Program 4.8 to use path compression by halving.
- 4.27 Remove the inefficiency mentioned in the text by adding an operation to Program 4.6 that combines *union* and *find*, providing an implementation in Program 4.8, and modifying Program 4.7 accordingly.
- o 4.28 Modify the interface (Program 4.6) and implementation (Program 4.8) to provide a function that will return the number of nodes known to be connected to a given node.
- 4.29 Modify Program 4.8 to use an array of structures instead of parallel arrays for the underlying data structure.

4.6 FIFO Queues and Generalized Queues

The *first-in, first-out (FIFO) queue* is another fundamental ADT that is similar to the pushdown stack, but that uses the opposite rule to decide which element to remove for *delete*. Rather than removing the most recently inserted element, we remove the element that has been in the queue the longest.

Perhaps our busy professor’s “in” box *should* operate like a FIFO queue, since the first-in, first-out order seems to be an intuitively fair way to decide what to do next. However, that professor might not ever answer the phone or get to class on time! In a stack, a memorandum can get buried at the bottom, but emergencies are handled when they arise; in a FIFO queue, we work methodically through the tasks, but each has to wait its turn.

FIFO queues are abundant in everyday life. When we wait in line to see a movie or to buy groceries, we are being processed according to a FIFO discipline. Similarly, FIFO queues are frequently used within computer systems to hold tasks that are yet to be accomplished when we want to provide services on a first-come, first-served basis. Another example, which illustrates the distinction between stacks and FIFO

F	F
I	FI
R	FIR
S	FIRS
*	FIRS
T	IRST
*	IRST
I	RSTI
N	RSTIN
*	RSTIN
S	TIN
*	TIN
F	INF
I	INF
*	NFI
R	NFIR
S	NFIRS
*	NFIRS
F	IIRS
*	IRS
T	RST
*	RST
O	STO
OUT	STOUT
*	STOUT
S	TOUT
*	TOUT
O	UT
*	UT
U	T
*	T

Figure 4.6
FIFO queue example

This list shows the result of the sequence of operations in the left column (top to bottom), where a letter denotes put and an asterisk denotes get. Each line displays the operation, the letter returned for get operations, and the contents of the queue in order from least recently inserted to most recently inserted, left to right.

Program 4.9 FIFO queue ADT interface

This interface is identical to the pushdown stack interface of Program 4.1, except for the names of the structure. The two ADTs differ only in the specification, which is not reflected in the code.

```
void QUEUEinit(int);
int QUEUEempty();
void QUEUEput(Item);
Item QUEUEget();
```

queues, is a grocery store's inventory of a perishable product. If the grocer puts new items on the front of the shelf and customers take items from the front, then we have a stack discipline, which is a problem for the grocer because items at the back of the shelf may stay there for a very long time and therefore spoil. By putting new items at the back of the shelf, the grocer ensures that the length of time any item has to stay on the shelf is limited by the length of time it takes customers to purchase the maximum number of items that fit on the shelf. This same basic principle applies to numerous similar situations.

Definition 4.3 A **FIFO queue** is an ADT that comprises two basic operations: insert (put) a new item, and delete (get) the item that was least recently inserted.

Program 4.9 is the interface for a FIFO queue ADT. This interface differs from the stack interface that we considered in Section 4.2 only in the nomenclature: to a compiler, say, the two interfaces are identical! This observation underscores the fact that the abstraction itself, which programmers normally do not define formally, is the essential component of an ADT. For large applications, which may involve scores of ADTs, the problem of defining them precisely is critical. In this book, we work with ADTs that capture essential concepts that we define in the text, but not in any formal language, other than via specific implementations. To discern the nature of ADTs, we need to consider examples of their use and to examine specific implementations.

Figure 4.6 shows how a sample FIFO queue evolves through a series of get and put operations. Each get decreases the size of the queue by 1 and each put increases the size of the queue by 1. In the figure, the items in the queue are listed in the order that they are put on

the queue, so that it is clear that the first item in the list is the one that is to be returned by the *get* operation. Again, in an implementation, we are free to organize the items any way that we want, as long as we maintain the illusion that the items are organized in this way.

To implement the FIFO queue ADT using a linked list, we keep the items in the list in order from least recently inserted to most recently inserted, as diagrammed in Figure 4.6. This order is the reverse of the order that we used for the stack implementation, but allows us to develop efficient implementations of the queue operations. We maintain two pointers into the list: one to the beginning (so that we can *get* the first element), and one to the end (so that we can *put* a new element onto the queue), as shown in Figure 4.7 and in the implementation in Program 4.10.

We can also use an array to implement a FIFO queue, although we have to exercise care to keep the running time constant for both the *put* and *get* operations. That performance goal dictates that we can not move the elements of the queue within the array, unlike what might be suggested by a literal interpretation of Figure 4.6. Accordingly, as we did with the linked-list implementation, we maintain two indices into the array: one to the beginning of the queue and one to the end of the queue. We consider the contents of the queue to be the elements between the indices. To *get* an element, we remove it from the beginning (head) of the queue and increment the head index; to *put* an element, we add it to the end (tail) of the queue and increment the tail index. A sequence of *put* and *get* operations causes the queue to appear to move through the array, as illustrated in Figure 4.8. When it hits the end of the array, we arrange for it to wrap around to the beginning. The details of this computation are in the code in Program 4.11.

Property 4.2 *We can implement the get and put operations for the FIFO queue ADT in constant time, using either arrays or linked lists.*

This fact is immediately clear when we inspect the code in Programs 4.10 and 4.11. ■

The same considerations that we discussed in Section 4.4 apply to space resources used by FIFO queues. The array representation requires that we reserve enough space for the maximum number of items expected throughout the computation, whereas the linked-list

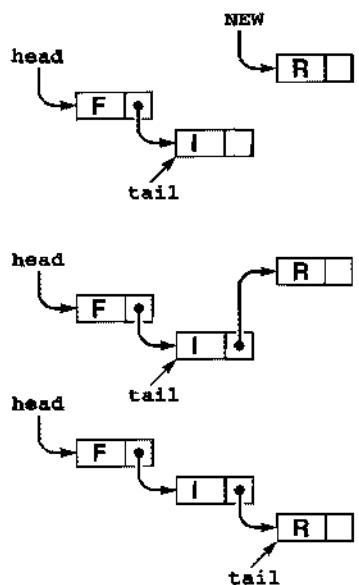


Figure 4.7
Linked-list queue

*In this linked-list representation of a queue, we insert new items at the end, so the items in the linked list are in order from least recently inserted to most recently inserted, from beginning to end. The queue is represented by two pointers *head* and *tail* which point to the first and final item, respectively. To get an item from the queue, we remove the item at the front of the list, in the same way as we did for stacks (see Figure 4.5). To put a new item onto the queue, we set the link field of the node referenced by *tail* to point to it (center), then update *tail* (bottom).*

Program 4.10 FIFO queue linked-list implementation

The difference between a FIFO queue and a pushdown stack (Program 4.5) is that new items are inserted at the end, rather than the beginning.

Accordingly, this program keeps a pointer `tail` to the last node of the list, so that the function `QUEUEput` can add a new node by linking that node to the node referenced by `tail` and then updating `tail` to point to the new node. The functions `QUEUEget`, `QUEUEinit`, and `QUEUEempty` are all identical to their counterparts for the linked-list pushdown-stack implementation of Program 4.5.

```
#include <stdlib.h>
#include "Item.h"
#include "QUEUE.h"
typedef struct QUEUENode* link;
struct QUEUENode { Item item; link next; };
static link head, tail;
link NEW(Item item, link next)
{
    link x = malloc(sizeof *x);
    x->item = item; x->next = next;
    return x;
}
void QUEUEinit(int maxN)
{
    head = NULL;
}
int QUEUEempty()
{
    return head == NULL;
}
QUEUEput(Item item)
{
    if (head == NULL)
        { head = (tail = NEW(item, head)); return; }
    tail->next = NEW(item, tail->next);
    tail = tail->next;
}
Item QUEUEget()
{
    Item item = head->item;
    link t = head->next;
    free(head); head = t;
    return item;
}
```

Program 4.11 FIFO queue array implementation

The contents of the queue are all the elements in the array between `head` and `tail`, taking into account the wraparound back to 0 when the end of the array is encountered. If `head` and `tail` are equal, then we consider the queue to be empty; but if `put` would make them equal, then we consider it to be full. As usual, we do not check such error conditions, but we make the size of the array 1 greater than the maximum number of elements that the client expects to see in the queue, so that we could augment this program to make such checks.

```
#include <stdlib.h>
#include "Item.h"
static Item *q;
static int N, head, tail;
void QUEUEinit(int maxN)
{ q = malloc((maxN+1)*sizeof(Item));
  N = maxN+1; head = N; tail = 0; }
int QUEUEempty()
{ return head % N == tail; }
void QUEUEput(Item item)
{ q[tail++] = item; tail = tail % N; }
Item QUEUEget()
{ head = head % N; return q[head++]; }
```

F	F
I	F I
R	F I R
S	F I R S
*	F
T	I R S T
*	I
I	R S T I
N	R S T I N
*	R
*	S
*	T
F	I N F
I	I N F I
*	I
R	N F I R
S	N F I R S
*	N
*	F
*	I
T	R S
*	R
O	T O
U	T O U
T	T O U T
*	S
*	T
*	O
*	U
T	T

representation uses space proportional to the number of elements in the data structure, at the cost of extra space for the links and extra time to allocate and deallocate memory for each operation.

Although we encounter stacks more often than we encounter FIFO queues, because of the fundamental relationship between stacks and recursive programs (see Chapter 5), we shall also encounter algorithms for which the queue is the natural underlying data structure. As we have already noted, one of the most frequent uses of queues and stacks in computational applications is to postpone computation. Although many applications that involve a queue of pending work operate correctly no matter what rule is used for *delete*, the overall running time or other resource usage may be dependent on the rule. When such applications involve a large number of *insert* and *delete* operations on data structures with a large number of items on them,

Figure 4.8
FIFO queue example, array implementation

This sequence shows the data manipulation underlying the abstract representation in Figure 4.6 when we implement the queue by storing the items in an array, keeping indices to the beginning and end of the queue, and wrapping the indices back to the beginning of the array when they reach the end of the array. In this example, the tail index wraps back to the beginning when the second T is inserted, and the head index wraps when the second S is removed.

performance differences are paramount. Accordingly, we devote a great deal of attention in this book to such ADTs. If we ignored performance, we could formulate a single ADT that encompassed *insert* and *delete*; since we do not ignore performance, each rule, in essence, constitutes a different ADT. To evaluate the effectiveness of a particular ADT, we need to consider two costs: the implementation cost, which depends on our choice of algorithm and data structure for the implementation; and the cost of the particular decision-making rule in terms of effect on the performance of the client. To conclude this section, we will describe a number of such ADTs, which we will be considering in detail throughout the book.

Specifically, pushdown stacks and FIFO queues are special instances of a more general ADT: the *generalized queue*. Instances of generalized queues differ in only the rule used when items are removed. For stacks, the rule is “remove the item that was most recently inserted”; for FIFO queues, the rule is “remove the item that was least recently inserted”; and there are many other possibilities, a few of which we now consider.

A simple but powerful alternative is the *random queue*, where the rule is to “remove a random item,” and the client can expect to get any of the items on the queue with equal probability. We can implement the operations of a random queue in constant time using an array representation (see Exercise 4.42). As do stacks and FIFO queues, the array representation requires that we reserve space ahead of time. The linked-list alternative is less attractive than it was for stacks and FIFO queues, however, because implementing both insertion and deletion efficiently is a challenging task (see Exercise 4.43). We can use random queues as the basis for randomized algorithms, to avoid, with high probability, worst-case performance scenarios (see Section 2.7).

We have described stacks and FIFO queues by identifying items according to the time that they were inserted into the queue. Alternatively, we can describe these abstract concepts in terms of a sequential listing of the items in order, and refer to the basic operations of inserting and deleting items from the beginning and the end of the list. If we insert at the end and delete at the end, we get a stack (precisely as in our array implementation); if we insert at the beginning and delete at the beginning, we also get a stack (precisely as in our linked-list implementation); if we insert at the end and delete at the beginning, we get a

FIFO queue (precisely as in our linked-list implementation); and if we insert at the beginning and delete at the end, we also get a FIFO queue (this option does not correspond to any of our implementations—we could switch our array implementation to implement it precisely, but the linked-list implementation is not suitable because of the need to back up the pointer to the end when we remove the item at the end of the list). Building on this point of view, we are led to the *deque* ADT, where we allow either insertion or deletion at either end. We leave the implementations for exercises (see Exercises 4.37 through 4.41), noting that the array-based implementation is a straightforward extension of Program 4.11, and that the linked-list implementation requires a doubly linked list, unless we restrict the deque to allow deletion at only one end.

In Chapter 9, we consider *priority queues*, where the items have keys and the rule for deletion is “remove the item with the smallest key.” The priority-queue ADT is useful in a variety of applications, and the problem of finding efficient implementations for this ADT has been a research goal in computer science for many years. Identifying and using the ADT in applications has been an important factor in this research: we can get an immediate indication whether or not a new algorithm is correct by substituting its implementation for an old implementation in a huge, complex application and checking that we get the same result. Moreover, we get an immediate indication whether a new algorithm is more efficient than an old one by noting the extent to which substituting the new implementation improves the overall running time. The data structures and algorithms that we consider in Chapter 9 for solving this problem are interesting, ingenious, and effective.

In Chapters 12 through 16, we consider *symbol tables*, which are generalized queues where the items have keys and the rule for deletion is “remove an item whose key is equal to a given key, if there is one.” This ADT is perhaps the most important one that we consider, and we shall examine dozens of implementations.

Each of these ADTs also give rise to a number of related, but different, ADTs that suggest themselves as an outgrowth of careful examination of client programs and the performance of implementations. In Sections 4.7 and 4.8, we consider numerous examples of

changes in the specification of generalized queues that lead to yet more different ADTs, which we shall consider later in this book.

Exercises

▷ 4.30 Give the contents of $q[0], \dots, q[4]$ after the execution of the operations illustrated in Figure 4.6, using Program 4.11. Assume that `maxN` is 10, as in Figure 4.8.

▷ 4.31 A letter means *put* and an asterisk means *get* in the sequence

E A S * Y * Q U E * * * S T * * * I O * N * * *.

Give the sequence of values returned by the *get* operations when this sequence of operations is performed on an initially empty FIFO queue.

4.32 Modify the array-based FIFO queue implementation in the text (Program 4.11) to call a function `QUEUEerror` if the client attempts to *get* when the queue is empty or to *put* when the queue is full.

4.33 Modify the linked-list-based FIFO queue implementation in the text (Program 4.10) to call a function `QUEUEerror` if the client attempts to *get* when the queue is empty or if there is no memory available from `malloc` for a *put*.

▷ 4.34 An uppercase letter means *put* at the beginning, a lowercase letter means *put* at the end, a plus sign means *get* from the beginning, and an asterisk means *get* from the end in the sequence

E A s + Y + Q U E * * + s t + * + I O * n + + *.

Give the sequence of values returned by the *get* operations when this sequence of operations is performed on an initially empty deque.

▷ 4.35 Using the conventions of Exercise 4.34, give a way to insert plus signs and asterisks in the sequence E a s Y so that the sequence of values returned by the *get* operations is (i) E a s Y ; (ii) Y a s E ; (iii) a Y s E ; (iv) a s Y E ; or, in each instance, prove that no such sequence exists.

• 4.36 Given two sequences, give an algorithm for determining whether or not it is possible to add plus signs and asterisks to make the first produce the second when interpreted as a sequence of deque operations in the sense of Exercise 4.35.

▷ 4.37 Write an interface for the deque ADT.

4.38 Provide an implementation for your deque interface (Exercise 4.37) that uses an array for the underlying data structure.

4.39 Provide an implementation for your deque interface (Exercise 4.37) that uses a doubly linked list for the underlying data structure.

4.40 Provide an implementation for the FIFO queue interface in the text (Program 4.9) that uses a circular list for the underlying data structure.

4.41 Write a client that tests your deque ADTs (Exercise 4.37) by reading, as the first argument on the command line, a string of commands like those given in Exercise 4.34 then performing the indicated operations. Add a function DQdump to the interface and implementations, and print out the contents of the deque after each operation, in the style of Figure 4.6.

○ **4.42** Build a random-queue ADT by writing an interface and an implementation that uses an array as the underlying data structure. Make sure that each operation takes constant time.

● **4.43** Build a random-queue ADT by writing an interface and an implementation that uses a linked list as the underlying data structure. Provide implementations for *insert* and *delete* that are as efficient as you can make them, and analyze their worst-case cost.

▷ **4.44** Write a client that picks numbers for a lottery by putting the numbers 1 through 99 on a random queue, then prints the result of removing five of them.

4.45 Write a client that takes an integer N from the first argument on the command line, then prints out N poker hands, by putting N items on a random queue (see Exercise 4.4), then printing out the result of picking five cards at a time from the queue.

● **4.46** Write a program that solves the connectivity problem by inserting all the pairs on a random queue and then taking them from the queue, using the quick-find-weighted algorithm (Program 1.3).

4.7 Duplicate and Index Items

For many applications, the abstract items that we process are *unique*, a quality that leads us to consider modifying our idea of how stacks, FIFO queues, and other generalized ADTs should operate. Specifically, in this section, we consider the effect of changing the specifications of stacks, FIFO queues, and generalized queues to disallow duplicate items in the data structure.

For example, a company that maintains a mailing list of customers might want to try to grow the list by performing *insert* operations from other lists gathered from many sources, but would not want the list to grow for an *insert* operation that refers to a customer already on the list. We shall see that the same principle applies in a variety of applications. For another example, consider the problem of routing a message through a complex communications network. We might try going through several paths simultaneously in the network,

L	L
A	LA
.	A L
S	LS
T	LST
I	LSTI
.	LST
N	LSTN
.	LST
F	LSTF
I	LSTFI
R	LSTFIR
.	LSTFI
S	LSTFI
T	LSTFI
.	LSTF
F	LST
.	L S
O	L SO
U	L SOU
.	L SO
T	L SOT
.	L SO
O	L S
.	L

Figure 4.9
Pushdown stack with no duplicates

This sequence shows the result of the same operations as those in Figure 4.1, but for a stack with no duplicate objects allowed. The gray squares mark situations where the stack is left unchanged because the item to be pushed is already on the stack. The number of items on the stack is limited by the number of possible distinct items.

but there is only one message, so any particular node in the network would want to have only one copy in its internal data structures.

One approach to handling this situation is to leave up to the clients the task of ensuring that duplicate items are not presented to the ADT, a task that clients presumably might carry out using some different ADT. But since the purpose of an ADT is to provide clients with clean solutions to applications problems, we might decide that detecting and resolving duplicates is a part of the problem that the ADT should help to solve.

The policy of disallowing duplicate items is a change in the *abstraction*: the interface, names of the operations, and so forth for such an ADT are the same as those for the corresponding ADT without the policy, but the behavior of the implementation changes in a fundamental way. In general, whenever we modify the specification of an ADT, we get a completely new ADT—one that has completely different properties. This situation also demonstrates the precarious nature of ADT specification: Being sure that clients and implementations adhere to the specifications in an interface is difficult enough, but enforcing a high-level policy such as this one is another matter entirely. Still, we are interested in algorithms that do so because clients can exploit such properties to solve problems in new ways, and implementations can take advantage of such restrictions to provide more efficient solutions.

Figure 4.9 shows how a modified no-duplicates stack ADT would operate for the example corresponding to Figure 4.1; Figure 4.10 shows the effect of the change for FIFO queues.

In general, we have a policy decision to make when a client makes an *insert* request for an item that is already in the data structure. Should we proceed as though the request never happened, or should we proceed as though the client had performed a *delete* followed by an *insert*? This decision affects the order in which items are ultimately processed for ADTs such as stacks and FIFO queues (see Figure 4.11), and the distinction is significant for client programs. For example, the company using such an ADT for a mailing list might prefer to use the new item (perhaps assuming that it has more up-to-date information about the customer), and the switching mechanism using such an ADT might prefer to ignore the new item (perhaps it has already taken steps to send along the message). Furthermore, this policy choice affects the implementations: the forget-the-old-item policy is generally more

F	F
I	FI
R	FIR
S	FIRS
*	FIRS
T	IRST
*	IRST
I	RSTI
N	RSTIN
*	RSTIN
S	STIN
*	STIN
T	IN
F	INF
*	INF
I	NF
R	NFR
S	NFRS
*	NFRS
F	RS
*	S
T	ST
*	ST
O	TO
U	TOU
T	TOU
*	TOU
O	U
*	U

Figure 4.10
FIFO queue with no duplicates, ignore-the-new-item policy

This sequence shows the result of the same operations as those in Figure 4.6, but for a queue with no duplicate objects allowed. The gray squares mark situations where the queue is left unchanged because the item to be put onto the queue is already there.

difficult to implement than the ignore-the-new-item policy, because it requires that we modify the data structure.

To implement generalized queues with no duplicate items, we assume that we have an abstract operation for testing item equality, as discussed in Section 4.1. Given such an operation, we still need to be able to determine whether a new item to be inserted is already in the data structure. This general case amounts to implementing the symbol-table ADT, so we shall consider it in the context of the implementations given in Chapters 12 through 15.

There is an important special case for which we have a straightforward solution, which is illustrated for the pushdown stack ADT in Program 4.12. This implementation assumes that the items are integers in the range 0 to $M - 1$. Then, it uses a second array, indexed by the item itself, to determine whether that item is in the stack. When we insert item i , we set the i th entry in the second array to 1; when we delete item i , we set the i th entry in the array to 0. Otherwise, we use the same code as before to insert and delete items, with one additional test: Before inserting an item, we can test to see whether it is already in the stack. If it is, we ignore the *push*. This solution does not depend on whether we use an array or linked-list (or some other) representation for the stack. Implementing an ignore-the-old-item policy involves more work (see Exercise 4.51).

In summary, one way to implement a stack with no duplicates using an ignore-the-new-item policy is to maintain *two* data structures: the first contains the items in the stack, as before, to keep track of the order in which the items in the stack were inserted; the second is an array that allows us to keep track of which items are in the stack, by using the item as an index. Using an array in this way is a special case of a symbol-table implementation, which is discussed in Section 12.2. We can apply the same technique to any generalized queue ADT, when we know the items to be integers in the range 0 to $M - 1$.

This special case arises frequently. The most important example is when the items in the data structure are themselves array indices, so we refer to such items as *index items*. Typically, we have a set of M objects, kept in yet another array, that we need to pass through a generalized queue structure as a part of a more complex algorithm. Objects are put on the queue by index and processed when they are

F	F
I	F I
R	F I R
S	F I R S
.	F I R S
T	I R S T
.	I R S T
I	R S T I
N	R S T I N
.	R S T I N
S	T I N
.	T I N
F	I N F
I	N F I
.	N F I
R	F I R
S	F I R S
.	F I R S
.	I R S
R	S
.	S
T	S T
.	S T
O	T O
U	T O U
T	O U T
.	O U T
U	T
.	T

Figure 4.11
FIFO queue with no duplicates, forget-the-old-item policy

This sequence shows the result of the same operations as in Figure 4.10, but using the (more difficult to implement) policy by which we always add a new item at the end of the queue. If there is a duplicate, we remove it.

Program 4.12 Stack with index items and no duplicates

This pushdown-stack implementation assumes that all items are integers between 0 and `maxN-1`, so that it can maintain an array `t` that has a nonzero value corresponding to each item in the stack. The array enables `STACKpush` to test quickly whether its argument is already on the stack, and to take no action if the test succeeds. We use only one bit per entry in `t`, so we could save space by using characters or bits instead of integers, if desired (see Exercise 12.12).

```
#include <stdlib.h>
static int *s, *t;
static int N;
void STACKinit(int maxN)
{
    int i;
    s = malloc(maxN*sizeof(int));
    t = malloc(maxN*sizeof(int));
    for (i = 0; i < maxN; i++) t[i] = 0;
    N = 0;
}
int STACKempty()
{
    return !N;
}
void STACKpush(int item)
{
    if (t[item] == 1) return;
    s[N++] = item; t[item] = 1;
}
int STACKpop()
{
    N--; t[s[N]] = 0; return s[N];
}
```

removed, and each object is to be processed precisely once. Using array indices in a queue with no duplicates accomplishes this goal directly.

Each of these choices (disallow duplicates, or do not; and use the new item, or do not) leads to a new ADT. The differences may seem minor, but they obviously affect the dynamic behavior of the ADT as seen by client programs, and affect our choice of algorithm and data structure to implement the various operations, so we have no alternative but to treat all the ADTs as different. Furthermore, we have other options to consider: For example, we might wish to modify

the interface to inform the client program when it attempts to insert a duplicate item, or to give the client the option whether to ignore the new item or to forget the old one.

When we informally use a term such as *pushdown stack*, *FIFO queue*, *deque*, *priority queue*, or *symbol table*, we are potentially referring to a family of ADTs, each with different sets of defined operations and different sets of conventions about the meanings of the operations, each requiring different and, in some cases, more sophisticated implementations to be able to support those operations efficiently.

Exercises

- ▷ 4.47 Draw a figure corresponding to Figure 4.9 for the stack ADT that disallows duplicates using a forget-the-old-item policy.
- 4.48 Modify the standard array-based stack implementation in Section 4.4 (Program 4.4) to disallow duplicates with an ignore-the-new-item policy. Use a brute-force approach that involves scanning through the whole stack.
- 4.49 Modify the standard array-based stack implementation in Section 4.4 (Program 4.4) to disallow duplicates with a forget-the-old-item policy. Use a brute-force approach that involves scanning through, and possibly rearranging, the whole stack.
- 4.50 Do Exercises 4.48 and 4.49 for the linked-list-based stack implementation in Section 4.4 (Program 4.5).
- 4.51 Develop a pushdown-stack implementation that disallows duplicates, using a forget-the-old-item policy for integer items between 0 and $M - 1$, and that uses constant time for both *push* and *pop*. Hint: Use a doubly linked list representation for the stack and keep pointers to nodes, rather than 0-1 values, in an item-indexed array.
- 4.52 Do Exercises 4.48 and 4.49 for the FIFO queue ADT.
- 4.53 Do Exercise 4.50 for the FIFO queue ADT.
- 4.54 Do Exercise 4.51 for the FIFO queue ADT.
- 4.55 Do Exercises 4.48 and 4.49 for the randomized-queue ADT.
- 4.56 Write a client program for your ADT from Exercise 4.55, which exercises a randomized queue with no duplicates.

4.8 First-Class ADTs

Our interfaces and implementations of stack and FIFO queue ADTs in Sections 4.2 through 4.7 provide clients with the capability to use

a *single* instance of a particular generalized stack or queue, and to achieve the important objective of hiding from the client the particular data structure used in the implementation. Such ADTs are widely useful, and will serve as the basis for many of the implementations that we consider in this book.

These objects are disarmingly simple when considered as ADTs themselves, however, because there is only one object in a given program. The situation is analogous to having a program, for example, that manipulates only one integer. We could perhaps increment, decrement, and test the value of the integer, but could not declare variables or use it as an argument or return value in a function, or even multiply it by another integer. In this section, we consider how to construct ADTs that we can manipulate in the same way that we manipulate built-in types in client programs, while still achieving the objective of hiding the implementation from the client.

Definition 4.4 A first-class data type is one for which we can have potentially many different instances, and which we can assign to variables which we can declare to hold the instances.

For example, we could use first-class data types as arguments and return values to functions.

The method that we will use to implement first-class data types applies to any data type: in particular, it applies to generalized queues, so it provides us with the capability to write programs that manipulate stacks and FIFO queues in much the same way that we manipulate other types of data in C. This capability is important in the study of algorithms because it provides us with a natural way to express high-level operations involving such ADTs. For example, we can speak of operations to *join* two queues—to combine them into one. We shall consider algorithms that implement such operations for the priority queue ADT (Chapter 9) and for the symbol table ADT (Chapter 12).

Some modern languages provide specific mechanisms for building first-class ADTs, but the idea transcends specific mechanisms. Being able to manipulate instances of ADTs in much the same way that we manipulate built-in data types such as `int` or `float` is an important goal in the design of many high-level programming languages, because it allows any applications program to be written such that the program manipulates the objects of central concern to the application; it allows

many programmers to work simultaneously on large systems, all using a precisely defined set of abstract operations; and it provides for those abstract operations to be implemented in many different ways without any changes to the applications code—for example for new machines and programming environments. Some languages even allow *operator overloading*, which allows us to use basic symbols such as + or * to define operators. C does not provide specific support for building first-class data types, but it does provide primitive operations that we can use to achieve that goal. There are a number of ways to proceed in C. To keep our focus on algorithms and data structures, as opposed to programming-language design issues, we do not consider all the alternatives; rather, we describe and adopt just one convention that we can use throughout the book.

To illustrate the basic approach, we begin by considering, as an example, a first-class data type and then a first-class ADT for the *complex-number* abstraction. Our goal is to be able to write programs like Program 4.13, which performs algebraic operations on complex numbers using operations defined in the ADT. We implement the *add* and *multiply* operations as standard C functions, since C does not support operator overloading.

Program 4.13 uses few properties of complex numbers; we now digress to consider these properties briefly. In one sense, we are not digressing at all, because it is interesting to contemplate the relationship between complex numbers themselves as a mathematical abstraction and this abstract representation of them in a computer program.

The number $i = \sqrt{-1}$ is an *imaginary* number. Although $\sqrt{-1}$ is meaningless as a real number, we name it i , and perform algebraic manipulations with i , replacing i^2 with -1 whenever it appears. A *complex number* consists of two parts, real and imaginary—complex numbers can be written in the form $a + bi$, where a and b are reals. To multiply complex numbers, we apply the usual algebraic rules, replacing i^2 with -1 whenever it appears. For example,

$$(a + bi)(c + di) = ac + bci + adi + bdi^2 = (ac - bd) + (ad + bc)i.$$

The real or imaginary parts might cancel out (have the value 0) when we perform a complex multiplication. For example,

$$(1 - i)(1 - i) = 1 - i - i + i^2 = -2i,$$

$$(1 + i)^4 = 4i^2 = -4,$$

Program 4.13 Complex numbers driver (roots of unity)

This client program performs a computation on complex numbers using an ADT that allows it to compute directly with the abstraction of interest by declaring variables of type `Complex` and using them as arguments and return values of functions. This program checks the ADT implementation by computing the powers of the roots of unity. It prints the table shown in Figure 4.12.

```
#include <stdio.h>
#include <math.h>
#include "COMPLEX.h"
#define PI 3.141592625
main(int argc, char *argv[])
{
    int i, j, N = atoi(argv[1]);
    Complex t, x;
    printf("%dth complex roots of unity\n", N);
    for (i = 0; i < N; i++)
    {
        float r = 2.0*PI*i/N;
        t = COMPLEXinit(cos(r), sin(r));
        printf("%2d %6.3f %6.3f ", i, Re(t), Im(t));
        for (x = t, j = 0; j < N-1; j++)
            x = COMPLEXmult(t, x);
        printf("%6.3f %6.3f\n", Re(x), Im(x));
    }
}
```

$$(1+i)^8 = 16.$$

Scaling the preceding equation by dividing through by $16 = (\sqrt{2})^8$, we find that

$$\left(\frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}}\right)^8 = 1.$$

In general, there are many complex numbers that evaluate to 1 when raised to a power. These are the *complex roots of unity*. Indeed, for each N , there are exactly N complex numbers z with $z^N = 1$. The numbers

$$\cos\left(\frac{2\pi k}{N}\right) + i \sin\left(\frac{2\pi k}{N}\right),$$

Program 4.14 First-class data type for complex numbers

This interface for complex numbers includes a `typedef` that allows implementations to declare variables of type `Complex` and to use these variables as function arguments and return values. However, the data type is not abstract, because this representation is not hidden from clients.

```
typedef struct { float Re; float Im; } Complex;
Complex COMPLEXinit(float, float);
float Re(Complex);
float Im(Complex);
Complex COMPLEXmult(Complex, Complex);
```

for $k = 0, 1, \dots, N - 1$ are easily shown to have this property (see Exercise 4.63). For example, taking $k = 1$ and $N = 8$ in this formula gives the particular eighth root of unity that we just discovered.

Program 4.13 is an example of a client program for the complex-numbers ADT that raises each of the N th roots of unity to the N th power, using the multiplication operation defined in the ADT. The output that it produces is shown in Figure 4.12: We expect that each number raised to the N th power gives the same result: 1, or $1 + 0i$.

This client program differs from the client programs that we have considered to this point in one major respect: it declares variables of type `Complex` and assigns values to such variables, including using them as arguments and return values in functions. Accordingly, we need to define the type `Complex` in the interface.

Program 4.14 is an interface for complex numbers that we might consider using. It defines the type `Complex` as a `struct` comprising two floats (for the real and imaginary part of the complex number), and declares four functions for processing complex numbers: initialize, extract real and imaginary parts, and multiply. Program 4.15 gives implementations of these functions, which are straightforward. Together, these two functions provide an effective implementation of a complex-number ADT that we can use successfully in client programs such as Program 4.13.

The interface in Program 4.14 specifies one particular representation for complex numbers—a structure containing two integers (the real and imaginary parts). By including this representation within the

0	1.000	0.000	1.000	0.000
1	0.707	0.707	1.000	0.000
2	-0.000	1.000	1.000	0.000
3	-0.707	0.707	1.000	0.000
4	-1.000	-0.000	1.000	0.000
5	-0.707	-0.707	1.000	-0.000
6	0.000	-1.000	1.000	0.000
7	0.707	-0.707	1.000	-0.000

Figure 4.12
Complex roots of unity

This table gives the output that is produced by Program 4.13 when invoked with a.out 8. The eight complex roots of unity are $\pm 1, \pm i$, and

$$\pm \frac{\sqrt{2}}{2} \pm \frac{\sqrt{2}}{2}i$$

(left two columns). Each of these eight numbers gives the result $1 + 0i$ when raised to the eighth power (right two columns).

Program 4.15 Complex-numbers data-type implementation

These function implementations for the complex numbers data type are straightforward. However, we would prefer not to separate them from the definition of the `Complex` type, which is defined in the interface for the convenience of the client.

```
#include "COMPLEX.h"
Complex COMPLEXinit(float Re, float Im)
    { Complex t; t.Re = Re; t.Im = Im; return t; }
float Re(Complex z)
    { return z.Re; }
float Im(Complex z)
    { return z.Im; }
Complex COMPLEXmult(Complex a, Complex b)
    { Complex t;
        t.Re = a.Re*b.Re - a.Im*b.Im;
        t.Im = a.Re*b.Im + a.Im*b.Re;
        return t;
    }
```

interface, however, we are making it available for use by client programs. Programmers often organize interfaces in this way. Essentially, doing so amounts to publishing a standard representation for a new data type that might be used by many client programs. In this example, client programs could refer directly to `t.Re` and `t.Im` for any variable `t` of type `Complex`. The advantage of allowing such access is that we thus ensure that clients that need to directly implement their own manipulations that may not be present in the type's suite of operations at least agree on the standard representation. The disadvantage of allowing clients direct access to the data is that we cannot change the representation without changing all the clients. In short, Program 4.14 is not an *abstract* data type, because the representation is not hidden by the interface.

Even for this simple example, the difficulty of changing representations is significant because there is another standard representation that we might wish to consider using: polar coordinates (see Exercise 4.62). For an application with more complicated data structures, the ability to change representations is a requirement. For example,

Program 4.16 First-class ADT for complex numbers

This interface provides clients with handles to complex number objects, but does not give any information about the representation—it is a struct that is not specified, except for its tag name.

```
typedef struct complex *Complex;
Complex COMPLEXinit(float, float);
float Re(Complex);
float Im(Complex);
Complex COMPLEXmult(Complex, Complex);
```

our company that needs to process mailing lists needs to use the same client program to process mailing lists in different formats. With a first-class ADT, the client programs can manipulate the data without direct access, but rather with indirect access, through operations defined in the ADT. An operation such as *extract customer name* then can have different implementations for different list formats. The most important implication of this arrangement is that we can change the data representation without having to change the client programs.

We use the term *handle* to describe a reference to an abstract object. Our goal is to give client programs handles to abstract objects that can be used in assignment statements and as arguments and return values of functions in the same way as built-in data types, while hiding the representation of the objects from the client program.

Program 4.16 is an example of such an interface for complex numbers that achieves this goal, and exemplifies the conventions that we shall use throughout this book. The handle is defined as a pointer to a structure that has a name tag, but is otherwise *not specified*. The client can use this handle as intended, but there can be no code in the client program that uses the handle in any other way: It cannot access a field in a structure by dereferencing the pointer because it does not have the names of any of the fields. In the interface, we define functions which accept handles as arguments and also return handles as values; and client programs can use those functions, all without knowing anything about the data structure that will be used to implement the interface.

Program 4.17 is an implementation of the interface of Program 4.16. It defines the specific data structure that will be used to implement handles and the data type itself; a function that allocates the memory for a new object and initializes its fields; functions that provide access to the fields (which we implement by dereferencing the handle pointer to access the specific fields in the argument objects); and functions that implement the ADT operations. All information specific to the data structure being used is guaranteed to be encapsulated in the implementation, because the client has no way to refer to it.

The distinction between the data type for complex numbers in the code in Programs 4.14 and 4.15 and the ADT for complex numbers in the code in Programs 4.16 and 4.17 is essential and is thus well worth careful study. It is a mechanism that we can use to develop and compare efficient algorithms for fundamental problems throughout this book. We shall not treat all the implications of using such a mechanism for software engineering in further detail, but it is a powerful and general mechanism that will serve us well in the study of algorithms and data structures and their application.

In particular, the issue of storage management is critical in the use of ADTs in software engineering. When we say $x = t$ in Program 4.13, where the variables are both of type `Complex`, we simply are assigning a pointer. The alternative would be to allocate memory for a new object and define an explicit *copy* function to copy the values in the object associated with t to the new object. This issue of *copy semantics* is an important one to address in any ADT design. We normally use pointer assignment (and therefore do not consider *copy* implementations for our ADTs) because of our focus on efficiency—this choice makes us less susceptible to excessive hidden costs when performing operations on huge data structures. The design of the C string data type is based on similar considerations.

The implementation of `COMPLEXmult` in Program 4.15 creates a new object for the result. Alternatively, more in the spirit of reserving explicit object-creation operations for the client, we could return the value in one of the arguments. As it stands, `COMPLEXmult` has a defect called a *memory leak*, that makes the program unusable for a huge number of multiplications. The problem is that each multiplication allocates memory for a new object, but we never execute any calls to `free`. For this reason, ADTs often contain explicit *destroy* operations

Program 4.17 Complex-numbers ADT implementation

By contrast with Program 4.15, this implementation of the complex-numbers ADT includes the structure definition (which is hidden from the client), as well as the function implementations. Objects are pointers to structures, so we dereference the pointer to refer to the fields.

```
#include <stdlib.h>
#include "COMPLEX.h"
struct complex { float Re; float Im; };
Complex COMPLEXinit(float Re, float Im)
{
    Complex t = malloc(sizeof *t);
    t->Re = Re; t->Im = Im;
    return t;
}
float Re(Complex z)
{
    return z->Re;
}
float Im(Complex z)
{
    return z->Im;
}
Complex COMPLEXmult(Complex a, Complex b)
{
    return COMPLEXinit(Re(a)*Re(b) - Im(a)*Im(b),
                       Re(a)*Im(b) + Im(a)*Re(b));
}
```

for use by clients. However, having the capability for *destroy* is no guarantee that clients will use it for each and every object created, and memory leaks are subtle defects that plague many large systems. For this reason, some programming environments have automatic mechanisms for the system to invoke *destroy*; other systems have *automatic memory allocation*, where the system takes responsibility to figure out which memory is no longer being used by programs, and to reclaim it. None of these solutions is entirely satisfactory. We rarely include *destroy* implementations in our ADTs, since these considerations are somewhat removed from the essential characteristics of our algorithms.

First-class ADTs play a central role in many of our implementations because they provide the necessary support for the abstract mechanisms for generic objects and collections of objects that we dis-

Program 4.18 First-class ADT interface for queues

We provide handles for queues in precisely the same manner as we did for complex numbers in Program 4.16: A handle is a pointer to a structure that is unspecified except for the tag name.

```
typedef struct queue *Q;
void QUEUEEdump(Q);
Q QUEUEinit(int maxN);
int QUEUEEmpty(Q);
void QUEUEput(Q, Item);
Item QUEUEget(Q);
```

```
6 13 51 64 71 84 90
4 23 26 34 38 62 78
8 28 33 48 54 56 75 81
2 15 17 37 43 47 50 53 61 80 82
12 25 30 32 36 49 52 63 74 79
3 14 22 27 31 42 46 59 77
9 19 20 29 39 45 69 70 73 76 83
5 11 18 24 35 44 57 58 67
0 1 21 40 41 55 66 72
7 10 16 60 65 68
```

Figure 4.13
Random-queue simulation

This table gives the output that is produced when Program 4.19 is invoked with 84 as the command-line argument. The 10 queues have an average of 8.4 items each, ranging from a low of six to a high of 11.

cussed in Section 4.1. Accordingly, we use `Item` for the type of the items that we manipulate in the generalized queue ADTs in this book (and include an `Item.h` interface file), secure in the knowledge that an appropriate implementation will make our code useful for whatever data type a client program might need.

To illustrate further the general nature of the basic mechanism, we consider next a first-class ADT for FIFO queues using the same basic scheme that we just used for complex numbers. Program 4.18 is the interface for this ADT. It differs from Program 4.9 in that it defines a queue handle (to be a pointer to an unspecified structure, in the standard manner) and each function takes a queue handle as an argument. With handles, client programs can manipulate multiple queues.

Program 4.19 is a driver program that exemplifies such a client. It randomly assigns N items to one of M FIFO queues, then prints out the contents of the queues, by removing the items one by one. Figure 4.13 is an example of the output produced by this program. Our interest in this program is to illustrate how the first-class data-type mechanism allows it to work with the queue ADT itself as a high-level object—it could easily be extended to test various methods of organizing queues to serve customers, and so forth.

Program 4.20 is an implementation of the FIFO queue ADT defined in Program 4.18, using linked lists for the underlying data structure. The primary difference between these implementations and those in Program 4.10 has to do with the variables `head` and `tail`.

Program 4.19 Queue client program (queue simulation)

The availability of object handles makes it possible to build compound data structures with ADT objects, such as the array of queues in this sample client program, which simulates a situation where customers waiting for service are assigned at random to one of M service queues.

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#include "QUEUE.h"
#define M 10
main(int argc, char *argv[])
{ int i, j, N = atoi(argv[1]);
  Q queues[M];
  for (i = 0; i < M; i++)
    queues[i] = QUEUEinit(N);
  for (i = 0; i < N; i++)
    QUEUEput(queues[rand() % M], j);
  for (i = 0; i < M; i++, printf("\n"))
    for (j = 0; !QUEUEempty(queues[i]); j++)
      printf("%3d ", QUEUEget(queues[i]));
}
```

In Program 4.10, we had only one queue, so we simply declared and used these variables in the implementation. In Program 4.20, each queue q has its own pointers `head` and `tail`, which we reference with the code $q->head$ and $q->tail$. The definition of `struct queue` in an implementation answers the question “what is a queue?” for that implementation: In this case, the answer is that a queue is pointer to a structure consisting of the links to the head and tail of the queue. In an array implementation, a queue is a pointer to a struct consisting of a pointer to an array and two integers: the size of the array and the number of elements currently on the queue (see Exercise 4.65). In general, the members of the structure are exactly the global or static variables from the one-object implementation.

With a carefully designed ADT, we can make use of the separation between client and implementations in many interesting ways. For example, we commonly use driver programs when developing or

Program 4.20 Linked-list implementation of first-class queue

The code for implementations that provide object handles is typically more cumbersome than the corresponding code for single objects (see Program 4.10). This code does not check for errors such as a client attempt to *get* from an empty queue or an unsuccessful *malloc* (see Exercise 4.33).

```
#include <stdlib.h>
#include "Item.h"
#include "QUEUE.h"
typedef struct QUEUEnode* link;
struct QUEUEnode { Item item; link next; };
struct queue { link head; link tail; };
link NEW(Item item, link next)
{ link x = malloc(sizeof *x);
  x->item = item; x->next = next;
  return x;
}
Q QUEUEinit(int maxN)
{ Q q = malloc(sizeof *q);
  q->head = NULL;
  return q;
}
int QUEUEempty(Q q)
{ return q->head == NULL; }
void QUEUEput(Q q, Item item)
{
  if (q->head == NULL)
    { q->tail = NEW(item, q->head);
      q->head = q->tail; return; }
  q->tail->next = NEW(item, q->tail->next);
  q->tail = q->tail->next;
}
Item QUEUEget(Q q)
{ Item item = q->head->item;
  link t = q->head->next;
  free(q->head); q->head = t;
  return item;
}
```

debugging ADT implementations. Similarly, we often use incomplete implementations of ADTs, called *stubs*, as placeholders while building systems to learn properties of clients, although this exercise can be tricky for clients that depend on the ADT implementation semantics.

As we saw in Section 4.3, the ability to have multiple instances of a given ADT in a single program can lead us to complicated situations. Do we want to be able to have stacks or queues with different types of objects on them? How about different types of objects on the same queue? Do we want to use different implementations for queues of the same type in a single client because we know of performance differences? Should information about the efficiency of implementations be included in the interface? What form should that information take? Such questions underscore the importance of understanding the basic characteristics of our algorithms and data structures and how client programs may use them effectively, which is, in a sense, the topic of this book. Full implementations, however, are exercises in software engineering, rather than in algorithms design, so we stop short of developing ADTs of such generality in this book (*see reference section*).

Despite its virtues, our mechanism for providing first-class ADTs comes at the (slight) cost of extra pointer dereferences and slightly more complicated implementation code, so we shall use the full mechanism for only those ADTs that require the use of handles as arguments or return values in interfaces. On the one hand, the use of first-class types might encompass the majority of the code in a small number of huge applications systems; on the other hand, an only-one-object arrangement—such as the stacks, FIFO queues, and generalized queues of Sections 4.2 through 4.7—and the use of `typedef` to specify the types of objects as described in Section 4.1 are quite serviceable techniques for many of the programs that we write. In this book, we introduce most of the algorithms and data structures that we consider in the latter context, then extend these implementations into first-class ADTs when warranted.

Exercises

- ▷ 4.57 Add a function `COMPLEXadd` to the ADT for complex numbers in the text (Programs 4.16 and 4.17).
- 4.58 Convert the equivalence-relations ADT in Section 4.5 to a first-class type.
- 4.59 Create a first-class ADT for use in programs that process playing cards.

•• 4.60 Write a program to determine empirically the probability that various poker hands are dealt, using your ADT from Exercise 4.59.

4.61 Create an ADT for points in the plane, and change the closest-point program in Chapter 3 Program 3.16 to a client program that uses your ADT.

○ 4.62 Develop an implementation for the complex-number ADT that is based on representing complex numbers in polar coordinates (that is, in the form $r e^{i\theta}$).

● 4.63 Use the identity $e^{i\theta} = \cos \theta + i \sin \theta$ to prove that $e^{2\pi i} = 1$ and that the N complex N th roots of unity are

$$\cos\left(\frac{2\pi k}{N}\right) + i \sin\left(\frac{2\pi k}{N}\right),$$

for $k = 0, 1, \dots, N - 1$.

4.64 List the N th roots of unity for N from 2 through 8.

4.65 Develop an implementation of the FIFO queue first-class ADT given in the text (Program 4.18) that uses an array as the underlying data structure.

▷ 4.66 Write an interface for a first-class pushdown-stack ADT.

4.67 Develop an implementation of your first-class pushdown-stack ADT from Exercise 4.66 that uses an array as the underlying data structure.

4.68 Develop an implementation of your first-class pushdown-stack ADT from Exercise 4.66 that uses a linked list as the underlying data structure.

○ 4.69 Modify the postfix-evaluation program in Section 4.3 to evaluate postfix expressions consisting of complex numbers with integer coefficients, using the first-class complex numbers ADT in the text (Programs 4.16 and 4.17). For simplicity, assume that the complex numbers all have nonnull integer coefficients for both real and imaginary parts and are written with no spaces. For example, your program should print the output $8+4i$ when given the input

$1+1i\ 0+1i\ +\ 1-2i\ *\ 3+4i\ +\ .$

4.9 Application-Based ADT Example

As a final example, we consider in this section an application-specific ADT that is representative of the relationship between applications domains and the algorithms and data structures of the type that we consider in this book. The example that we shall consider is the *polynomial* ADT. It is drawn from *symbolic mathematics*, where we use the computer to help us manipulate abstract mathematical objects.

Our goal is to be able to perform computations such as

$$\left(1 - x + \frac{x^2}{2} - \frac{x^3}{6}\right)\left(1 + x + x^2 + x^3\right) = 1 + \frac{x^2}{2} + \frac{x^3}{3} - \frac{2x^4}{3} + \frac{x^5}{3} - \frac{x^6}{6}.$$

Program 4.21 Polynomial client (binomial coefficients)

This client program uses the polynomial ADT that is defined in the interface Program 4.22 to perform algebraic manipulations with polynomials. It takes an integer N and a floating-point number p from the command line, computes $(x + 1)^N$, and checks the result by evaluating the resulting polynomial at $x = p$.

```
#include <stdio.h>
#include <stdlib.h>
#include "POLY.h"
main(int argc, char *argv[])
{
    int N = atoi(argv[1]); float p = atof(argv[2]);
    Poly t, x; int i, j;
    printf("Binomial coefficients\n");
    t = POLYadd(POLYterm(1, 1), POLYterm(1, 0));
    for (i = 0, x = t; i < N; i++)
        { x = POLYmult(t, x); showPOLY(x); }
    printf("%f\n", POLYeval(x, p));
}
```

We also want to be able to evaluate the polynomial for a given value of x . For $x = 0.5$, both sides of this equation have the value 1.1328125. The operations of multiplying, adding, and evaluating polynomials are at the heart of a great many mathematical calculations. Program 4.21 is a simple example that performs the symbolic operations corresponding to the polynomial equations

$$\begin{aligned}(x+1)^2 &= x^2 + 2x + 1, \\ (x+1)^3 &= x^3 + 3x^2 + 3x + 1, \\ (x+1)^4 &= x^4 + 4x^3 + 6x^2 + 4x + 1, \\ (x+1)^5 &= x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1, \\ \dots\end{aligned}$$

The same basic ideas extend to include operations such as composition, integration, differentiation, knowledge of special functions, and so forth.

The first step is to define the *polynomial* ADT, as illustrated in the interface Program 4.22. For a well-understood mathematical

abstraction such as a polynomial, the specification is so clear as to be unspoken (in the same way as for the ADT for complex numbers that we discussed in Section 4.8): We want instances of the ADT to behave precisely in the same manner as the well-understood mathematical abstraction.

To implement the functions defined in the interface, we need to choose a particular data structure to represent polynomials and then to implement algorithms that manipulate the data structure to produce the behavior that client programs expect from the ADT. As usual, the choice of data structure affects the potential efficiency of the algorithms, and we are free to consider several. Also as usual, we have the choice of using a linked representation or an array representation. Program 4.23 is an implementation using an array representation; the linked-list representation is left as an exercise (see Exercise 4.70).

To *add* two polynomials, we add their coefficients. If the polynomials are represented as arrays, the *add* function amounts to a single loop through the arrays, as shown in Program 4.23. To *multiply* two polynomials, we use the elementary algorithm based on the distributive law. We multiply one polynomial by each term in the other, line up the results so that powers of x match, then add the terms to get the final result. The following table summarizes the computation for $(1 - x + x^2/2 - x^3/6)(1 + x + x^2 + x^3)$:

$$\begin{array}{r}
 1 - x + \frac{x^2}{2} - \frac{x^3}{6} \\
 \times -x^2 + \frac{x^3}{2} - \frac{x^4}{6} \\
 \hline
 + x^2 - x^3 + \frac{x^4}{2} - \frac{x^5}{6} \\
 + x^3 - x^4 + \frac{x^5}{2} - \frac{x^6}{6} \\
 \hline
 1 + \frac{x^2}{2} + \frac{x^3}{3} - \frac{2x^4}{3} + \frac{x^5}{3} - \frac{x^6}{6}
 \end{array}$$

The computation seems to require time proportional to N^2 to multiply two polynomials. Finding a faster algorithm for this task is a significant challenge. We shall consider this topic in detail in Part 8, where we shall see that it is possible to accomplish the task in time proportional to

Program 4.22 First-class ADT interface for polynomials

As usual, a handle to a polynomial is a pointer to a structure that is unspecified except for the tag name.

```
typedef struct poly *Poly;
void showPOLY(Poly);
Poly POLYterm(int, int);
Poly POLYadd(Poly, Poly);
Poly POLYmult(Poly, Poly);
float POLYeval(Poly, float);
```

$N^{3/2}$ using a divide-and-conquer algorithm, and in time proportional to $N \lg N$ using the fast Fourier transform.

The implementation of the *evaluate* function in Program 4.23 uses a classic efficient algorithm known as *Horner's algorithm*. A naive implementation of the function involves a direct computation using a function that computes x^N . This approach takes quadratic time. A less naive implementation involves saving the values of x^i in a table, then using them in a direct computation. This approach takes linear extra space. Horner's algorithm is a direct optimal linear algorithm based on parenthesizations such as

$$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = (((a_4x + a_3)x + a_2)x + a_1)x + a_0.$$

Horner's method is often presented as a time-saving trick, but it is actually an early and outstanding example of an elegant and efficient algorithm, which reduces the time required for this essential computational task from quadratic to linear. The calculation that we performed in Program 4.2 for converting ASCII strings to integers is a version of Horner's algorithm. We shall encounter Horner's algorithm again, in Chapter 14 and Part 5, as the basis for an important computation related to certain symbol-table and string-search implementations.

For simplicity and efficiency, POLYadd modifies one of its arguments; if we choose to use this implementation in an application, we should note that fact in the specification (see Exercise 4.71). Moreover, we have memory leaks, particularly in POLYmult, which creates a new polynomial to hold the result (see Exercise 4.72).

As usual, the array representation for implementing the polynomial ADT is but one possibility. If exponents are huge and there are

Program 4.23 Array implementation of polynomial ADT

In this implementation of a first-class ADT for polynomials, a polynomial is a structure containing the degree and a pointer to an array of coefficients. For simplicity in this code, each addition operation modifies one of its arguments and each multiplication operation creates a new object. Another ADT operation to destroy objects (and to free the associated memory) might be needed for some applications.

```
#include <stdlib.h>
#include "POLY.h"
struct poly { int N; int *a; };
Poly POLYterm(int coeff, int exp)
{ int i; Poly t = malloc(sizeof(*t));
  t->a = malloc((exp+1)*sizeof(int));
  t->N = exp+1; t->a[exp] = coeff;
  for (i = 0; i < exp; i++) t->a[i] = 0;
  return t;
}
Poly POLYadd(Poly p, Poly q)
{ int i; Poly t;
  if (p->N < q->N) { t = p; p = q; q = t; }
  for (i = 0; i < q->N; i++) p->a[i] += q->a[i];
  return p;
}
Poly POLYmult(Poly p, Poly q)
{ int i, j;
  Poly t = POLYterm(0, (p->N-1)+(q->N-1));
  for (i = 0; i < p->N; i++)
    for (j = 0; j < q->N; j++)
      t->a[i+j] += p->a[i]*q->a[j];
  return t;
}
float POLYeval(Poly p, float x)
{ int i; double t = 0.0;
  for (i = p->N-1; i >= 0; i--)
    t = t*x + p->a[i];
  return t;
}
```

not many terms, a linked-list representation might be more appropriate. For example, we would not want to use Program 4.23 to perform a multiplication such as

$$(1 + x^{1000000})(1 + x^{2000000}) = 1 + x^{1000000} + x^{2000000} + x^{3000000},$$

because it would use an array with space for hundreds of thousands of unused coefficients. Exercise 4.70 explores the linked list option in more detail.

Exercises

4.70 Provide an implementation for the polynomial ADT given in the text (Program 4.22) that uses linked lists as the underlying data structure. Your lists should not contain any nodes corresponding to terms with coefficient value 0.

▷ 4.71 Modify the implementation of `POLYadd` in Program 4.23 such that it operates in a manner similar to `POLYmult` (and does not modify either of its arguments).

○ 4.72 Modify the polynomial ADT interface, implementation, and client in the text (Programs 4.21 through 4.23) such that there are no memory leaks. To do so, define new operations `POLYdestroy` and `POLYcopy`, which should free the memory for an object and copy one object's values to another, respectively; and modify `POLYadd` and `POLYmult` to destroy their arguments and return a newly created object, by convention.

○ 4.73 Extend the polynomial ADT given in the text to include integration and differentiation of polynomials.

○ 4.74 Modify your polynomial ADT from Exercise 4.73 to ignore all terms with exponents greater than or equal to an integer M , which is provided by the client at initialization.

●● 4.75 Extend your polynomial ADT from Exercise 4.73 to include polynomial division and composition.

● 4.76 Develop an ADT that allows clients to perform addition and multiplication of arbitrarily long integers.

● 4.77 Modify the postfix-evaluation program in Section 4.3 to evaluate postfix expressions consisting of arbitrarily long integers, using the ADT that you developed for Exercise 4.76.

●● 4.78 Write a client program that uses your polynomial ADT from Exercise 4.75 to evaluate integrals by using Taylor series approximations of functions, manipulating them symbolically.

4.79 Develop an ADT that provides clients with the ability to perform algebraic operations on vectors of floating-point numbers.

4.80 Develop an ADT that provides clients with the ability to perform algebraic operations on matrices of abstract objects for which addition, subtraction, multiplication, and division are defined.

4.81 Write an interface for a character-string ADT, which includes operations for creating a string, comparing two strings, concatenating two strings, copying one string to another, and returning the string length.

4.82 Provide an implementation for your string ADT interface from Exercise 4.81, using the C string library where appropriate.

4.83 Provide an implementation for your string ADT interface from Exercise 4.81, using a linked list for the underlying representation. Analyze the worst-case running time of each operation.

4.84 Write an interface and an implementation for an index set ADT, which processes sets of integers in the range 0 to $M - 1$ (where M is a defined constant) and includes operations for creating a set, computing the union of two sets, computing the intersection of two sets, computing the complement of a set, computing the difference of two sets, and printing out the contents of a set. In your implementation, use an array of $M - 1$ 0-1 values to represent each set.

4.85 Write a client program that tests your ADT from Exercise 4.84.

4.10 Perspective

There are three primary reasons for us to be aware of the fundamental concepts underlying ADTs as we embark on the study of algorithms and data structures:

- ADTs are an important software-engineering tool in widespread use, and many of the algorithms that we study serve as implementations for fundamental ADTs that are widely applicable.
- ADTs help us to encapsulate the algorithms that we develop, so that we can use the same code for many different purposes.
- ADTs provide a convenient mechanism for our use in the process of developing and comparing the performance of algorithms.

Ideally, ADTs embody the common-sense principle that we are obligated to describe precisely the ways in which we manipulate our data. The client-interface-implementation mechanism that we have considered in detail in this chapter is convenient for this task in C, and provides us with C code that has a number of desirable properties. Many modern languages have specific support that allows the development of programs with similar properties, but the general approach

transcends particular languages—when we do not have specific language support, we adopt programming conventions to maintain the separation that we would like to have among clients, interfaces, and implementations.

As we consider an ever-expanding set of choices in specifying the behavior of our ADTs, we are faced with an ever-expanding set of challenges in providing efficient implementations. The numerous examples that we have considered illustrate ways of meeting such challenges. We continually strive to achieve the goal of implementing all the operations efficiently, but we are unlikely to have a general-purpose implementation that can do so for all sets of operations. This situation works against the principles that lead us to ADTs in the first place, because in many cases implementors of ADTs need to know properties of client programs to know which implementations of associated ADTs will perform most efficiently, and implementors of client programs need to know performance properties of various implementations to know which to choose for a particular application. As ever, we must strike a balance. In this book, we consider numerous approaches to implementations for variants of fundamental ADTs, all of which have important applications.

We can use one ADT to build another. We have used the pointer and structure abstractions provided by C to build linked lists, then we have used linked lists or the array abstraction provided by C to build pushdown stacks, then we use pushdown stacks to get the capability to evaluate arithmetic expressions. The ADT concept allows us to construct large systems on different layers of abstraction, from the machine-language instructions provided by the computer, to the various capabilities provided by the programming language, to sorting, searching and other higher-level capabilities provided by algorithms as discussed in Parts 3 and 4 of this book, to the even higher levels of abstraction that the various applications require, as discussed in Parts 5 through 8. ADTs are one point on the continuum of developing ever more powerful abstract mechanisms that is the essence of using computers effectively in problem solving.



CHAPTER FIVE

Recursion and Trees

THE CONCEPT OF recursion is fundamental in mathematics and computer science. The simple definition is that a recursive program in a programming language is one that calls itself (just as a recursive function in mathematics is one that is defined in terms of itself). A recursive program cannot call itself always, or it would never stop (just as a recursive function cannot be defined in terms of itself always, or the definition would be circular); so a second essential ingredient is that there must be a *termination condition* when the program can cease to call itself (and when the mathematical function is not defined in terms of itself). All practical computations can be couched in a recursive framework.

The study of recursion is intertwined with the study of recursively defined structures known as *trees*. We use trees both to help us understand and analyze recursive programs and as explicit data structures. We have already encountered an application of trees (although not a recursive one), in Chapter 1. The connection between recursive programs and trees underlies a great deal of the material in this book. We use trees to understand recursive programs; we use recursive programs to build trees; and we draw on the fundamental relationship between both (and recurrence relations) to analyze algorithms. Recursion helps us to develop elegant and efficient data structures and algorithms for all manner of applications.

Our primary purpose in this chapter is to examine recursive programs and data structures as practical tools. First, we discuss the relationship between mathematical recurrences and simple recursive

programs, and we consider a number of examples of practical recursive programs. Next, we examine the fundamental recursive scheme known as *divide and conquer*, which we use to solve fundamental problems in several later sections of this book. Then, we consider a general approach to implementing recursive programs known as *dynamic programming*, which provides effective and elegant solutions to a wide class of problems. Next, we consider trees, their mathematical properties, and associated algorithms in detail, including basic methods for *tree traversal* that underlie recursive tree-processing programs. Finally, we consider closely related algorithms for processing graphs—we look specifically at a fundamental recursive program, *depth-first search*, that serves as the basis for many graph-processing algorithms.

As we shall see, many interesting algorithms are simply expressed with recursive programs, and many algorithm designers prefer to express methods recursively. We also investigate nonrecursive alternatives in detail. Not only can we often devise simple stack-based algorithms that are essentially equivalent to recursive algorithms, but also we can often find nonrecursive alternatives that achieve the same final result through a different sequence of computations. The recursive formulation provides a structure within which we can seek more efficient alternatives.

A full discussion of recursion and trees could fill an entire book, for they arise in many applications throughout computer science, and are pervasive outside of computer science as well. Indeed, it might be said that *this* book is filled with a discussion of recursion and trees, for they are present, in a fundamental way, in every one of the book's chapters.

5.1 Recursive Algorithms

A *recursive algorithm* is one that solves a problem by solving one or more smaller instances of the same problem. To implement recursive algorithms in C, we use *recursive functions*—a recursive function is one that calls itself. Recursive functions in C correspond to recursive definitions of mathematical functions. We begin our study of recursion by examining programs that directly evaluate mathematical functions. The basic mechanisms extend to provide a general-purpose programming paradigm, as we shall see.

Program 5.1 Factorial function (recursive implementation)

This recursive function computes the function $N!$, using the standard recursive definition. It returns the correct value when called with N nonnegative and sufficiently small that $N!$ can be represented as an `int`.

```
int factorial(int N)
{
    if (N == 0) return 1;
    return N*factorial(N-1);
}
```

Recurrence relations (see Section 2.5) are recursively defined functions. A recurrence relation defines a function whose domain is the nonnegative integers either by some initial values or (recursively) in terms of its own values on smaller integers. Perhaps the most familiar such function is the *factorial* function, which is defined by the recurrence relation

$$N! = N \cdot (N - 1)! , \quad \text{for } N \geq 1 \text{ with } 0! = 1.$$

This definition corresponds directly to the recursive C function in Program 5.1.

Program 5.1 is equivalent to a simple loop. For example, the following `for` loop performs the same computation:

```
for (t = 1, i = 1; i <= N; i++) t *= i;
```

As we shall see, it is always possible to transform a recursive program into a nonrecursive one that performs the same computation. Conversely, we can express without loops any computation that involves loops, using recursion, as well.

We use recursion because it often allows us to express complex algorithms in a compact form, without sacrificing efficiency. For example, the recursive implementation of the factorial function obviates the need for local variables. The cost of the recursive implementation is borne by the mechanisms in the programming systems that support function calls, which use the equivalent of a built-in pushdown stack. Most modern programming systems have carefully engineered mechanisms for this task. Despite this advantage, as we shall see, it is all too easy to write a simple recursive function that is extremely inefficient,

Program 5.2 A questionable recursive program

If the argument N is odd, this function calls itself with $3N + 1$ as an argument; if N is even, it calls itself with $N/2$ as an argument. We cannot use induction to prove that this program terminates, because not every recursive call uses an argument smaller than the one given.

```
int puzzle(int N)
{
    if (N == 1) return 1;
    if (N % 2 == 0)
        return puzzle(N/2);
    else return puzzle(3*N+1);
}
```

and we need to exercise care to avoid being burdened with intractable implementations.

Program 5.1 illustrates the basic features of a recursive program: it calls itself (with a smaller value of its argument), and it has a termination condition in which it directly computes its result. We can use mathematical induction to convince ourselves that the program works as intended:

- It computes $0!$ (basis).
- Under the assumption that it computes $k!$ for $k < N$ (inductive hypothesis), it computes $N!$.

Reasoning like this can provide us with a quick path to developing algorithms that solve complex problems, as we shall see.

In a programming language such as C, there are few restrictions on the kinds of programs that we write, but we strive to limit ourselves in our use of recursive functions to those that embody inductive proofs of correctness like the one outlined in the previous paragraph. Although we do not consider formal correctness proofs in this book, we are interested in putting together complicated programs for difficult tasks, and we need to have some assurance that the tasks will be solved properly. Mechanisms such as recursive functions can provide such assurances while giving us compact implementations. Practically speaking, the connection to mathematical induction tells us that we should ensure that our recursive functions satisfy two basic properties:

- They must explicitly solve a basis case.

```
puzzle(3)
  puzzle(10)
    puzzle(5)
      puzzle(16)
        puzzle(8)
          puzzle(4)
            puzzle(2)
              puzzle(1)
```

Figure 5.1
Example of a recursive call chain

This nested sequence of function calls eventually terminates, but we cannot prove that the recursive function in Program 5.2 does not have arbitrarily deep nesting for some argument. We prefer recursive programs that always invoke themselves with smaller arguments.

Program 5.3 Euclid's algorithm

One of the oldest-known algorithms, dating back over 2000 years, is this recursive method for finding the greatest common divisors of two integers.

```
int gcd(int m, int n)
{
    if (n == 0) return m;
    return gcd(n, m % n);
}
```

- Each recursive call must involve smaller values of the arguments.
- These points are vague—they amount to saying that we should have a valid inductive proof for each recursive function that we write. Still, they provide useful guidance as we develop implementations.

Program 5.2 is an amusing example that illustrates the need for an inductive argument. It is a recursive function that violates the rule that each recursive call must involve smaller values of the arguments, so we cannot use mathematical induction to understand it. Indeed, it is not known whether or not this computation terminates for every N , if there are no bounds on the size of N . For small integers that can be represented as ints, we can check that the program terminates (see Figure 5.1 and Exercise 5.4), but for large integers (64-bit words, say), we do not know whether or not this program goes into an infinite loop.

Program 5.3 is a compact implementation of *Euclid's algorithm* for finding the greatest common divisor of two integers. It is based on the observation that the greatest common divisor of two integers x and y with $x > y$ is the same as the greatest common divisor of y and $x \bmod y$ (the remainder when x is divided by y). A number t divides both x and y if and only if t divides both y and $x \bmod y$, because x is equal to $x \bmod y$ plus a multiple of y . The recursive calls made for an example invocation of this program are shown in Figure 5.2. For Euclid's algorithm, the depth of the recursion depends on arithmetic properties of the arguments (it is known to be logarithmic).

Program 5.4 is an example with multiple recursive calls. It is another expression evaluator, performing essentially the same computations as Program 4.2, but on prefix (rather than postfix) expressions,

```
gcd(314159, 271828)
gcd(271828, 42331)
gcd(42331, 17842)
gcd(17842, 6647)
gcd(6647, 4458)
gcd(4458, 2099)
gcd(2099, 350)
gcd(350, 349)
gcd(349, 1)
gcd(1, 0)
```

Figure 5.2
Example of Euclid's algorithm

This nested sequence of function calls illustrates the operation of Euclid's algorithm in discovering that 314159 and 271828 are relatively prime.

```

eval() * + 7 * * 4 6 + 8 9 5
eval() + 7 * * 4 6 + 8 9
  eval() 7
  eval() * * 4 6 + 8 9
    eval() * 4 6
      eval() 4
      eval() 6
    return 24 = 4*6
  eval() + 8 9
    eval() 8
    eval() 9
    return 17 = 8 + 9
  return 408 = 24*17
return 415 = 7+408
eval() 5
return 2075 = 415*5

```

Figure 5.3
Prefix expression evaluation example

This nested sequence of function calls illustrates the operation of the recursive prefix-expression-evaluation algorithm on a sample expression. For simplicity, the expression arguments are shown here. The algorithm itself never explicitly decides the extent of its argument string; rather, it takes what it needs from the front of the string.

Program 5.4 Recursive program to evaluate prefix expressions

To evaluate a prefix expression, we either convert a number from ASCII to binary (in the `while` loop at the end), or perform the operation indicated by the first character in the expression on the two operands, evaluated recursively. This function is recursive, but it uses a global array containing the expression and an index to the current character in the expression. The pointer is advanced past each subexpression evaluated.

```

char *a; int i;
int eval()
{
  int x = 0;
  while (a[i] == ' ') i++;
  if (a[i] == '+')
    { i++; return eval() + eval(); }
  if (a[i] == '*')
    { i++; return eval() * eval(); }
  while ((a[i] >= '0') && (a[i] <= '9'))
    x = 10*x + (a[i++]-'0');
  return x;
}

```

and letting recursion take the place of the explicit pushdown stack. In this chapter, we shall see many other examples of recursive programs and equivalent programs that use pushdown stacks. We shall examine the specific relationship between several pairs of such programs in detail.

Figure 5.3 shows the operation of Program 5.4 on a sample prefix expression. The multiple recursive calls mask a complex series of computations. Like most recursive programs, this program is best understood inductively: Assuming that it works properly for simple expressions, we can convince ourselves that it works properly for complex ones. This program is a simple example of a *recursive descent parser*—we can use the same process to convert C programs into machine code.

A precise inductive proof that Program 5.4 evaluates the expression properly is certainly much more challenging to write than are the proofs for functions with integer arguments that we have been

discussing, and we shall encounter recursive programs and data structures that are even more complicated than this one throughout this book. Accordingly, we do not pursue the idealistic goal of providing complete inductive proofs of correctness for every recursive program that we write. In this case, the ability of the program to “know” how to separate the operands corresponding to a given operator seems mysterious at first (perhaps because we cannot immediately see how to do this separation at the top level), but is actually a straightforward calculation (because the path to pursue at each function call is unambiguously determined by the first character in the expression).

In principle, we can replace any for loop by an equivalent recursive program. Often, the recursive program is a more natural way to express the computation than the for loop, so we may as well take advantage of the mechanism provided by the programming system that supports recursion. There is one hidden cost, however, that we need to bear in mind. As is plain from the examples that we examined in Figures 5.1 through 5.3, when we execute a recursive program, we are nesting function calls, until we reach a point where we do not do a recursive call, and we return instead. In most programming environments, such nested function calls are implemented using the equivalent of built-in pushdown stacks. We shall examine the nature of such implementations throughout this chapter. The *depth of the recursion* is the maximum degree of nesting of the function calls over the course of the computation. Generally, the depth will depend on the input. For example, the depths of the recursions for the examples depicted in Figures 5.2 and 5.3 are 9 and 4, respectively. When using a recursive program, we need to take into account that the programming environment has to maintain a pushdown stack of size proportional to the depth of the recursion. For huge problems, the space needed for this stack might prevent us from using a recursive solution.

Data structures built from nodes with pointers are inherently recursive. For example, our definition of linked lists in Chapter 3 (Definition 3.3) is recursive. Therefore, recursive programs provide natural implementations of many commonly used functions for manipulating such data structures. Program 5.5 comprises four examples. We use such implementations frequently throughout the book, primarily because they are so much easier to understand than are their nonrecursive counterparts. However, we must exercise caution in using programs

such as those in Program 5.5 when processing huge lists, because the depth of the recursion for those functions can be proportional to the length of the lists, so the space required for the recursive stack might become prohibitive.

Some programming environments automatically detect and eliminate *tail recursion*, when the last action of a function is a recursive call, because it is not strictly necessary to add to the depth of the recursion in such a case. This improvement would effectively transform the count, traversal, and deletion functions in Program 5.5 into loops, but it does not apply to the reverse-order traversal function.

In Sections 5.2 and 5.3, we consider two families of recursive algorithms that represent essential computational paradigms. Then, in Sections 5.4 through 5.7, we consider recursive data structures that serve as the basis for a very large fraction of the algorithms that we consider.

Exercises

- ▷ 5.1 Write a recursive program to compute $\lg(N!)$.
- 5.2 Modify Program 5.1 to compute $N! \bmod M$, such that overflow is no longer an issue. Try running your program for $M = 997$ and $N = 10^3, 10^4, 10^5$, and 10^6 , to get an indication of how your programming system handles deeply nested recursive calls.
- ▷ 5.3 Give the sequences of argument values that result when Program 5.2 is invoked for each of the integers 1 through 9.
- 5.4 Find the value of $N < 10^6$ for which Program 5.2 makes the maximum number of recursive calls.
- ▷ 5.5 Provide a nonrecursive implementation of Euclid's algorithm.
- ▷ 5.6 Give the figure corresponding to Figure 5.2 for the result of running Euclid's algorithm for the inputs 89 and 55.
- 5.7 Give the recursive depth of Euclid's algorithm when the input values are two consecutive Fibonacci numbers (F_N and F_{N+1}).
- ▷ 5.8 Give the figure corresponding to Figure 5.3 for the result of recursive prefix-expression evaluation for the input $+ * * 12 12 12 144$.
- 5.9 Write a recursive program to evaluate postfix expressions.
- 5.10 Write a recursive program to evaluate infix expressions. You may assume that operands are always enclosed in parentheses.
- 5.11 Write a recursive program that converts infix expressions to postfix.
- 5.12 Write a recursive program that converts postfix expressions to infix.

Program 5.5 Examples of recursive functions for linked lists

These recursive functions for simple list-processing tasks are easy to express, but may not be useful for huge lists because the depth of the recursion may be proportional to the length of the list.

The first function, `count`, counts the number of nodes on the list. The second, `traverse`, calls the function `visit` for each node on the list, from beginning to end. These two functions are both also easy to implement with a `for` or `while` loop. The third function, `traverseR`, does not have a simple iterative counterpart. It calls the function `visit` for every node on the list, but in reverse order.

The fourth function, `delete`, makes the structural changes needed for a given item to be deleted from a list. It returns a link to the (possibly altered) remainder of the list—the link returned is `x`, except when `x->item` is `v`, when the link returned is `x->next` (and the recursion stops).

```
int count(link x)
{
    if (x == NULL) return 0;
    return 1 + count(x->next);
}

void traverse(link h, void (*visit)(link))
{
    if (h == NULL) return;
    (*visit)(h);
    traverse(h->next, visit);
}

void traverseR(link h, void (*visit)(link))
{
    if (h == NULL) return;
    traverseR(h->next, visit);
    (*visit)(h);
}

link delete(link x, Item v)
{
    if (x == NULL) return NULL;
    if (eq(x->item, v))
        { link t = x->next; free(x); return t; }
    x->next = delete(x->next, v);
    return x;
}
```

5.13 Write a recursive program for solving the Josephus problem (see Section 3.3).

5.14 Write a recursive program that deletes the final element of a linked list.

5.15 Write a recursive program for reversing the order of the nodes in a linked list (see Program 3.7). *Hint:* Use a global variable.

5.2 Divide and Conquer

Many of the recursive programs that we consider in this book use two recursive calls, each operating on about one-half of the input. This recursive scheme is perhaps the most important instance of the well-known *divide-and-conquer* paradigm for algorithm design, which serves as the basis for many of our most important algorithms.

As an example, let us consider the task of finding the maximum among N items stored in an array $a[0], \dots, a[N-1]$. We can easily accomplish this task with a single pass through the array, as follows:

```

Y max(0, 10)
Y max(0, 5)
T max(0, 2)
    T max(0, 1)
        T max(0, 0)
            I max(1, 1)
N max(2, 2)
Y max(3, 5)
Y max(3, 4)
    Y max(3, 3)
        E max(4, 4)
X max(5, 5)
P max(6, 10)
P max(6, 8)
M max(6, 7)
    A max(6, 6)
    M max(7, 7)
    P max(8, 8)
L max(9, 10)
L max(9, 9)
E max(10, 10)

```

Figure 5.4
A recursive approach to finding the maximum

This sequence of function calls illustrates the dynamics of finding the maximum with a recursive algorithm.

The recursive divide-and-conquer solution given in Program 5.6 is also a simple (entirely different) algorithm for the same problem; we use it to illustrate the divide-and-conquer concept.

Most often, we use the divide-and-conquer approach because it provides solutions faster than those available with simple iterative algorithms (we shall discuss several examples at the end of this section), but it also is worthy of close examination as a way of understanding the nature of certain fundamental computations.

Figure 5.4 shows the recursive calls that are made when Program 5.6 is invoked for a sample array. The underlying structure seems complicated, but we normally do not need to worry about it—we depend on a proof by induction that the program works, and we use a recurrence relation to analyze the program's performance.

As usual, the code itself suggests the proof by induction that it performs the desired computation:

- It finds the maximum for arrays of size 1 explicitly and immediately.
- For $N > 1$, it partitions the array into two arrays of size less than N , finds the maximum of the two parts by the inductive

Program 5.6 Divide-and-conquer to find the maximum

This function divides a file $a[1], \dots, a[r]$ into $a[1], \dots, a[m]$ and $a[m+1], \dots, a[r]$, finds the maximum elements in the two parts (recursively), and returns the larger of the two as the maximum element in the whole file. It assumes that `Item` is a first-class type for which `>` is defined. If the file size is even, the two parts are equal in size; if the file size is odd, the size of the first part is 1 greater than the size of the second part.

```
Item max(Item a[], int l, int r)
{ Item u, v; int m = (l+r)/2;
  if (l == r) return a[l];
  u = max(a, l, m);
  v = max(a, m+1, r);
  if (u > v) return u; else return v;
}
```

hypothesis, and returns the larger of these two values, which must be the maximum value in the whole array.

Moreover, we can use the recursive structure of the program to understand its performance characteristics.

Property 5.1 *A recursive function that divides a problem of size N into two independent (nonempty) parts that it solves recursively calls itself less than N times.*

If the parts are one of size k and one of size $N - k$, then the total number of recursive function calls that we use is

$$T_N = T_k + T_{N-k} + 1, \quad \text{for } N \geq 1 \text{ with } T_1 = 0.$$

The solution $T_N = N - 1$ is immediate by induction. If the sizes sum to a value less than N , the proof that the number of calls is less than $N - 1$ follows the same inductive argument. We can prove analogous results under general conditions (see Exercise 5.20). ■

Program 5.6 is representative of many divide-and-conquer algorithms with precisely the same recursive structure, but other examples may differ in two primary respects. First, Program 5.6 does a constant amount of work on each function call, so its total running time is linear. Other divide-and-conquer algorithms may perform more work

0	10								
0	5	6	10						
0	2	3	5	6	10				
0	1	2	2	3	5	6	10		
0	0	1	1	2	2	3	5	6	10
1	1	2	2	3	5	6	10		
2	2	3	5	6	10				
3	5	6	10						
3	4	5	5	6	10				
3	3	4	4	5	5	6	10		
4	4	5	5	6	10				
5	5	6	10						
6	10								
6	8	9	10						
6	7	8	8	9	10				
6	6	7	7	8	8	9	10		
7	7	8	8	9	10				
8	8	9	10						
9	10								
9	9	10	10						
10	10								

Figure 5.5
Example of internal stack dynamics

This sequence is an idealistic representation of the contents of the internal stack during the sample computation of Figure 5.4. We start with the left and right indices of the whole subarray on the stack. Each line depicts the result of popping two indices and, if they are not equal, pushing four indices, which delimit the left subarray and the right subarray after the popped subarray is divided into two parts. In practice, the system keeps return addresses and local variables on the stack, instead of this specific representation of the work to be done, but this model suffices to describe the computation.

on each function call, as we shall see, so determining the total running time requires more intricate analysis. The running time of such algorithms depends on the precise manner of division into parts. Second, Program 5.6 is representative of divide-and-conquer algorithms for which the parts sum to make the whole. Other divide-and-conquer algorithms may divide into smaller parts that constitute less than the whole problem, or overlapping parts that total up to more than the whole problem. These algorithms are still proper recursive algorithms because *each* part is smaller than the whole, but analyzing them is more difficult than analyzing Program 5.6. We shall consider the analysis of these different types of algorithms in detail as we encounter them.

For example, the binary-search algorithm that we studied in Section 2.6 is a divide-and-conquer algorithm that divides a problem in half, then works on just one of the halves. We examine a recursive implementation of binary search in Chapter 12.

Figure 5.5 indicates the contents of the internal stack maintained by the programming environment to support the computation in Figure 5.4. The model depicted in the figure is idealistic, but it gives useful insights into the structure of the divide-and-conquer computation. If a program has two recursive calls, the actual internal stack contains one entry corresponding to the first function call while that function is being executed (which contains values of arguments, local variables, and a return address), then a similar entry corresponding to the second function call while that function is being executed. The alternative that is depicted in Figure 5.5 is to put the two entries on the stack at once, keeping all the subtasks remaining to be done explicitly on the stack. This arrangement plainly delineates the computation, and sets the stage for more general computational schemes, such as those that we examine in Sections 5.6 and 5.8.

Figure 5.6 depicts the structure of the divide-and-conquer find-the-maximum computation. It is a recursive structure: the node at the top contains the size of the input array, the structure for the left subarray is drawn at the left and the structure for the right subarray is drawn at the right. We will formally define and discuss tree structures of this type in Sections 5.4 and 5.5. They are useful for understanding the structure of any program involving nested function calls—recursive programs in particular. Also shown in Figure 5.6 is the same tree, but with each node labeled with the return value for the corresponding

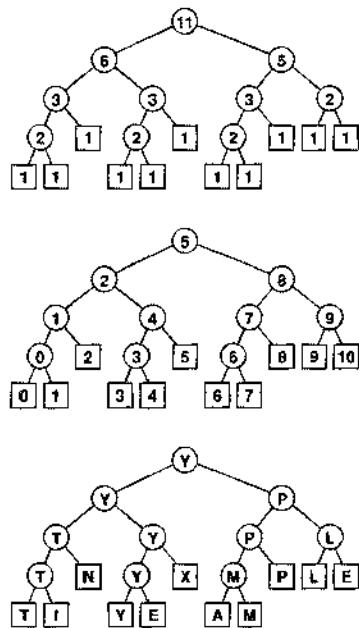


Figure 5.6
Recursive structure of find-the-maximum algorithm.

The divide-and-conquer algorithm splits a problem of size 11 into one of size 6 and one of size 5, a problem of size 6 into two problems of size 3, and so forth, until reaching problems of size 1 (top). Each circle in these diagrams represents a call on the recursive function, to the nodes just below connected to it by lines (squares are those calls for which the recursion terminates). The diagram in the middle shows the value of the index into the middle of the file that we use to effect the split; the diagram at the bottom shows the return value.

Program 5.7 Solution to the towers of Hanoi

We shift the tower of disks to the right by (recursively) shifting all but the bottom disk to the left, then shifting the bottom disk to the right, then (recursively) shifting the tower back onto the bottom disk.

```
void hanoi(int N, int d)
{
    if (N == 0) return;
    hanoi(N-1, -d);
    shift(N, d);
    hanoi(N-1, -d);
}
```

function call. In Section 5.7, we shall consider the process of building explicit linked structures that represent trees like this one.

No discussion of recursion would be complete without the ancient *towers of Hanoi* problem. We have three pegs and N disks that fit onto the pegs. The disks differ in size, and are initially arranged on one of the pegs, in order from largest (disk N) at the bottom to smallest (disk 1) at the top. The task is to move the stack of disks to the right one position (peg), while obeying the following rules: (i) only one disk may be shifted at a time; and (ii) no disk may be placed on top of a smaller one. One legend says that the world will end when a certain group of monks accomplishes this task in a temple with 40 golden disks on three diamond pegs.

Program 5.7 gives a recursive solution to the problem. It specifies which disk should be shifted at each step, and in which direction (+ means move one peg to the right, cycling to the leftmost peg when on the rightmost peg; and - means move one peg to the left, cycling to the rightmost peg when on the leftmost peg). The recursion is based on the following idea: To move N disks one peg to the right, we first move the top $N - 1$ disks one peg to the left, then shift disk N one peg to the right, then move the $N - 1$ disks one more peg to the left (onto disk N). We can verify that this solution works by induction. Figure 5.7 shows the moves for $N = 5$ and the recursive calls for $N = 3$. An underlying pattern is evident, which we now consider in detail.

First, the recursive structure of this solution immediately tells us the number of moves that the solution requires.

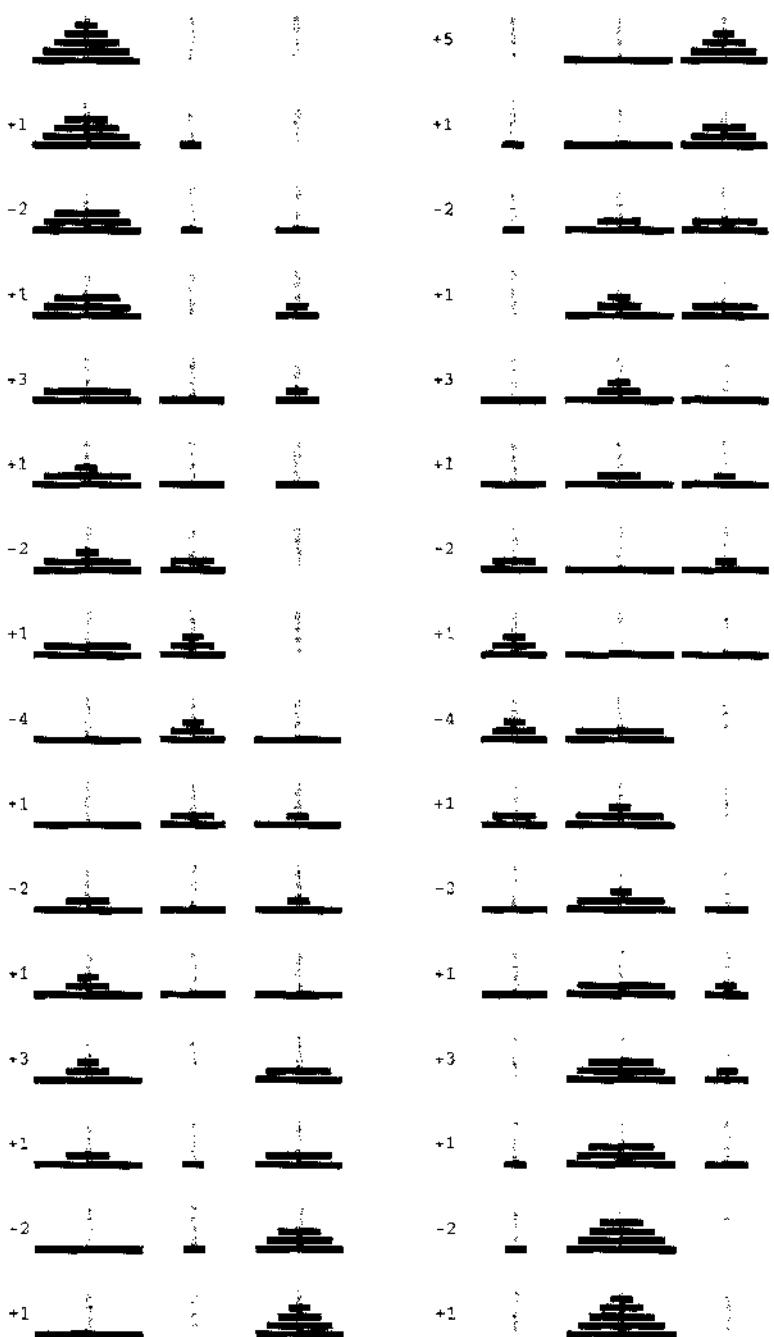
Figure 5.7
Towers of Hanoi

This diagram depicts the solution to the towers of Hanoi problem for five disks. We shift the top four disks left one position (left column), then move disk 5 to the right, then shift the top four disks left one position (right column). The sequence of function calls that follows constitutes the computation for three disks. The computed sequence of moves is +1 -2 +1 +3 +1 -2 +1, which appears four times in the solution (for example, the first seven moves).

```

hanoi(3, +1)
  hanoi(2, -1)
    hanoi(1, +1)
      hanoi(0, -1)
        shift(1, +1)
        hanoi(0, -1)
      shift(2, -1)
      hanoi(1, +1)
        hanoi(0, -1)
        shift(1, +1)
        hanoi(0, -1)
      shift(3, +1)
      hanoi(2, -1)
        hanoi(1, +1)
          hanoi(0, -1)
          shift(1, +1)
          hanoi(0, -1)
        shift(2, -1)
        hanoi(1, +1)
          hanoi(0, -1)
          shift(1, +1)
          hanoi(0, -1)

```



Property 5.2 *The recursive divide-and-conquer algorithm for the towers of Hanoi problem produces a solution that has $2^N - 1$ moves.*

As usual, it is immediate from the code that the number of moves satisfies a recurrence. In this case, the recurrence satisfied by the number of disk moves is similar to Formula 2.5:

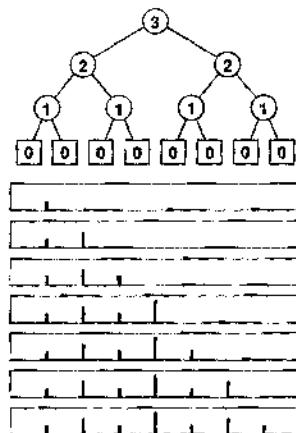
$$T_N = 2T_{N-1} + 1, \quad \text{for } N \geq 2 \text{ with } T_1 = 1.$$

We can verify the stated result directly by induction: we have $T(1) = 2^1 - 1 = 1$; and, if $T(k) = 2^k - 1$ for $k < N$, then $T(N) = 2(2^{N-1} - 1) + 1 = 2^N - 1$. ■

If the monks are moving disks at the rate of one per second, it will take at least 348 centuries for them to finish (see Figure 2.1), assuming that they do not make a mistake. The end of the world is likely be even further off than that because those monks presumably never have had the benefit of being able to use Program 5.7, and might not be able to figure out so quickly which disk to move next. We now consider an analysis of the method that leads to a simple (nonrecursive) method that makes the decision easy. While we may not wish to let the monks in on the secret, it is relevant to numerous important practical algorithms.

To understand the towers of Hanoi solution, let us consider the simple task of drawing the markings on a ruler. Each inch on the ruler has a mark at the $1/2$ inch point, slightly shorter marks at $1/4$ inch intervals, still shorter marks at $1/8$ inch intervals, and so forth. Our task is to write a program to draw these marks at any given resolution, assuming that we have at our disposal a procedure `mark(x, h)` to make a mark h units high at position x .

If the desired resolution is $1/2^n$ inches, we rescale so that our task is to put a mark at every point between 0 and 2^n , endpoints not included. Thus, the middle mark should be n units high, the marks in the middle of the left and right halves should be $n-1$ units high, and so forth. Program 5.8 is a straightforward divide-and-conquer algorithm to accomplish this objective; Figure 5.8 illustrates it in operation on a small example. Recursively speaking, the idea behind the method is the following. To make the marks in an interval, we first divide the interval into two equal halves. Then, we make the (shorter) marks in the left half (recursively), the long mark in the middle, and the (shorter) marks in the right half (recursively). Iteratively speaking, Figure 5.8



```

rule(0, 8, 3)
  rule(0, 4, 2)
    rule(0, 2, 1)
      rule(0, 1, 0)
        mark(1, 1)
        rule(1, 2, 0)
      mark(2, 2)
    rule(2, 4, 1)
      rule(2, 3, 0)
      mark(3, 1)
      rule(3, 4, 0)
    mark(4, 3)
  rule(4, 8, 2)
    rule(4, 6, 1)
      rule(4, 5, 0)
      mark(5, 1)
      rule(5, 6, 0)
    mark(6, 2)
    rule(6, 8, 1)
      rule(6, 7, 0)
      mark(7, 1)
      rule(7, 8, 0)

```

Figure 5.8
Ruler-drawing function calls

This sequence of function calls constitutes the computation for drawing a ruler of length 8, resulting in marks of lengths 1, 2, 1, 3, 1, 2, and 1.

Program 5.8 Divide and conquer to draw a ruler

To draw the marks on a ruler, we draw the marks on the left half, then draw the longest mark in the middle, then draw the marks on the right half. This program is intended to be used with $r - l$ equal to a power of 2—a property that it preserves in its recursive calls (see Exercise 5.27).

```

rule(int l, int r, int h)
{ int m = (l+r)/2;
  if (h > 0)
  {
    rule(l, m, h-1);
    mark(m, h);
    rule(m, r, h-1);
  }
}

```

illustrates that the method makes the marks in order, from left to right—the trick lies in computing the lengths. The recursion tree in the figure helps us to understand the computation: Reading down, we see that the length of the mark decreases by 1 for each recursive function call. Reading across, we get the marks in the order that they are drawn, because, for any given node, we first draw the marks associated with the function call on the left, then the mark associated with the node, then the marks associated with the function call on the right.

We see immediately that the sequence of lengths is precisely the same as the sequence of disks moved for the towers of Hanoi problem. Indeed, a simple proof that they are identical is that the recursive programs are the same. Put another way, our monks could use the marks on a ruler to decide which disk to move.

Moreover, both the towers of Hanoi solution in Program 5.7 and the ruler-drawing program in Program 5.8 are variants of the basic divide-and-conquer scheme exemplified by Program 5.6. All three solve a problem of size 2^n by dividing it into two problems of size 2^{n-1} . For finding the maximum, we have a linear-time solution in the size of the input; for drawing a ruler and for solving the towers of Hanoi, we have a linear-time solution in the size of the output. For the towers of Hanoi, we normally think of the solution as being

exponential time, because we measure the size of the problem in terms of the number of disks, n .

It is easy to draw the marks on a ruler with a recursive program, but is there some simpler way to compute the length of the i th mark, for any given i ? Figure 5.9 shows yet another simple computational process that provides the answer to this question. The i th number printed out by both the towers of Hanoi program and the ruler program is nothing other than the number of trailing 0 bits in the binary representation of i . We can prove this property by induction by correspondence with a divide-and-conquer formulation for the process of printing the table of n -bit numbers: Print the table of $(n - 1)$ -bit numbers, each preceded by a 0 bit, then print the table of $(n - 1)$ -bit numbers each preceded by a 1-bit (see Exercise 5.25).

For the towers of Hanoi problem, the implication of the correspondence with n -bit numbers is a simple algorithm for the task. We can move the pile one peg to the right by iterating the following two steps until done:

- Move the small disk to the right if n is odd (left if n is even).
- Make the only legal move not involving the small disk.

That is, after we move the small disk, the other two pegs contain two disks, one smaller than the other. The only legal move not involving the small disk is to move the smaller one onto the larger one. Every other move involves the small disk for the same reason that every other number is odd and that every other mark on the rule is the shortest. Perhaps our monks *do* know this secret, because it is hard to imagine how they might be deciding which moves to make otherwise.

A formal proof by induction that every other move in the towers of Hanoi solution involves the small disk (beginning and ending with such moves) is instructive: For $n = 1$, there is just one move, involving the small disk, so the property holds. For $n > 1$, the assumption that the property holds for $n - 1$ implies that it holds for n by the recursive construction: The first solution for $n - 1$ begins with a small-disk move, and the second solution for $n - 1$ ends with a small-disk move, so the solution for n begins and ends with a small-disk move. We put a move not involving the small disk in between two moves that do involve the small disk (the move ending the first solution for $n - 1$ and the move beginning the second solution for $n - 1$), so the property that every other move involves the small disk is preserved.

0	0	0	0	1	
0	0	0	1	0	1
0	0	0	1	1	
0	0	1	0	0	2
0	0	1	0	1	
0	0	1	1	0	1
0	0	1	1	1	
0	1	0	0	0	3
0	1	0	0	1	
0	1	0	1	0	1
0	1	0	1	1	
0	1	1	0	0	2
0	1	1	0	1	
0	1	1	1	0	1
0	1	1	1	1	
1	0	0	0	0	4
1	0	0	0	1	
1	0	0	1	0	1
1	0	0	1	1	
1	0	1	0	0	2
1	0	1	0	1	
1	0	1	1	0	1
1	0	1	1	1	
1	1	0	0	0	3
1	1	0	0	1	
1	1	0	1	0	1
1	1	0	1	1	
1	1	1	0	0	2
1	1	1	0	1	
1	1	1	1	0	1
1	1	1	1	1	

Figure 5.9
Binary counting and the ruler function

Computing the ruler function is equivalent to counting the number of trailing zeros in the even N -bit numbers.

Program 5.9 Nonrecursive program to draw a ruler

In contrast to Program 5.8, we can also draw a ruler by first drawing all the marks of length 1, then drawing all the marks of length 2, and so forth. The variable t carries the length of the marks and the variable j carries the number of marks in between two successive marks of length t . The outer for loop increments t and preserves the property $j = 2^{t-1}$. The inner for loop draws all the marks of length t .

```
rule(int l, int r, int h)
{
    int i, j, t;
    for (t = 1, j = 1; t <= h; j += j, t++)
        for (i = 0; l+j+i <= r; i += j+j)
            mark(l+j+i, t);
}
```

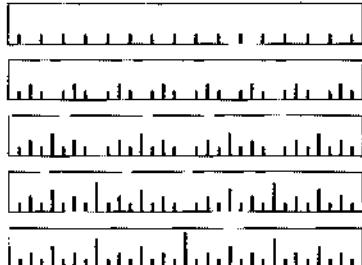


Figure 5.10
Drawing a ruler in bottom-up order

To draw a ruler nonrecursively, we alternate drawing marks of length 1 and skipping positions, then alternate drawing marks of length 2 and skipping remaining positions, then alternate drawing marks of length 3 and skipping remaining positions, and so forth.

Program 5.9 is an alternate way to draw a ruler that is inspired by the correspondence to binary numbers (see Figure 5.10). We refer to this version of the algorithm as a *bottom-up* implementation. It is not recursive, but it is certainly suggested by the recursive algorithm. This correspondence between divide-and-conquer algorithms and the binary representations of numbers often provides insights for analysis and development of improved versions, such as bottom-up approaches. We consider this perspective to understand, and possibly to improve, each of the divide-and-conquer algorithms that we examine.

The bottom-up approach involves rearranging the *order* of the computation when we are drawing a ruler. Figure 5.11 shows another example, where we rearrange the order of the three function calls in the recursive implementation. It reflects the recursive computation in the way that we first described it: Draw the middle mark, then draw the left half, then draw the right half. The pattern of drawing the marks is complex, but is the result of simply exchanging two statements in Program 5.8. As we shall see in Section 5.6, the relationship between Figures 5.8 and 5.11 is akin to the distinction between postfix and prefix in arithmetic expressions.

Drawing the marks in order as in Figure 5.8 might be preferable to doing the rearranged computations contained in Program 5.9 and indicated in Figure 5.11, because we can draw an arbitrarily long

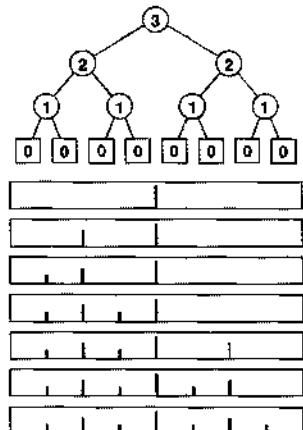
ruler, if we imagine a drawing device that simply moves on to the next mark in a continuous scroll. Similarly, to solve the towers of Hanoi problem, we are constrained to produce the sequence of disk moves in the order that they are to be performed. In general, many recursive programs depend on the subproblems being solved in a particular order. For other computations (see, for example, Program 5.6), the order in which we solve the subproblems is irrelevant. For such computations, the only constraint is that we must solve the subproblems before we can solve the main problem. Understanding when we have the flexibility to reorder the computation not only is a secret to success in algorithm design, but also has direct practical effects in many contexts. For example, this matter is critical when we consider implementing algorithms on parallel processors.

The bottom-up approach corresponds to the general method of algorithm design where we solve a problem by first solving trivial subproblems, then combining those solutions to solve slightly bigger subproblems, and so forth, until the whole problem is solved. This approach might be called *combine and conquer*.

It is a small step from drawing rulers to drawing two-dimensional patterns such as Figure 5.12. This figure illustrates how a simple recursive description can lead to a computation that appears to be complex (see Exercise 5.30).

Recursively defined geometric patterns such as Figure 5.12 are sometimes called *fractals*. If more complicated drawing primitives are used, and more complicated recursive invocations are involved (especially including recursively-defined functions on reals and in the complex plane), patterns of remarkable diversity and complexity can be developed. Another example, demonstrated in Figure 5.13, is the *Koch star*, which is defined recursively as follows: A Koch star of order 0 is the simple hill example of Figure 4.3, and a Koch star of order n is a Koch star of order $n - 1$ with each line segment replaced by the star of order 0, scaled appropriately.

Like the ruler-drawing and the towers of Hanoi solutions, these algorithms are linear in the number of steps, but that number is exponential in the maximum depth of the recursion (see Exercises 5.29 and 5.33). They also can be directly related to counting in an appropriate number system (see Exercise 5.34).



```

rule(0, 8, 3)
mark(4, 3)
rule(0, 4, 2)
mark(2, 2)
rule(0, 2, 1)
mark(1, 1)
rule(0, 1, 0)
rule(1, 2, 0)
rule(2, 4, 1)
mark(3, 1)
rule(2, 3, 0)
rule(3, 4, 0)
rule(4, 8, 2)
mark(6, 2)
rule(4, 6, 1)
mark(5, 1)
rule(4, 5, 0)
rule(5, 6, 0)
rule(6, 8, 1)
mark(7, 1)
rule(6, 7, 0)
rule(7, 8, 0)

```

Figure 5.11
Ruler-drawing function calls
(preorder version)

This sequence indicates the result of drawing marks before the recursive calls, instead of in between them.

Table 5.1 Basic divide-and-conquer algorithms

Binary search (see Chapters 2 and 12) and mergesort (see Chapter 8) are prototypical divide-and-conquer algorithms that provide guaranteed optimal performance for searching and sorting, respectively. The recurrences indicate the nature of the divide-and-conquer computation for each algorithm. (See Sections 2.5 and 2.6 for derivations of the solutions in the rightmost column.) Binary search splits a problem in half, does 1 comparison, then makes a recursive call for one of the halves. Mergesort splits a problem in half, then works on both halves recursively, then does N comparisons. Throughout the book, we shall consider numerous other algorithms developed with these recursive schemes.

	recurrence	approximate solution
binary search		
comparisons	$C_N = C_{N/2} + 1$	$\lg N$
mergesort		
recursive calls	$A_N = 2A_{N/2} + 1$	N
comparisons	$C_N = 2C_{N/2} + N$	$N \lg N$

The towers of Hanoi problem, ruler-drawing problem, and fractals are amusing; and the connection to binary numbers is surprising, but our primary interest in all of these topics is that they provide us with insights in understanding the basic algorithm design paradigm of divide in half and solve one or both halves independently, which is perhaps the most important such technique that we consider in this book. Table 5.1 includes details about binary search and mergesort, which not only are important and widely used practical algorithms, but also exemplify the divide-and-conquer algorithm design paradigm.

Quicksort (see Chapter 7) and binary-tree search (see Chapter 12) represent a significant variation on the basic divide-and-conquer theme where the problem is split into subproblems of size $k - 1$ and $N - k$, for some value k , which is determined by the input. For random input, these algorithms divide a problem into subproblems that are half the size (as in mergesort or in binary search) *on the average*. We study the analysis of the effects of this difference when we discuss these algorithms.

Other variations on the basic theme that are worthy of consideration include these: divide into parts of varying size, divide into more than two parts, divide into overlapping parts, and do various amounts of work in the nonrecursive part of the algorithm. In general, divide-and-conquer algorithms involve doing work to split the input into pieces, or to merge the results of processing two independent solved portions of the input, or to help things along after half of the input has been processed. That is, there may be code before, after, or in between the two recursive calls. Naturally, such variations lead to algorithms more complicated than are binary search and mergesort, and are more difficult to analyze. We consider numerous examples in this book; we return to advanced applications and analysis in Part 8.

Exercises

5.16 Write a recursive program that finds the maximum element in an array, based on comparing the first element in the array against the maximum element in the rest of the array (computed recursively).

5.17 Write a recursive program that finds the maximum element in a linked list.

5.18 Modify the divide-and-conquer program for finding the maximum element in an array (Program 5.6) to divide an array of size N into one part of size $k = 2^{\lceil \lg N \rceil - 1}$ and another of size $N - k$ (so that the size of at least one of the parts is a power of 2).

5.19 Draw the tree corresponding to the recursive calls that your program from Exercise 5.18 makes when the array size is 11.

• **5.20** Prove by induction that the number of function calls made by any divide-and-conquer algorithm that divides a problem into parts that constitute the whole, then solves the parts recursively, is linear.

• **5.21** Prove that the recursive solution to the towers of Hanoi problem (Program 5.7) is optimal. That is, show that any solution *requires* at least $2^N - 1$ moves.

▷ **5.22** Write a recursive program that computes the length of the i th mark in a ruler with $2^n - 1$ marks.

• **5.23** Examine tables of n -bit numbers, such as Figure 5.9, to discover a property of the i th number that determines the direction of the i th move (indicated by the sign bit in Figure 5.7) for solving the towers of Hanoi problem.

5.24 Write a program that produces a solution to the towers of Hanoi problem by filling in an array that holds all the moves, as in Program 5.9.

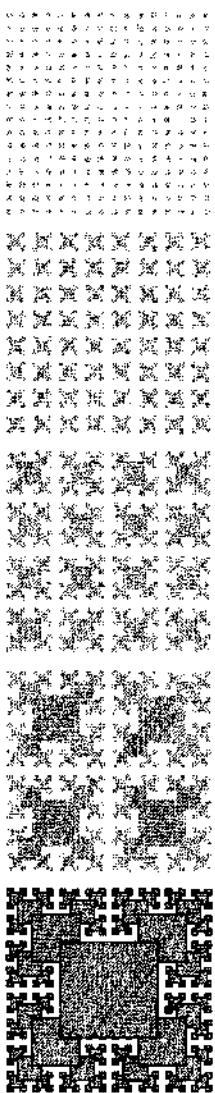
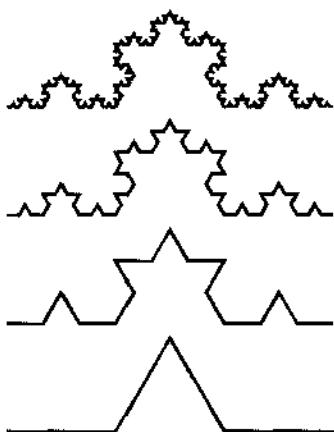


Figure 5.12
Two-dimensional fractal star

This fractal is a two-dimensional version of Figure 5.10. The outlined boxes in the bottom diagram highlight the recursive structure of the computation.



```

/kochR
{
  2 copy ge {dup 0 rlineto }
  {
    3 div
    2 copy kochR
    60 rotate
    2 copy kochR
    -120 rotate
    2 copy kochR
    60 rotate
    2 copy kochR
  } ifelse
  pop pop
} def
0 0 moveto
27 81 kochR
0 27 moveto
9 81 kochR
0 54 moveto
3 81 kochR
0 81 moveto
1 81 kochR
stroke

```

Figure 5.13
Recursive PostScript for Koch fractal

This modification to the PostScript program of Figure 4.3 transforms the output into a fractal (see text).

- 5.25 Write a recursive program that fills in an n -by- 2^n array with 0s and 1s such that the array represents all the n -bit binary numbers, as depicted in Figure 5.9.
- 5.26 Draw the results of using the recursive ruler-drawing program (Program 5.8) for these unintended values of the arguments: `rule(0, 11, 4)`, `rule(4, 20, 4)`, and `rule(7, 30, 5)`.
- 5.27 Prove the following fact about the ruler-drawing program (Program 5.8): If the difference between its first two arguments is a power of 2, then both of its recursive calls have this property also.
- 5.28 Write a function that computes efficiently the number of trailing 0s in the binary representation of an integer.
- 5.29 How many squares are there in Figure 5.12 (counting the ones that are covered up by bigger squares)?
- 5.30 Write a recursive C program that outputs a PostScript program that draws the bottom diagram in Figure 5.12, in the form of a list of function calls `x y r box`, which draws an r -by- r square at (x, y) . Implement `box` in PostScript (see Section 4.3).
- 5.31 Write a bottom-up nonrecursive program (similar to Program 5.9) that draws the bottom diagram in Figure 5.12, in the manner described in Exercise 5.30.
- 5.32 Write a PostScript program that draws the bottom diagram in Figure 5.12.
- ▷ 5.33 How many line segments are there in a Koch star of order n ?
- 5.34 Drawing a Koch star of order n amounts to executing a sequence of commands of the form “rotate α degrees, then draw a line segment of length $1/3^n$.” Find a correspondence with number systems that gives you a way to draw the star by incrementing a counter, then computing the angle α from the counter value.
- 5.35 Modify the Koch star program in Figure 5.13 to produce a different fractal based on a five-line figure for order 0, defined by 1-unit moves east, north, east, south, and east, in that order (see Figure 4.3).
- 5.36 Write a recursive divide-and-conquer function to draw an approximation to a line segment in an integer coordinate space, given the endpoints. Assume that all coordinates are between 0 and M . Hint: First plot a point close to the middle.

5.3 Dynamic Programming

An essential characteristic of the divide-and-conquer algorithms that we considered in Section 5.2 is that they partition the problem into independent subproblems. When the subproblems are not independent,

the situation is more complicated, primarily because direct recursive implementations of even the simplest algorithms of this type can require unthinkable amounts of time. In this section, we consider a systematic technique for avoiding this pitfall for an important class of problems.

For example, Program 5.10 is a direct recursive implementation of the recurrence that defines the Fibonacci numbers (see Section 2.3). *Do not use this program:* It is spectacularly inefficient. Indeed, the number of recursive calls to compute F_N is exactly F_{N+1} . But F_N is about ϕ^N , where $\phi \approx 1.618$ is the golden ratio. The awful truth is that Program 5.10 is an *exponential-time* algorithm for this trivial computation. Figure 5.14, which depicts the recursive calls for a small example, makes plain the amount of recomputation that is involved.

By contrast, it is easy to compute F_N in *linear* (proportional to N) time, by computing the first N Fibonacci numbers and storing them in an array:

```
F[0] = 0; F[1] = 1;  
for (i = 2; i <= N; i++)  
    F[i] = F[i-1] + F[i-2];
```

The numbers grow exponentially, so the array is small—for example, $F_{45} = 1836311903$ is the largest Fibonacci number that can be represented as a 32-bit integer, so an array of size 46 will do.

This technique gives us an immediate way to get numerical solutions for any recurrence relation. In the case of Fibonacci numbers, we can even dispense with the array, and keep track of just the previous two values (see Exercise 5.37); for many other commonly encountered recurrences (see, for example, Exercise 5.40), we need to maintain the array with all the known values.

A recurrence is a recursive function with integer values. Our discussion in the previous paragraph leads to the conclusion that we can evaluate any such function by computing all the function values in order starting at the smallest, using previously computed values at each step to compute the current value. We refer to this technique as *bottom-up dynamic programming*. It applies to any recursive computation, provided that we can afford to save all the previously computed values. It is an algorithm-design technique that has been used successfully for a wide range of problems. We have to pay attention to a

Program 5.10 Fibonacci numbers (recursive implementation)

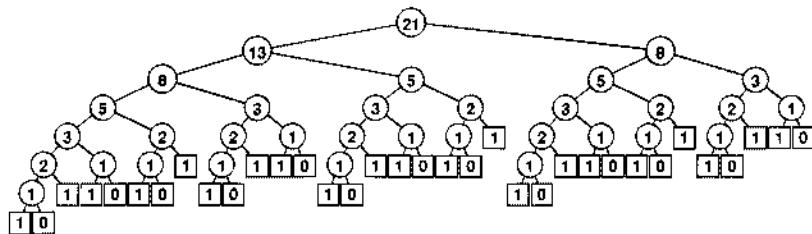
This program, although compact and elegant, is not usable because it takes exponential time to compute F_N . The running time to compute F_{N+1} is $\phi \approx 1.6$ times as long as the running time to compute F_N . For example, since $\phi^9 > 60$, if we notice that our computer takes about a second to compute F_N , we know that it will take more than a minute to compute F_{N+9} and more than an hour to compute F_{N+18} .

```
int F(int i)
{
    if (i < 1) return 0;
    if (i == 1) return 1;
    return F(i-1) + F(i-2);
}
```

simple technique that can improve the running time of an algorithm from exponential to linear!

Top-down dynamic programming is an even simpler view of the technique that allows us to execute recursive functions at the same cost as (or less cost than) bottom-up dynamic programming, in an automatic way. We instrument the recursive program to save each value that it computes (as its final action), and to check the saved values to avoid recomputing any of them (as its first action). Program 5.11 is the mechanical transformation of Program 5.10 that reduces its running time to be linear via top-down dynamic programming. Figure 5.15 shows the drastic reduction in the number of recursive calls achieved by this simple automatic change. Top-down dynamic programming is also sometimes called *memoization*.

For a more complicated example, consider the *knapsack problem*: A thief robbing a safe finds it filled with N types of items of varying size and value, but has only a small knapsack of capacity M to use to carry the goods. The knapsack problem is to find the combination of items which the thief should choose for the knapsack in order to maximize the total value of all the stolen items. For example, with the item types depicted in Figure 5.16, a thief with a knapsack of size 17 can take five A's (but not six) for a total take of 20, or a D and an E for a total take of 24, or one of many other combinations. Our goal is



to find an efficient algorithm that somehow finds the maximum among all the possibilities, given any set of items and knapsack capacity.

There are many applications in which solutions to the knapsack problem are important. For example, a shipping company might wish to know the best way to load a truck or cargo plane with items for shipment. In such applications, other variants to the problem might arise as well: for example, there might be a limited number of each kind of item available, or there might be two trucks. Many such variants can be handled with the same approach that we are about to examine for solving the basic problem just stated; others turn out to be much more difficult. There is a fine line between feasible and infeasible problems of this type, which we shall examine in Part 8.

In a recursive solution to the knapsack problem, each time that we choose an item, we assume that we can (recursively) find an optimal way to pack the rest of the knapsack. For a knapsack of size `cap`, we determine, for each item `i` among the available item types, what total value we could carry by placing `i` in the knapsack with an optimal packing of other items around it. That optimal packing is simply the one we have discovered (or will discover) for the smaller knapsack of size `cap-items[i].size`. This solution exploits the principle that optimal decisions, once made, do not need to be changed. Once we know how to pack knapsacks of smaller capacities with optimal sets of items, we do not need to reexamine those problems, regardless of what the next items are.

Program 5.12 is a direct recursive solution based on this discussion. Again, this program is not feasible for use in solving actual problems, because it takes exponential time due to massive recomputation (see Figure 5.17), but we can automatically apply top-down dynamic programming to eliminate this problem, as shown in Pro-

Figure 5.14
Structure of recursive algorithm for Fibonacci numbers

The picture of the recursive calls needed to used to compute F_8 by the standard recursive algorithm illustrates how recursion with overlapping subproblems can lead to exponential costs. In this case, the second recursive call ignores the computations done during the first, which results in massive recompuation because the effect multiplies recursively. The recursive calls to compute $F_8 = 8$ (which are reflected in the right subtree of the root and the left subtree of the left subtree of the root) are listed below.

8 F(6)
5 F(5)
3 F(4)
2 F(3)
1 F(2)
1 F(1)
0 F(0)
1 F(1)
1 F(2)
1 F(1)
0 F(0)
2 F(3)
1 F(2)
1 F(1)
0 F(0)
1 F(1)
3 F(4)
2 F(3)
1 F(2)
1 F(1)
0 F(0)
1 F(1)
1 F(2)
1 F(1)
0 F(0)

Program 5.11 Fibonacci numbers (dynamic programming)

By saving the values that we compute in an array external to the recursive procedure, we explicitly avoid any recomputation. This program computes F_N in time proportional to N , in stark contrast to the $O(\phi^N)$ time used by Program 5.10.

```
int F(int i)
{ int t;
  if (knownF[i] != unknown) return knownF[i];
  if (i == 0) t = 0;
  if (i == 1) t = 1;
  if (i > 1) t = F(i-1) + F(i-2);
  return knownF[i] = t;
}
```

gram 5.13. As before, this technique eliminates all recomputation, as shown in Figure 5.18.

By design, dynamic programming eliminates all recomputation in *any* recursive program, subject only to the condition that we can afford to save the values of the function for arguments smaller than the call in question.

Property 5.3 *Dynamic programming reduces the running time of a recursive function to be at most the time required to evaluate the function for all arguments less than or equal to the given argument, treating the cost of a recursive call as constant.*

See Exercise 5.50. ■

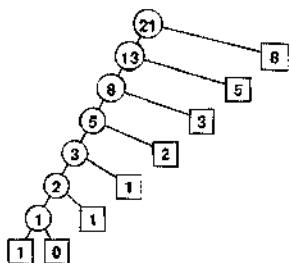


Figure 5.15
Top-down dynamic programming for computing Fibonacci numbers

This picture of the recursive calls used to compute F_5 by the top-down dynamic programming implementation of the recursive algorithm illustrates how saving computed values cuts the cost from exponential (see Figure 5.14) to linear.

For the knapsack problem, this property implies that the running time is proportional to NM . Thus, we can solve the knapsack problem easily when the capacity is not huge; for huge capacities, the time and space requirements may be prohibitively large.

Bottom-up dynamic programming applies to the knapsack problem, as well. Indeed, we can use the bottom-up approach any time that we use the top-down approach, although we need to take care to ensure that we compute the function values in an appropriate order, so that each value that we need has been computed when we need it. For functions with single integer arguments such as the two that we have

Program 5.12 Knapsack problem (recursive implementation)

As we warned about the recursive solution to the problem of computing the Fibonacci numbers, *do not use this program*, because it will take exponential time and therefore may not ever run to completion even for small problems. It does, however, represent a compact solution that we can improve easily (see Program 5.13). This code assumes that items are structures with a size and a value, defined with

```
typedef struct { int size; int val; } Item;
```

and that we have an array of N items of type Item. For each possible item, we calculate (recursively) the maximum value that we could achieve by including that item, then take the maximum of all those values.

```
int knap(int cap)
{ int i, space, max, t;
  for (i = 0, max = 0; i < N; i++)
    if ((space = cap-items[i].size) >= 0)
      if ((t = knap(space) + items[i].val) > max)
        max = t;
  return max;
}
```

considered, we simply proceed in increasing order of the argument (see Exercise 5.53); for more complicated recursive functions, determining a proper order can be a challenge.

For example, we do not need to restrict ourselves to recursive functions with single integer arguments. When we have a function with multiple integer arguments, we can save solutions to smaller subproblems in multidimensional arrays, one for each argument. Other situations involve no integer arguments at all, but rather use an abstract discrete problem formulation that allows us to decompose problems into smaller ones. We shall consider examples of such problems in Parts 5 through 8.

In top-down dynamic programming, we save known values; in bottom-up dynamic programming, we precompute them. We generally prefer top-down to bottom-up dynamic programming, because

- It is a mechanical transformation of a natural problem solution.
- The order of computing the subproblems takes care of itself.
- We may not need to compute answers to all the subproblems.

	0	1	2	3	4
item	A	B	C	D	E
size	3	4	7	8	9
val	4	5	10	11	13

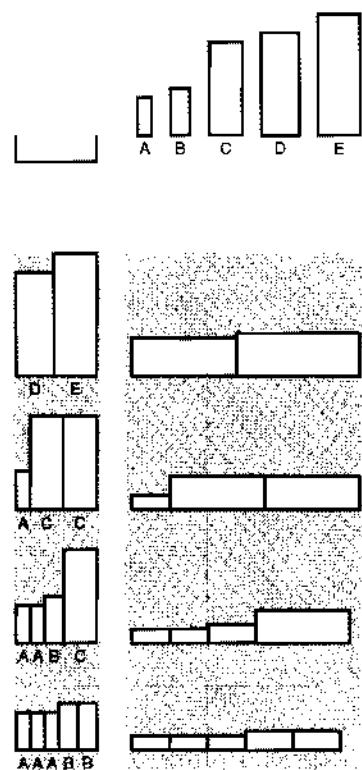


Figure 5.16
Knapsack example

An instance of the knapsack problem (top) consists of a knapsack capacity and a set of items of varying size (horizontal dimension) and value (vertical dimension). This figure shows four different ways to fill a knapsack of size 17, two of which lead to the highest possible total value of 24.

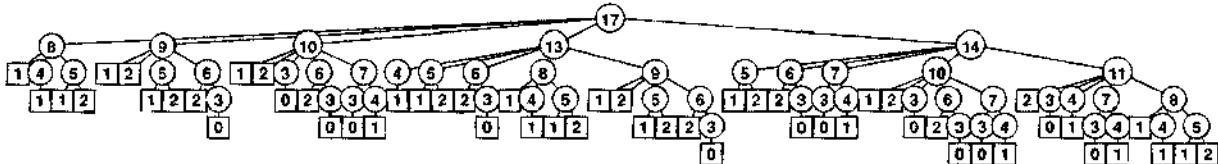


Figure 5.17
Recursive structure of knapsack algorithm.

This tree represents the recursive call structure of the simple recursive knapsack algorithm in Program 5.12. The number in each node represents the remaining capacity in the knapsack. The algorithm suffers the same basic problem of exponential performance due to massive recomputation for overlapping subproblems that we considered in computing Fibonacci numbers (see Figure 5.14).

Dynamic-programming applications differ in the nature of the subproblems and in the amount of information that we need to save regarding the subproblems.

A crucial point that we cannot overlook is that dynamic programming becomes ineffective when the number of possible function values that we might need is so high that we cannot afford to save (top-down) or precompute (bottom-up) all of them. For example, if M and the item sizes are 64-bit quantities or floating-point numbers in the knapsack problem, we will not be able to save values by indexing into an array. This distinction causes more than a minor annoyance—it poses a fundamental difficulty. No good solution is known for such problems; we will see in Part 8 that there is good reason to believe that no good solution exists.

Dynamic programming is an algorithm-design technique that is primarily suited for the advanced problems of the type that we shall consider in Parts 5 through 8. Most of the algorithms that we discuss in Parts 2 through 4 are divide-and-conquer methods with nonoverlapping subproblems, and we are focusing on subquadratic or sublinear, rather than subexponential, performance. However, top-down dynamic programming is a basic technique for developing efficient implementations of recursive algorithms that belongs in the toolbox of anyone engaged in algorithm design and implementation.

Exercises

- ▷ 5.37 Write a function that computes $F_N \bmod M$, using only a constant amount of space for intermediate calculations.
- 5.38 What is the largest N for which F_N can be represented as a 64-bit integer?
- 5.39 Draw the tree corresponding to Figure 5.15 for the case where we exchange the recursive calls in Program 5.11.
- 5.40 Write a function that uses bottom-up dynamic programming to compute the value of P_N defined by the recurrence

$$P_N = \lfloor N/2 \rfloor + P_{\lfloor N/2 \rfloor} + P_{\lceil N/2 \rceil}, \quad \text{for } N \geq 1 \text{ with } P_0 = 0.$$

Program 5.13 Knapsack problem (dynamic programming)

This mechanical modification to the code of Program 5.12 reduces the running time from exponential to linear. We simply save any function values that we compute, then retrieve any saved values whenever we need them (using a sentinel value to represent unknown values), rather than making recursive calls. We save the index of the item, so that we can reconstruct the contents of the knapsack after the computation, if we wish: `itemKnown[M]` is in the knapsack, the remaining contents are the same as for the optimal knapsack of size $M - \text{itemKnown}[M].\text{size}$ so `itemKnown[M-items[M].size]` is in the knapsack, and so forth.

```
int knap(int M)
{
    int i, space, max, maxi, t;
    if (maxKnown[M] != unknown) return maxKnown[M];
    for (i = 0, max = 0; i < N; i++)
        if ((space = M-items[i].size) >= 0)
            if ((t = knap(space) + items[i].val) > max)
                { max = t; maxi = i; }
    maxKnown[M] = max; itemKnown[M] = items[maxi];
    return max;
}
```

Draw a plot of N versus $P_N - N \lg N / 2$ for $0 \leq N \leq 1024$.

5.41 Write a function that uses top-down dynamic programming to solve Exercise 5.40.

○ 5.42 Draw the tree corresponding to Figure 5.15 for your function from Exercise 5.41, when invoked for $N = 23$.

5.43 Draw a plot of N versus the number of recursive calls that your function from Exercise 5.41 makes to compute P_N , for $0 \leq N \leq 1024$. (For the purposes of this calculation, start your program from scratch for each N .)

5.44 Write a function that uses bottom-up dynamic programming to compute the value of C_N defined by the recurrence

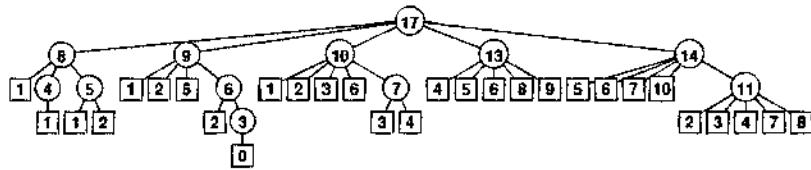
$$C_N = N + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}), \quad \text{for } N \geq 1 \text{ with } C_0 = 1.$$

5.45 Write a function that uses top-down dynamic programming to solve Exercise 5.44.

○ 5.46 Draw the tree corresponding to Figure 5.15 for your function from Exercise 5.45, when invoked for $N = 23$.

Figure 5.18
Top-down dynamic programming for knapsack algorithm

As it did for the Fibonacci numbers computation, the technique of saving known values reduces the cost of the knapsack algorithm from exponential (see Figure 5.17) to linear.



5.47 Draw a plot of N versus the number of recursive calls that your function from Exercise 5.45 makes to compute C_N , for $0 \leq N \leq 1024$. (For the purposes of this calculation, start your program from scratch for each N .)

- ▷ 5.48 Give the contents of the arrays `maxKnown` and `itemKnown` that are computed by Program 5.13 for the call `knap(17)` with the items in Figure 5.18.
- ▷ 5.49 Give the tree corresponding to Figure 5.18 under the assumption that the items are considered in decreasing order of their size.
- 5.50 Prove Property 5.3.
- 5.51 Write a function that solves the knapsack problem using a bottom-up dynamic programming version of Program 5.12.
- 5.52 Write a function that solves the knapsack problem using top-down dynamic programming, but using a recursive solution based on computing the optimal number of a particular item to include in the knapsack, based on (recursively) knowing the optimal way to pack the knapsack without that item.
- 5.53 Write a function that solves the knapsack problem using a bottom-up dynamic programming version of the recursive solution described in Exercise 5.52.
- 5.54 Use dynamic programming to solve Exercise 5.4. Keep track of the total number of function calls that you save.

5.55 Write a program that uses top-down dynamic programming to compute the binomial coefficient $\binom{N}{k}$, based on the recursive rule

$$\binom{N}{k} = \binom{N-1}{k} + \binom{N-1}{k-1}$$

with $\binom{N}{0} = \binom{N}{N} = 1$.

5.4 Trees

Trees are a mathematical abstraction that play a central role in the design and analysis of algorithms because

- We use trees to describe dynamic properties of algorithms.

- We build and use explicit data structures that are concrete realizations of trees.

We have already seen examples of both of these uses. We designed algorithms for the connectivity problem that are based on tree structures in Chapter 1, and we described the call structure of recursive algorithms with tree structures in Sections 5.2 and 5.3.

We encounter trees frequently in everyday life—the basic concept is a familiar one. For example, many people keep track of ancestors or descendants with a family tree; as we shall see, much of our terminology is derived from this usage. Another example is found in the organization of sports tournaments; this usage was studied by Lewis Carroll, among others. A third example is found in the organizational chart of a large corporation; this usage is suggestive of the hierarchical decomposition that characterizes divide-and-conquer algorithms. A fourth example is a parse tree of an English sentence into its constituent parts; such trees are intimately related to the processing of computer languages, as discussed in Part 5. Figure 5.19 gives a typical example of a tree—one that describes the structure of this book. We touch on numerous other examples of applications of trees throughout the book.

In computer applications, one of the most familiar uses of tree structures is to organize file systems. We keep files in *directories* (which are also sometimes called *folders*) that are defined recursively as sequences of directories and files. This recursive definition again reflects a natural recursive decomposition, and is identical to the definition of a certain type of tree.

There are many different types of trees, and it is important to understand the distinction between the abstraction and the concrete representation with which we are working for a given application. Accordingly, we shall consider the different types of trees and their representations in detail. We begin our discussion by defining trees as abstract objects, and by introducing most of the basic associated terminology. We shall discuss informally the different types of trees that we need to consider in decreasing order of generality:

- Trees
- Rooted trees
- Ordered trees
- M -ary trees and binary trees

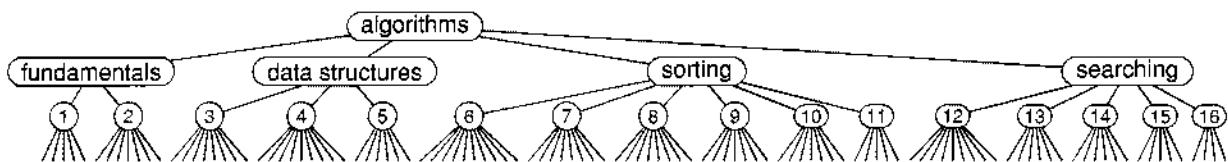


Figure 5.19
A tree

This tree depicts the parts, chapters, and sections in this book. There is a node for each entity. Each node is connected to its constituent parts by links down to them, and is connected to the large part to which it belongs by a link up to that part.

After developing a context with this informal discussion, we move to formal definitions and consider representations and applications. Figure 5.20 illustrates many of the basic concepts that we discuss and then define.

A *tree* is a nonempty collection of vertices and edges that satisfies certain requirements. A *vertex* is a simple object (also referred to as a *node*) that can have a name and can carry other associated information; an *edge* is a connection between two vertices. A *path* in a tree is a list of distinct vertices in which successive vertices are connected by edges in the tree. The defining property of a tree is that there is precisely one path connecting any two nodes. If there is more than one path between some pair of nodes, or if there is no path between some pair of nodes, then we have a graph; we do not have a tree. A disjoint set of trees is called a *forest*.

A *rooted tree* is one where we designate one node as the *root* of a tree. In computer science, we normally reserve the term *tree* to refer to rooted trees, and use the term *free tree* to refer to the more general structure described in the previous paragraph. In a rooted tree, any node is the root of a *subtree* consisting of it and the nodes below it.

There is exactly one path between the root and each of the other nodes in the tree. The definition implies no direction on the edges; we normally think of the edges as all pointing away from the root or all pointing towards the root, depending upon the application. We usually draw rooted trees with the root at the top (even though this convention seems unnatural at first), and we speak of node y as being *below* node x (and x as *above* y) if x is on the path from y to the root (that is, if y is below x as drawn on the page and is connected to x by a path that does not pass through the root). Each node (except the root) has exactly one node above it, which is called its *parent*; the nodes directly below a node are called its *children*. We sometimes carry the analogy to family trees further and refer to the *grandparent* or the *sibling* of a node.

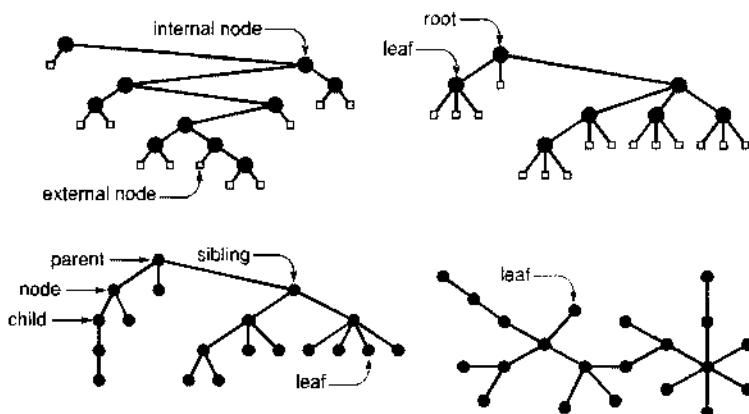


Figure 5.20
Types of trees

These diagrams show examples of a *binary tree* (top left), a *ternary tree* (top right), a *rooted tree* (bottom left), and a *free tree* (bottom right).

Nodes with no children are called *leaves*, or *terminal* nodes. To correspond to the latter usage, nodes with at least one child are sometimes called *nonterminal* nodes. We have seen an example in this chapter of the utility of distinguishing these types of nodes. In trees that we use to present the call structure of recursive algorithms (see, for example, Figure 5.14) the nonterminal nodes (circles) represent function invocations with recursive calls and the terminal nodes (squares) represent function invocations with no recursive calls.

In certain applications, the way in which the children of each node are ordered is significant; in other applications, it is not. An *ordered tree* is a rooted tree in which the order of the children at every node is specified. Ordered trees are a natural representation: for example, we place the children in some order when we draw a tree. As we shall see, this distinction is also significant when we consider representing trees in a computer.

If each node *must* have a specific number of children appearing in a specific order, then we have an *M-ary tree*. In such a tree, it is often appropriate to define special external nodes that have no children. Then, external nodes can act as dummy nodes for reference by nodes that do not have the specified number of children. In particular, the simplest type of *M-ary tree* is the *binary tree*. A *binary tree* is an ordered tree consisting of two types of nodes: external nodes with no children and internal nodes with exactly two children. Since the two children of each internal node are ordered, we refer to the *left child*

and the *right child* of internal nodes: every internal node must have both a left and a right child, although one or both of them might be an external node. A *leaf* in an M -ary tree is an internal node whose children are all external.

That is the basic terminology. Next, we shall consider formal definitions, representations, and applications of, in increasing order of generality,

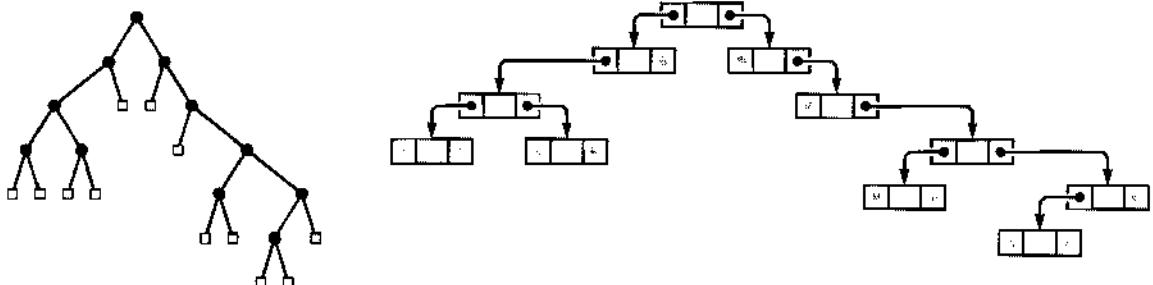
- Binary trees and M -ary trees
- Ordered trees
- Rooted trees
- Free trees

That is, a binary tree is a special type of ordered tree, an ordered tree is a special type of rooted tree, and a rooted tree is a special type of free tree. The different types of trees arise naturally in various applications, and it is important to be aware of the distinctions when we consider ways of representing trees with concrete data structures. By starting with the most specific abstract structure, we shall be able to consider concrete representations in detail, as will become clear.

Definition 5.1 *A binary tree is either an external node or an internal node connected to a pair of binary trees, which are called the left subtree and the right subtree of that node.*

This definition makes it plain that the binary tree itself is an abstract mathematical concept. When we are working with a computer representation, we are working with just one concrete realization of that abstraction. The situation is no different from representing real numbers with floats, integers with ints, and so forth. When we draw a tree with a node at the root connected by edges to the left subtree on the left and the right subtree on the right, we are choosing a convenient concrete representation. There are many different ways to represent binary trees (see, for example, Exercise 5.62) that are surprising at first, but, upon reflection, that are to be expected, given the abstract nature of the definition.

The concrete representation that we use most often when we implement programs that use and manipulate binary trees is a structure with two links (a left link and a right link) for internal nodes (see Figure 5.21). These structures are similar to linked lists, but they have two links per node, rather than one. Null links correspond to



external nodes. Specifically, we add a link to our standard linked list representation from Section 3.3, as follows:

```
typedef struct node *link;
struct node { Item item; link l, r; };
```

which is nothing more than C code for Definition 5.1. Links are references to nodes, and a node consists of an item and a pair of links. Thus, for example, we implement the abstract operation *move to the left subtree* with a pointer reference such as `x = x->l`.

This standard representation allows for efficient implementation of operations that call for moving *down* the tree from the root, but not for operations that call for moving *up* the tree from a child to its parent. For algorithms that require such operations, we might add a third link to each node, pointing to the parent. This alternative is analogous to a doubly linked list. As with linked lists (see Figure 3.6), we keep tree nodes in an array and use indices instead of pointers as links in certain situations. We examine a specific instance of such an implementation in Section 12.7. We use other binary-tree representations for certain specific algorithms, most notably in Chapter 9.

Because of all the different possible representations, we might develop a binary-tree ADT that encapsulates the important operations that we want to perform, and that separates the use and implementation of these operations. We do not take this approach in this book because

- We most often use the two-link representation.
- We use trees to implement higher-level ADTs, and wish to focus on those.
- We work with algorithms whose efficiency depends on a particular representation—a fact that might be lost in an ADT.

Figure 5.21
Binary-tree representation

The standard representation of a binary tree uses nodes with two links: a left link to the left subtree and a right link to the right subtree. Null links correspond to external nodes.

These are the same reasons that we use familiar concrete representations for arrays and linked lists. The binary-tree representation depicted in Figure 5.21 is a fundamental tool that we are now adding to this short list.

For linked lists, we began by considering elementary operations for inserting and deleting nodes (see Figures 3.3 and 3.4). For the standard representation of binary trees, such operations are not necessarily elementary, because of the second link. If we want to delete a node from a binary tree, we have to reconcile the basic problem that we may have two children to handle after the node is gone, but only one parent. There are three natural operations that do not have this difficulty: insert a new node at the bottom (replace a null link with a link to a new node), delete a leaf (replace the link to it by a null link), and combine two trees by creating a new root with a left link pointing to one tree and the right link pointing to the other one. We use these operations extensively when manipulating binary trees.

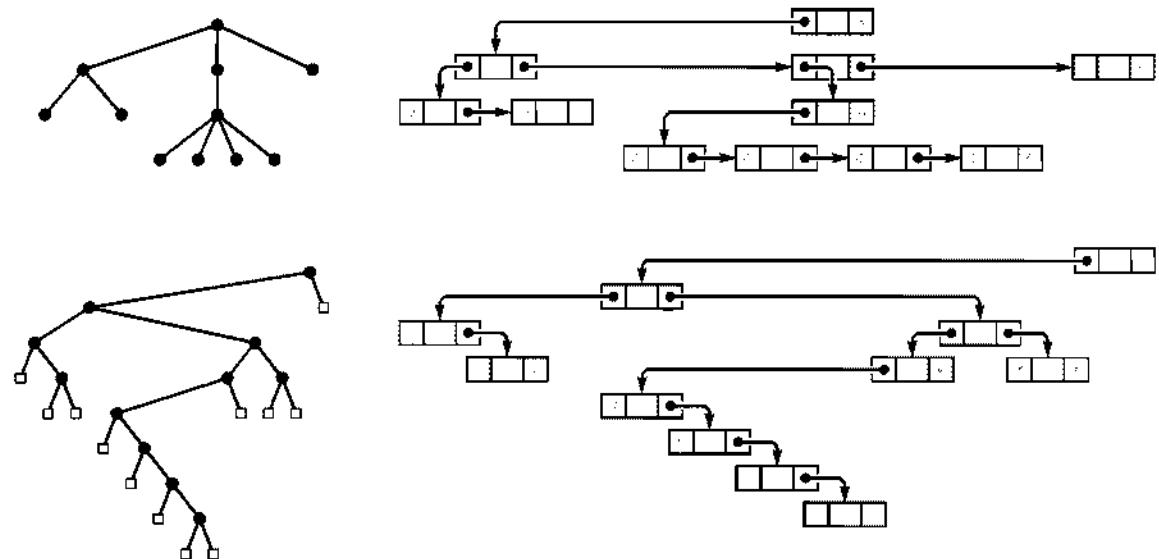
Definition 5.2 *An M-ary tree is either an external node or an internal node connected to an ordered sequence of M trees that are also M-ary trees.*

We normally represent nodes in M -ary trees either as structures with M named links (as in binary trees) or as arrays of M links. For example, in Chapter 15, we consider 3-ary (or *ternary*) trees where we use structures with three named links (left, middle, and right) each of which has specific meaning for associated algorithms. Otherwise, the use of arrays to hold the links is appropriate because the value of M is fixed, although, as we shall see, we have to pay particular attention to excessive use of space when using such a representation.

Definition 5.3 *A tree (also called an ordered tree) is a node (called the root) connected to a sequence of disjoint trees. Such a sequence is called a forest.*

The distinction between ordered trees and M -ary trees is that nodes in ordered trees can have any number of children, whereas nodes in M -ary trees must have precisely M children. We sometimes use the term *general tree* in contexts where we want to distinguish ordered trees from M -ary trees.

Because each node in an ordered tree can have any number of links, it is natural to consider using a linked list, rather than an array,



to hold the links to the node's children. Figure 5.22 is an example of such a representation. From this example, it is clear that each node then contains two links, one for the linked list connecting it to its siblings, the other for the linked list of its children.

Property 5.4 *There is a one-to-one correspondence between binary trees and ordered forests.*

The correspondence is depicted in Figure 5.22. We can represent any forest as a binary tree by making the left link of each node point to its leftmost child, and the right link of each node point to its sibling on the right. ■

Definition 5.4 *A rooted tree (or unordered tree) is a node (called the root) connected to a multiset of rooted trees. (Such a multiset is called an unordered forest.)*

The trees that we encountered in Chapter 1 for the connectivity problem are unordered trees. Such trees may be defined as ordered trees where the order in which the children of a node are considered is not significant. We could also choose to define unordered trees as comprising a set of parent–child relationships among nodes. This choice would seem to have little relation to the recursive structures

Figure 5.22
Tree representation

Representing an ordered tree by keeping a linked list of the children of each node is equivalent to representing it as a binary tree. The diagram on the right at the top shows a linked-list-of-children representation of the tree on the left at the top, with the list implemented in the right links of nodes, and each node's left link pointing to the first node in the linked list of its children. The diagram on the right at the bottom shows a slightly rearranged version of the diagram above it, and clearly represents the binary tree at the left on the bottom. That is, we can consider the binary tree as representing the tree.

that we are considering, but it is perhaps the concrete representation that is most true to the abstract notion.

We could choose to represent an unordered tree in a computer with an ordered tree, recognizing that many different ordered trees might represent the same unordered tree. Indeed, the converse problem of determining whether or not two different ordered trees represent the same unordered tree (the *tree-isomorphism* problem) is a difficult one to solve.

The most general type of tree is one where no root node is distinguished. For example, the spanning trees resulting from the connectivity algorithms in Chapter 1 have this property. To define properly *unrooted, unordered trees*, or *free trees*, we start with a definition for *graphs*.

Definition 5.5 *A graph is a set of nodes together with a set of edges that connect pairs of distinct nodes (with at most one edge connecting any pair of nodes).*

We can envision starting at some node and following an edge to the constituent node for the edge, then following an edge from that node to another node, and so on. A sequence of edges leading from one node to another in this way with no node appearing twice is called a *simple path*. A graph is *connected* if there is a simple path connecting any pair of nodes. A path that is simple except that the first and final nodes are the same is called a *cycle*.

Every tree is a graph; which graphs are trees? We consider a graph to be a tree if it satisfies any of the following four conditions:

- G has $N - 1$ edges and no cycles.
- G has $N - 1$ edges and is connected.
- Exactly one simple path connects each pair of vertices in G .
- G is connected, but does not remain connected if any edge is removed.

Any one of these conditions is necessary and sufficient to prove the other three. Formally, we should choose one of them to serve as a definition of a *free tree*; informally, we let them collectively serve as the definition.

We represent a free tree simply as a collection of edges. If we choose to represent a free tree as an unordered, ordered or even a binary tree, we need to recognize that, in general, there are many different ways to represent each free tree.

The tree abstraction arises frequently, and the distinctions discussed in this section are important, because knowing different tree abstractions is often an essential ingredient in finding an efficient algorithm and corresponding data structure for a given problem. We often work directly with concrete representations of trees without regard to a particular abstraction, but we also often profit from working with the proper tree abstraction, then considering various concrete representations. We shall see numerous examples of this process throughout the book.

Before moving back to algorithms and implementations, we consider a number of basic mathematical properties of trees; these properties will be of use to us in the design and analysis of tree algorithms.

Exercises

▷ 5.56 Give representations of the free tree in Figure 5.20 as a rooted tree and as a binary tree.

• 5.57 How many different ways are there to represent the free tree in Figure 5.20 as an ordered tree?

▷ 5.58 Draw three ordered trees that are isomorphic to the ordered tree in Figure 5.20. That is, you should be able to transform the four trees to one another by exchanging children.

○ 5.59 Assume that trees contain items for which `eq` is defined. Write a recursive program that deletes all the leaves in a binary tree with items equal to a given item (see Program 5.5).

○ 5.60 Change the divide-and conquer function for finding the maximum item in an array (Program 5.6) to divide the array into k parts that differ by at most 1 in size, recursively find the maximum in each part, and return the maximum of the maxima.

5.61 Draw the 3-ary and 4-ary trees corresponding to using $k = 3$ and $k = 4$ in the recursive construction suggested in Exercise 5.60, for an array of 11 elements (see Figure 5.6).

○ 5.62 Binary trees are equivalent to binary strings that have one more 0 bit than 1 bit, with the additional constraint that, at any position k , the number of 0 bits that appear strictly to the left of k is no larger than the number of 1 bits strictly to the left of k . A binary tree is either a 0 or two such strings

concatenated together, preceded by a 1. Draw the binary tree that corresponds to the string

1 1 1 0 0 1 0 1 1 0 0 0 1 0 1 1 0 0 0.

- 5.63 Ordered trees are equivalent to balanced strings of parentheses: An ordered tree either is null or is a sequence of ordered trees enclosed in parentheses. Draw the ordered tree that corresponds to the string

((() (() ()) ()) (() () ())).

- 5.64 Write a program to determine whether or not two arrays of N integers between 0 and $N - 1$ represent isomorphic unordered trees, when interpreted (as in Chapter 1) as parent-child links in a tree with nodes numbered between 0 and $N - 1$. That is, your program should determine whether or not there is a way to renumber the nodes in one tree such that the array representation of the one tree is identical to the array representation of the other tree.
- 5.65 Write a program to determine whether or not two binary trees represent isomorphic unordered trees.
- ▷ 5.66 Draw all the ordered trees that could represent the tree defined by the set of edges 0-1, 1-2, 1-3, 1-4, 4-5.
- 5.67 Prove that, if a connected graph of N nodes has the property that removing any edge disconnects the graph, then the graph has $N - 1$ edges and no cycles.

5.5 Mathematical Properties of Binary Trees

Before beginning to consider tree-processing algorithms, we continue in a mathematical vein by considering a number of basic properties of trees. We focus on binary trees, because we use them frequently throughout this book. Understanding their basic properties will lay the groundwork for understanding the performance characteristics of various algorithms that we will encounter—not only of those that use binary trees as explicit data structures, but also of divide-and-conquer recursive algorithms and other similar applications.

Property 5.5 *A binary tree with N internal nodes has $N + 1$ external nodes.*

We prove this property by induction: A binary tree with no internal nodes has one external node, so the property holds for $N = 0$. For $N > 0$, any binary tree with N internal nodes has k internal nodes in its left subtree and $N - 1 - k$ internal nodes in its right subtree for

some k between 0 and $N - 1$, since the root is an internal node. By the inductive hypothesis, the left subtree has $k + 1$ external nodes and the right subtree has $N - k$ external nodes, for a total of $N + 1$. ■

Property 5.6 *A binary tree with N internal nodes has $2N$ links: $N - 1$ links to internal nodes and $N + 1$ links to external nodes.*

In any rooted tree, each node, except the root, has a unique parent, and every edge connects a node to its parent, so there are $N - 1$ links connecting internal nodes. Similarly, each of the $N + 1$ external nodes has one link, to its unique parent. ■

The performance characteristics of many algorithms depend not just on the number of nodes in associated trees, but on various structural properties.

Definition 5.6 *The level of a node in a tree is one higher than the level of its parent (with the root at level 0). The height of a tree is the maximum of the levels of the tree's nodes. The path length of a tree is the sum of the levels of all the tree's nodes. The internal path length of a binary tree is the sum of the levels of all the tree's internal nodes. The external path length of a binary tree is the sum of the levels of all the tree's external nodes.*

A convenient way to compute the path length of a tree is to sum, for all k , the product of k and the number of nodes at level k .

These quantities also have simple recursive definitions that follow directly from the recursive definitions of trees and binary trees. For example, the height of a tree is 1 greater than the maximum of the height of the subtrees of its root, and the path length of a tree with N nodes is the sum of the path lengths of the subtrees of its root plus $N - 1$. The quantities also relate directly to the analysis of recursive algorithms. For example, for many recursive computations, the height of the corresponding tree is precisely the maximum depth of the recursion, or the size of the stack needed to support the computation.

Property 5.7 *The external path length of any binary tree with N internal nodes is $2N$ greater than the internal path length.*

We could prove this property by induction, but an alternate proof (which also works for Property 5.6) is instructive. Observe that any binary tree can be constructed by the following process: Start with the

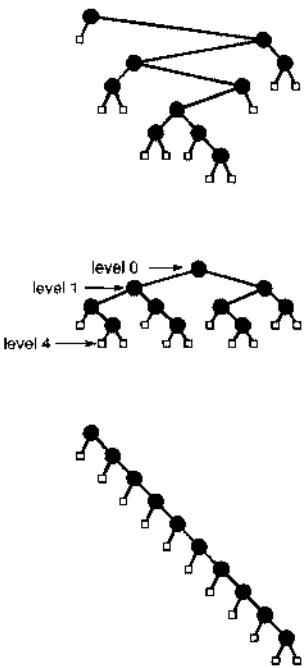


Figure 5.23
Three binary trees with 10 internal nodes

The binary tree shown at the top has height 7, internal path length 31 and external path length 51. A fully balanced binary tree (center) with 10 internal nodes has height 4, internal path length 19 and external path length 39 (no binary tree with 10 nodes has smaller values for any of these quantities). A degenerate binary tree (bottom) with 10 internal nodes has height 10, internal path length 45 and external path length 65 (no binary tree with 10 nodes has larger values for any of these quantities).

binary tree consisting of one external node. Then, repeat the following N times: Pick an external node and replace it by a new internal node with two external nodes as children. If the external node chosen is at level k , the internal path length is increased by k , but the external path length is increased by $k + 2$ (one external node at level k is removed, but two at level $k + 1$ are added). The process starts with a tree with internal and external path lengths both 0 and, for each of N steps, increases the external path length by 2 more than the internal path length. ■

Property 5.8 *The height of a binary tree with N internal nodes is at least $\lg N$ and at most $N - 1$.*

The worst case is a degenerate tree with only one leaf, with $N - 1$ links from the root to the leaf (see Figure 5.23). The best case is a balanced tree with 2^i internal nodes at every level i except the bottom level (see Figure 5.23). If the height is h , then we must have

$$2^{h-1} < N + 1 \leq 2^h,$$

since there are $N + 1$ external nodes. This inequality implies the property stated: The best-case height is precisely equal to $\lg N$ rounded up to the nearest integer. ■

Property 5.9 *The internal path length of a binary tree with N internal nodes is at least $N \lg(N/4)$ and at most $N(N - 1)/2$.*

The worst case and the best case are achieved for the same trees referred to in the discussion of Property 5.8 and depicted in Figure 5.23. The internal path length of the worst-case tree is $0 + 1 + 2 + \dots + (N - 1) = N(N - 1)/2$. The best case tree has $(N + 1)$ external nodes at height no more than $\lfloor \lg N \rfloor$. Multiplying these and applying Property 5.7, we get the bound $(N + 1)\lfloor \lg N \rfloor - 2N < N \lg(N/4)$. ■

As we shall see, binary trees appear extensively in computer applications, and performance is best when the binary trees are fully balanced (or nearly so). For example, the trees that we use to describe divide-and-conquer algorithms such as binary search and mergesort are fully balanced (see Exercise 5.74). In Chapters 9 and 13, we shall examine explicit data structures that are based on balanced trees.

These basic properties of trees provide the information that we need to develop efficient algorithms for a number of practical problems. More detailed analyses of several of the specific algorithms

that we shall encounter require sophisticated mathematical analysis, although we can often get useful estimates with straightforward inductive arguments like the ones that we have used in this section. We discuss further mathematical properties of trees as needed in the chapters that follow. At this point, we are ready to move back to algorithmic matters.

Exercises

- ▷ 5.68 How many external nodes are there in an M -ary tree with N internal nodes? Use your answer to give the amount of memory required to represent such a tree, assuming that links and items require one word of memory each.
- 5.69 Give upper and lower bounds on the height of an M -ary tree with N internal nodes.
- 5.70 Give upper and lower bounds on the internal path length of an M -ary tree with N internal nodes.
- 5.71 Give upper and lower bounds on the number of leaves in a binary tree with N nodes.
- 5.72 Show that if the levels of the external nodes in a binary tree differ by a constant, then the height is $O(\log N)$.
- 5.73 A *Fibonacci tree* of height $n > 2$ is a binary tree with a Fibonacci tree of height $n - 1$ in one subtree and a Fibonacci tree of height $n - 2$ in the other subtree. A Fibonacci tree of height 0 is a single external node, and a Fibonacci tree of height 1 is a single internal node with two external children (see Figure 5.14). Give the height and external path length of a Fibonacci tree of height n , as a function of N , the number of nodes in the tree.
- 5.74 A *divide-and-conquer tree* of N nodes is a binary tree with a root labeled N , a divide-and-conquer tree of $\lfloor N/2 \rfloor$ nodes in one subtree, and a divide-and-conquer tree of $\lfloor N/2 \rfloor$ nodes in the other subtree. (Figure 5.6 depicts a divide-and-conquer tree.) Draw divide-and-conquer trees with 11, 15, 16, and 23 nodes.
- 5.75 Prove by induction that the internal path length of a divide-and-conquer tree is between $N \lg N$ and $N \lg N + N$.
- 5.76 A *combine-and-conquer tree* of N nodes is a binary tree with a root labeled N , a combine-and-conquer tree of $\lceil N/2 \rceil$ nodes in one subtree, and a combine-and-conquer tree of $\lceil N/2 \rceil$ nodes in the other subtree (see Exercise 5.18). Draw combine-and-conquer trees with 11, 15, 16, and 23 nodes.
- 5.77 Prove by induction that the internal path length of a combine-and-conquer tree is between $N \lg N$ and $N \lg N + N$.
- 5.78 A *complete binary tree* is one with all levels filled, except possibly the final one, which is filled from left to right, as illustrated in Figure 5.24. Prove that the internal path length of a complete tree with N nodes is between $N \lg N$ and $N \lg N + N$.

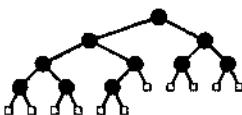
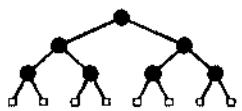


Figure 5.24
Complete binary trees with
seven and 10 internal nodes

When the number of external nodes is a power of 2 (top), the external nodes in a complete binary tree are all at the same level. Otherwise (bottom), the external nodes appear on two levels, with the internal nodes to the left of the external nodes on the next-to-bottom level.

5.6 Tree Traversal

Before considering algorithms that construct binary trees and trees, we consider algorithms for the most basic tree-processing function: *tree traversal*: Given a pointer to a tree, we want to process every node in the tree systematically. In a linked list, we move from one node to the next by following the single link; for trees, however, we have decisions to make, because there may be multiple links to follow.

We begin by considering the process for binary trees. For linked lists, we had two basic options (see Program 5.5): process the node and then follow the link (in which case we would visit the nodes in order), or follow the link and then process the node (in which case we would visit the nodes in reverse order). For binary trees, we have two links, and we therefore have three basic orders in which we might visit the nodes:

- *Preorder*, where we visit the node, then visit the left and right subtrees
- *Inorder*, where we visit the left subtree, then visit the node, then visit the right subtree
- *Postorder*, where we visit the left and right subtrees, then visit the node

We can implement these methods easily with a recursive program, as shown in Program 5.14, which is a direct generalization of the linked-list-traversal program in Program 5.5. To implement traversals in the other orders, we permute the function calls in Program 5.14 in the appropriate manner. Figure 5.26 shows the order in which we visit the nodes in a sample tree for each order. Figure 5.25 shows the sequence of function calls that is executed when we invoke Program 5.14 on the sample tree in Figure 5.26.

We have already encountered the same basic recursive processes on which the different tree-traversal methods are based, in divide-and-conquer recursive programs (see Figures 5.8 and 5.11), and in arithmetic expressions. For example, doing preorder traversal corresponds to drawing the marks on the ruler first, then making the recursive calls (see Figure 5.11); doing inorder traversal corresponds to moving the biggest disk in the towers of Hanoi solution in between recursive calls that move all of the others; doing postorder traversal corresponds to evaluating postfix expressions, and so forth. These correspondences

Program 5.14 Recursive tree traversal

This recursive function takes a link to a tree as an argument and calls the function `visit` with each of the nodes in the tree as argument. As is, the function implements a preorder traversal; if we move the call to `visit` between the recursive calls, we have an inorder traversal; and if we move the call to `visit` after the recursive calls, we have a postorder traversal.

```
void traverse(link h, void (*visit)(link))
{
    if (h == NULL) return;
    (*visit)(h);
    traverse(h->l, visit);
    traverse(h->r, visit);
}
```

give us immediate insight into the mechanisms behind tree traversal. For example, we know that every other node in an inorder traversal is an external node, for the same reason that every other move in the towers of Hanoi problem involves the small disk.

It is also useful to consider nonrecursive implementations that use an explicit pushdown stack. For simplicity, we begin by considering an abstract stack that can hold items or trees, initialized with the tree to be traversed. Then, we enter into a loop, where we pop and process the top entry on the stack, continuing until the stack is empty. If the popped entity is an item, we visit it; if the popped entity is a tree, then we perform a sequence of push operations that depends on the desired ordering:

- For *preorder*, we push the right subtree, then the left subtree, and then the node.
- For *inorder*, we push the right subtree, then the node, and then the left subtree.
- For *postorder*, we push the node, then the right subtree, and then the left subtree.

We do not push null trees onto the stack. Figure 5.27 shows the stack contents as we use each of these three methods to traverse the sample tree in Figure 5.26. We can easily verify by induction that this method produces the same output as the recursive one for any binary tree.

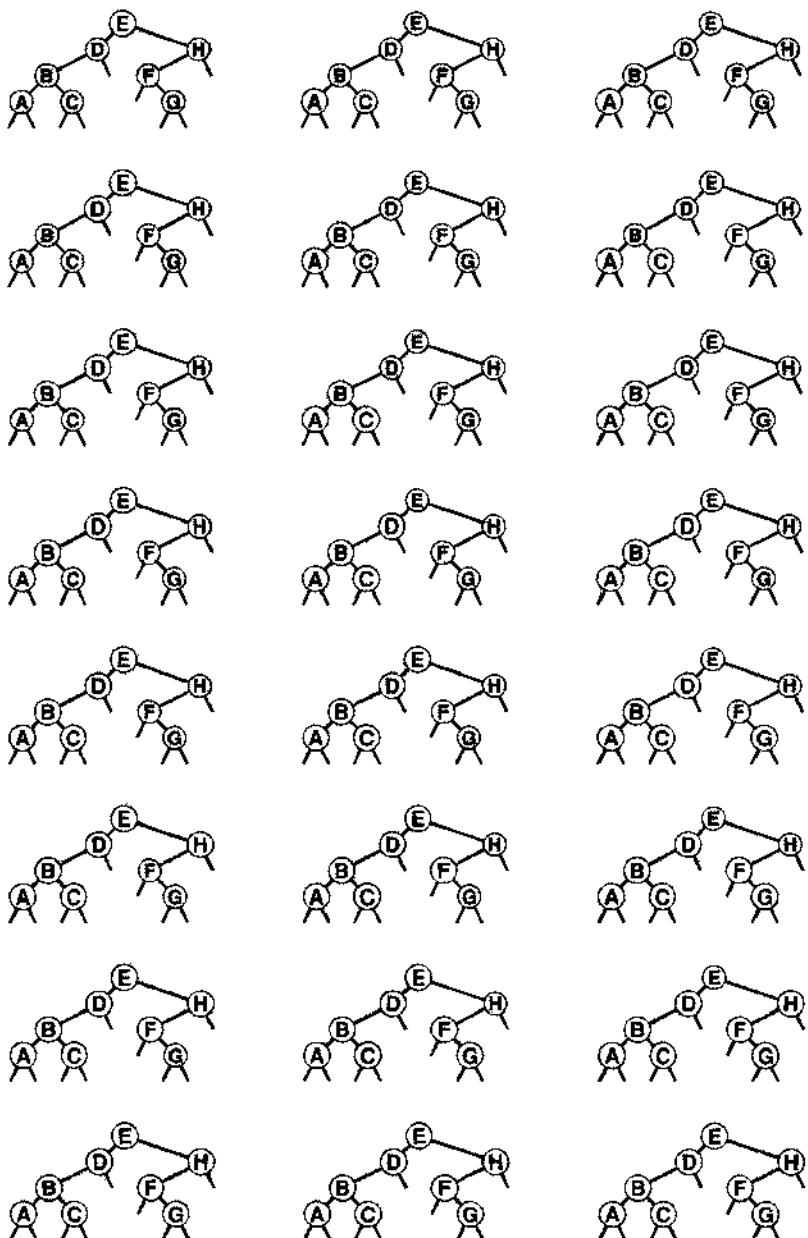
```
traverse E
visit E
traverse D
visit D
traverse B
visit B
traverse A
visit A
traverse *
traverse *
traverse C
visit C
traverse *
traverse *
traverse *
traverse *
traverse H
visit H
traverse F
visit F
traverse *
traverse G
visit G
traverse *
traverse *
traverse *
```

Figure 5.25
Preorder-traversal function calls

This sequence of function calls constitutes preorder traversal for the example tree in Figure 5.26.

Figure 5.26
Tree-traversal orders

These sequences indicate the order in which we visit nodes for pre-order (left), inorder (center), and postorder (right) tree traversal.



Program 5.15 Preorder traversal (nonrecursive)

This nonrecursive stack-based function is functionally equivalent to its recursive counterpart, Program 5.14.

```
void traverse(link h, void (*visit)(link))
{
    STACKinit(max); STACKpush(h);
    while (!STACKempty())
    {
        (*visit)(h = STACKpop());
        if (h->r != NULL) STACKpush(h->r);
        if (h->l != NULL) STACKpush(h->l);
    }
}
```

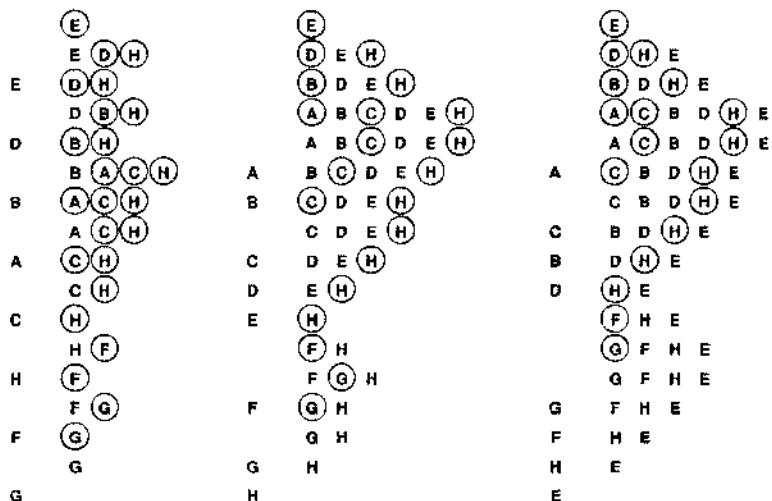
The scheme described in the previous paragraph is a conceptual one that encompasses the three traversal methods, but the implementations that we use in practice are slightly simpler. For example, for preorder, we do not need to push nodes onto the stack (we visit the root of each tree that we pop), and we therefore can use a simple stack that contains only one type of item (tree link), as in the nonrecursive implementation in Program 5.15. The system stack that supports the recursive program contains return addresses and argument values, rather than items or nodes, but the actual sequence in which we do the computations (visit the nodes) is the same for the recursive and the stack-based methods.

A fourth natural traversal strategy is simply to visit the nodes in a tree as they appear on the page, reading down from top to bottom and from left to right. This method is called *level-order* traversal because all the nodes on each level appear together, in order. Figure 5.28 shows how the nodes of the tree in Figure 5.26 are visited in level order.

Remarkably, we can achieve level-order traversal by substituting a queue for the stack in Program 5.15, as shown in Program 5.16. For preorder, we use a LIFO data structure; for level order, we use a FIFO data structure. These programs merit careful study, because they represent approaches to organizing work remaining to be done that differ in an essential way. In particular, level order does *not* correspond

Figure 5.27
Stack contents for tree-traversal algorithms

These sequences indicate the stack contents for preorder (left), inorder (center), and postorder (right) tree traversal (see Figure 5.26), for an idealized model of the computation, similar to the one that we used in Figure 5.5, where we put the item and its two subtrees on the stack, in the indicated order.

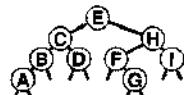
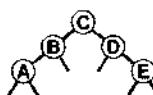
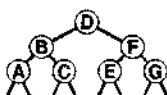


to a recursive implementation relates to the recursive structure of the tree.

Preorder, postorder, and level order are well defined for forests as well. To make the definitions consistent, think of a forest as a tree with an imaginary root. Then, the preorder rule is “visit the root, then visit each of the subtrees,” the postorder rule is “visit each of the subtrees, then visit the root.” The level-order rule is the same as for binary trees. Direct implementations of these methods are straightforward generalizations of the stack-based preorder traversal programs (Programs 5.14 and 5.15) and the queue-based level-order traversal program (Program 5.16) for binary trees that we just considered. We omit consideration of implementations because we consider a more general procedure in Section 5.8.

Exercises

▷ 5.79 Give preorder, inorder, postorder, and level-order traversals of the following binary trees:



Program 5.16 Level-order traversal

Switching the underlying data structure in preorder traversal (see Program 5.15) from a stack to a queue transforms the traversal into a level-order one.

```
void traverse(link h, void (*visit)(link))
{
    QUEUEinit(max); QUEUEput(h);
    while (!QUEUEempty())
    {
        (*visit)(h = QUEUEget());
        if (h->l != NULL) QUEUEput(h->l);
        if (h->r != NULL) QUEUEput(h->r);
    }
}
```

▷ 5.80 Show the contents of the queue during the level order traversal (Program 5.16) depicted in Figure 5.28, in the style of Figure 5.27.

5.81 Show that preorder for a forest is the same as preorder for the corresponding binary tree (see Property 5.4), and that postorder for a forest is the same as inorder for the binary tree.

○ 5.82 Give a nonrecursive implementation of inorder traversal.

● 5.83 Give a nonrecursive implementation of postorder traversal.

● 5.84 Write a program that takes as input the preorder and inorder traversals of a binary tree, and produces as output the level-order traversal of the tree.

5.7 Recursive Binary-Tree Algorithms

The tree-traversal algorithms that we considered in Section 5.6 exemplify the basic fact that we are led to consider recursive algorithms for binary trees, because of these trees' very nature as recursive structures. Many tasks admit direct recursive divide-and-conquer algorithms, which essentially generalize the traversal algorithms. We process a tree by processing the root node and (recursively) its subtrees; we can do computation before, between, or after the recursive calls (or possibly all three).

We frequently need to find the values of various structural parameters for a tree, given only a link to the tree. For example, Program 5.17

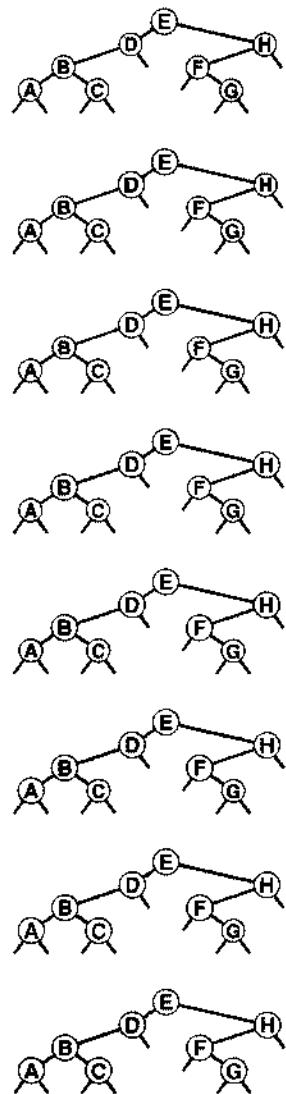


Figure 5.28
Level-order traversal

This sequence depicts the result of visiting nodes in order from top to bottom and left to right in the tree.

Program 5.17 Computation of tree parameters

We can use simple recursive procedures such as these to learn basic structural properties of trees.

```
int count(link h)
{
    if (h == NULL) return 0;
    return count(h->l) + count(h->r) + 1;
}
int height(link h)
{ int u, v;
    if (h == NULL) return -1;
    u = height(h->l); v = height(h->r);
    if (u > v) return u+1; else return v+1;
}
```

comprises recursive functions for computing the number of nodes in and the height of a given tree. The functions follow immediately from Definition 5.6. Neither of these functions depends on the order in which the recursive calls are processed: they process all the nodes in the tree and return the same answer if we, for example, exchange the recursive calls. Not all tree parameters are so easily computed: for example, a program to compute efficiently the internal path length of a binary tree is more challenging (see Exercises 5.88 through 5.90).

Another function that is useful whenever we write programs that process trees is one that prints out or draws the tree. For example, Program 5.18 is a recursive procedure that prints out a tree in the format illustrated in Figure 5.29. We can use the same basic recursive scheme to draw more elaborate representations of trees, such as those that we use in the figures in this book (see Exercise 5.85).

Program 5.18 is an inorder traversal—if we print the item before the recursive calls, we get a preorder traversal, which is also illustrated in Figure 5.29. This format is a familiar one that we might use, for example, for a family tree, or to list files in a tree-based file system, or to make an outline of a printed document. For example, doing a preorder traversal of the tree in Figure 5.19 gives a version of the table of contents of this book.

Program 5.18 Quick tree-print function

This recursive program keeps track of the tree height and uses that information for indentation in printing out a representation of the tree that we can use to debug tree-processing programs (see Figure 5.29). It assumes that items in nodes are characters.

```
void printnode(char c, int h)
{
    int i;
    for (i = 0; i < h; i++) printf(" ");
    printf("%c\n", c);
}
void show(link x, int h)
{
    if (x == NULL) { printnode('*', h); return; }
    show(x->r, h+1);
    printnode(x->item, h);
    show(x->l, h+1);
}
```

Our first example of a program that builds an explicit binary tree structure is associated with the find-the-maximum application that we considered in Section 5.2. Our goal is to build a *tournament*: a binary tree where the item in every internal node is a copy of the larger of the items in its two children. In particular, the item at the root is a copy of the largest item in the tournament. The items in the leaves (nodes with no children) constitute the data of interest, and the rest of the tree is a data structure that allows us to find the largest of the items efficiently.

Program 5.19 is a recursive program that builds a tournament from the items in an array. A modification of Program 5.6, it thus uses a divide-and-conquer recursive strategy: To build a tournament for a single item, we create (and return) a leaf containing that item. To build a tournament for $N > 1$ items, we use the divide-and-conquer strategy: Divide the items in half, build tournaments for each half, and create a new node with links to the two tournaments and with an item that is a copy of the larger of the items in the roots of the two tournaments.

Figure 5.30 is an example of an explicit tree structure that might be built by Program 5.19. Building a recursive data structure such

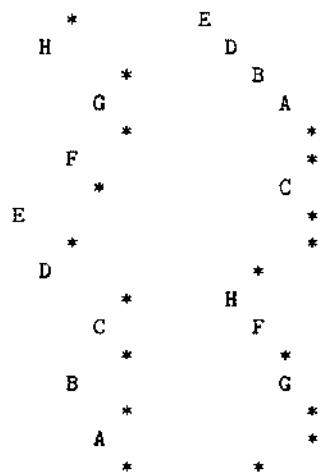


Figure 5.29
Printing a tree (inorder and preorder)

The output at the left results from using Program 5.18 on the sample tree in Figure 5.26, and exhibits the tree structure in a manner similar to the graphical representation that we have been using, rotated 90 degrees. The output at the right is from the same program with the print statement moved to the beginning; it exhibits the tree structure in a familiar outline format.

Program 5.19 Construction of a tournament

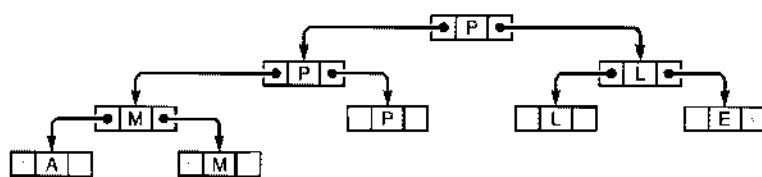
This recursive function divides a file $a[1], \dots, a[r]$ into the two parts $a[1], \dots, a[m]$ and $a[m+1], \dots, a[r]$, builds tournaments for the two parts (recursively), and makes a tournament for the whole file by setting links in a new node to the recursively built tournaments and setting its item value to the larger of the items in the roots of the two recursively built tournaments.

```

typedef struct node *link;
struct node { Item item; link l, r };
link NEW(Item item, link l, link r)
{ link x = malloc(sizeof *x);
  x->item = item; x->l = l; x->r = r;
  return x;
}
link max(Item a[], int l, int r)
{ int m = (l+r)/2; Item u, v;
  link x = NEW(a[m], NULL, NULL);
  if (l == r) return x;
  x->l = max(a, l, m);
  x->r = max(a, m+1, r);
  u = x->l->item; v = x->r->item;
  if (u > v)
    x->item = u; else x->item = v;
  return x;
}

```

as this one is perhaps preferable in some situations to finding the maximum by scanning the data, as we did in Program 5.6, because the tree structure provides us with the flexibility to perform other operations. The very operation that we use to build the tournament is an important example: Given two tournaments, we can combine them into a single tournament in constant time, by creating a new node, making its left link point to one of the tournaments and its right link point to the other, and taking the larger of the two items (at the roots of the two given tournaments) as the largest item in the combined tournament. We also can consider algorithms for adding items, removing items, and performing other operations. We shall not



consider such operations in any further detail here because similar data structures with this flexibility are the topic of Chapter 9.

Indeed, tree-based implementations for several of the generalized queue ADTs that we discussed in Section 4.6 are a primary topic of discussion for much of this book. In particular, many of the algorithms in Chapters 12 through 15 are based on *binary search trees*, which are explicit trees that correspond to binary search, in a relationship analogous to the relationship between the explicit structure of Figure 5.30 and the recursive find-the-maximum algorithm (see Figure 5.6). The challenge in implementing and using such structures is to ensure that our algorithms remain efficient after a long sequence of *insert*, *delete*, and other operations.

Our second example of a program that builds a binary tree is a modification of our prefix-expression-evaluation program in Section 5.1 (Program 5.4) to construct a tree representing a prefix expression, instead of just evaluating it (see Figure 5.31). Program 5.20 uses the same recursive scheme as Program 5.4, but the recursive function returns a link to a tree, rather than a value. We create a new tree node for each character in the expression: Nodes corresponding to operators have links to their operands, and the leaf nodes contain the variables (or constants) that are inputs to the expression.

Translation programs such as compilers often use such internal tree representations for programs, because the trees are useful for many purposes. For example, we might imagine operands corresponding to variables that take on values, and we could generate machine code to evaluate the expression represented by the tree with a postorder traversal. Or, we could use the tree to print out the expression in infix with an inorder traversal or in postfix with a postorder traversal.

We considered the few examples in this section to introduce the concept that we can build and process explicit linked tree structures with recursive programs. To do so effectively, we need to consider

Figure 5.30
Explicit tree for finding the maximum (tournament)

This figure depicts the explicit tree structure that is constructed by Program 5.19 from the input A M P L E. The data items are in the leaves. Each internal node has a copy of the larger of the items in its two children, so, by induction, the largest item is at the root.

Program 5.20 Construction of a parse tree

Using the same strategy that we used to evaluate prefix expressions (see Program 5.4), this program builds a parse tree from a prefix expression. For simplicity, we assume that operands are single characters. Each call of the recursive function creates a new node with the next character from the input as the token. If the token is an operand, we return the new node; if it is an operator, we set the left and right pointers to the tree built (recursively) for the two arguments.

```
char *a; int i;
typedef struct Tnode* link;
struct Tnode { char token; link l, r; };
link NEW(char token, link l, link r)
{ link x = malloc(sizeof *x);
  x->token = token; x->l = l; x->r = r;
  return x;
}
link parse()
{ char t = a[i++];
  link x = NEW(t, NULL, NULL);
  if ((t == '+') || (t == '*'))
    { x->l = parse(); x->r = parse(); }
  return x;
}
```

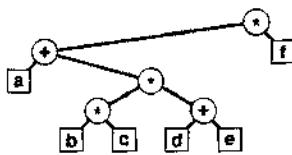


Figure 5.31
Parse tree

This tree is constructed by Program 5.20 for the prefix expression $* + a * b c + d e f$. It is a natural way to represent the expression: Each operand is in a leaf (which we draw here as an external node), and each operator is to be applied to the expressions represented by the left and right subtrees of the node containing the operator.

the performance of various algorithms, alternate representations, non-recursive alternatives, and many other details. However, we shall defer consideration of tree-processing programs in further detail until Chapter 12, because we use trees primarily for descriptive purposes in Chapters 7 through 11. We return to explicit tree implementations in Chapter 12 because they form the basis of numerous algorithms that we consider in Chapters 12 through 15.

Exercises

- o 5.85 Modify Program 5.18 to output a PostScript program that draws the tree, in a format like that used in Figure 5.23, but without the small boxes to

represent the external nodes. Use `moveto` and `lineto` to draw lines, and the user-defined operator

```
/node { newpath moveto currentpoint 4 0 360 arc fill} def
```

to draw nodes. After this definition, the call `node` draws a black dot at the coordinates on the stack (see Section 4.3).

▷ 5.86 Write a program that counts the leaves in a binary tree.

▷ 5.87 Write a program that counts the number of nodes in a binary tree that have one external and one internal child.

▷ 5.88 Write a recursive program that computes the internal path length of a binary tree, using Definition 5.6.

5.89 Determine the number of function calls made by your program when it is computing the internal path length of a binary tree. Prove your answer by induction.

• 5.90 Write a recursive program that computes the internal path length of a binary tree in time proportional to the number of nodes in the tree.

○ 5.91 Write a recursive program that deletes all the leaves with a given key from a tournament (see Exercise 5.59).

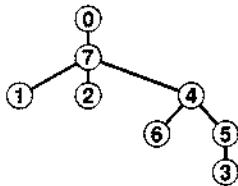
5.8 Graph Traversal

For our final example of a recursive program in this chapter, we consider one of the most important of all recursive programs: recursive graph traversal, or *depth-first search*. This method for systematically visiting all the nodes in a graph is a direct generalization of the tree-traversal methods that we considered in Section 5.6, and it serves as the basis for many basic algorithms for processing graphs (see Part 7). It is a simple recursive algorithm. Starting at any node v , we

- Visit v .
- (Recursively) visit each (*unvisited*) node attached to v .

If the graph is connected, we eventually reach all of the nodes. Program 5.21 is an implementation of this recursive procedure.

For example, suppose that we use the adjacency-list representation depicted in the sample graph in Figure 3.15. Figure 5.32 shows the recursive calls made during the depth-first search of this graph, and the sequence on the left in Figure 5.33 depicts the way in which we follow the edges in the graph. We follow each edge in the graph, with one of two possible outcomes: if the edge takes us to a node that we have already visited, we ignore it; if it takes us to a node that we have



visit 0
 visit 7 (first on 0's list)
 visit 1 (first on 7's list)
 check 7 on 1's list
 check 0 on 1's list
 visit 2 (second on 7's list)
 check 7 on 2's list
 check 0 on 2's list
 check 0 on 7's list
 visit 4 (fourth on 7's list)
 visit 6 (first on 4's list)
 check 4 on 6's list
 check 0 on 6's list
 visit 5 (second on 4's list)
 check 0 on 5's list
 check 4 on 5's list
 visit 3 (third on 5's list)
 check 5 on 3's list
 check 4 on 3's list
 check 7 on 4's list
 check 3 on 4's list
 check 5 on 0's list
 check 2 on 0's list
 check 1 on 0's list
 check 6 on 0's list

Figure 5.32
Depth-first-search function calls

This sequence of function calls constitutes depth-first search for the example graph in Figure 3.15. The tree that depicts the recursive-call structure (top) is called the depth-first-search tree.

Program 5.21 Depth-first search

To visit all the nodes connected to node k in a graph, we mark it as *visited*, then (recursively) visit all the unvisited nodes on k 's adjacency list.

```

void traverse(int k, void (*visit)(int))
{
    link t;
    (*visit)(k); visited[k] = 1;
    for (t = adj[k]; t != NULL; t = t->next)
        if (!visited[t->v]) traverse(t->v, visit);
}
  
```

not yet visited, we follow it there via a recursive call. The set of all edges that we follow in this way forms a spanning tree for the graph.

The difference between depth-first search and general tree traversal (see Program 5.14) is that we need to guard explicitly against visiting nodes that we have already visited. In a tree, we never encounter any such nodes. Indeed, if the graph is a tree, recursive depth-first search starting at the root is equivalent to preorder traversal.

Property 5.10 *Depth-first search requires time proportional to $V + E$ in a graph with V vertices and E edges, using the adjacency lists representation.*

In the adjacency lists representation, there is one list node corresponding to each edge in the graph, and one list head pointer corresponding to each vertex in the graph. Depth-first search touches all of them, at most once. ■

Because it also takes time proportional to $V + E$ to build the adjacency lists representation from an input sequence of edges (see Program 3.19), depth-first search gives us a linear-time solution to the connectivity problem of Chapter 1. For huge graphs, however, the union–find solutions might still be preferable, because representing the whole graph takes space proportional to E , while the union–find solutions take space only proportional to V .

As we did with tree traversal, we can define a graph-traversal method that uses an explicit stack, as depicted in Figure 5.34. We can think of an abstract stack that holds dual entries: a node and a pointer into that node's adjacency list. With the stack initialized

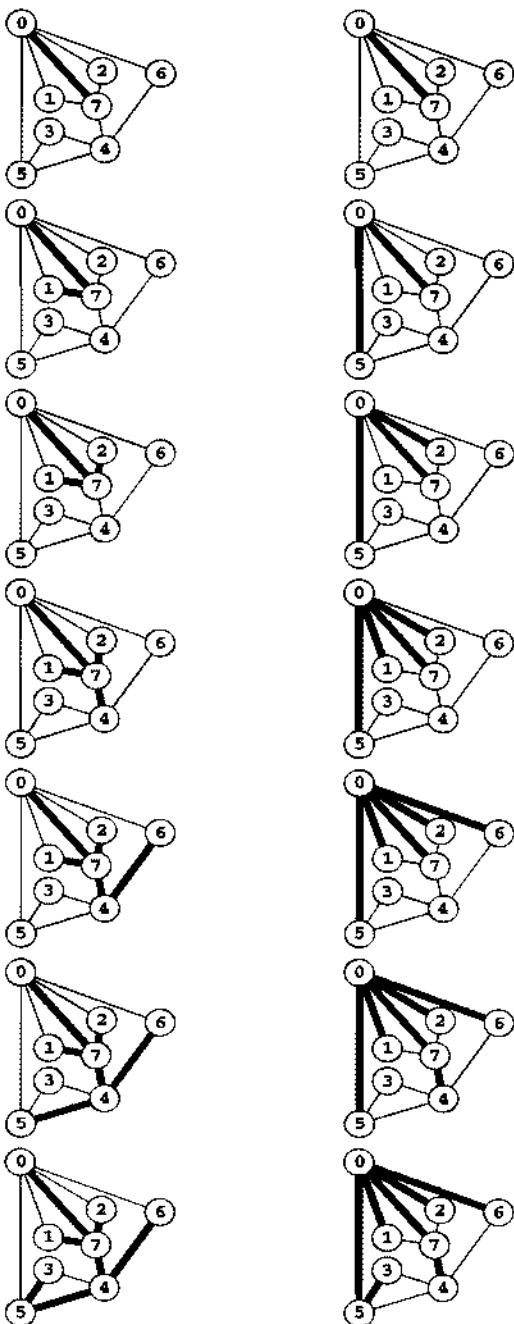


Figure 5.33
Depth-first search and
breadth-first search

Depth-first search (left) moves from node to node, backing up to the previous node to try the next possibility whenever it has tried every possibility at a given node. Breadth-first search (right) exhausts all the possibilities at one node before moving to the next.

Figure 5.34
Depth-first-search stack dynamics

We can think of the pushdown stack supporting depth-first search as containing a node and a reference to that node's adjacency list (indicated by a circled node) (left). Thus, we begin with node 0 on the stack, with reference to the first node on its list, node 7. Each line indicates the result of popping the stack, pushing a reference to the next node on the list for nodes that have been visited, and pushing an entry on the stack for nodes that have not been visited. Alternatively, we can think of the process as simply pushing all nodes adjacent to any unvisited node onto the stack (right).

0	0	(7)	0	7	5	2	1	6
7	7	(1)	0	5	7	1	2	0
1	1	(7)	7	2	0	5	1	2
		1	0	7	2	0	5	
		7	2	0	5			
2	2	(7)	7	0	0	5	2	
		2	0	7	0	0	5	
		7	0	0	5			
		7	4	0	5			
4	4	6	0	5	4	6	5	7
6	6	4	4	5	0	5	6	4
		6	0	4	5	0	5	0
		4	5	0	5		5	7
5	5	0	4	7	0	5	5	0
		5	4	4	7	0	5	4
		5	3	4	7	0	5	3
3	3	5	4	7	0	5	3	5
		3	4	4	7	0	5	4
		4	7	0	5		7	3
		4	3	0	5		3	5
0	0	5				5	1	2
		0	2			1	2	6
		0	1			2	6	
		0	6			6		

to the start node and a pointer initialized to the first node on that node's adjacency list, the depth-first search algorithm is equivalent to entering into a loop, where we visit the node at the top of the stack (if it has not already been visited); save the node referenced by the current adjacency-list pointer; update the adjacency list reference to the next node (popping the entry if at the end of the adjacency list); and push a stack entry for the saved node, referencing the first node on its adjacency list.

Alternatively, as we did for tree traversal, we can consider the stack to contain links to nodes only. With the stack initialized to the start node, we enter into a loop where we visit the node at the top of the stack (if it has not already been visited), then push all the nodes adjacent to it onto the stack. Figure 5.34 illustrates that both of these methods are equivalent to depth-first search for our example graph, and the equivalence indeed holds in general.

Program 5.22 Breadth-first search

To visit all the nodes connected to node k in a graph, we put k onto a FIFO queue, then enter into a loop where we get the next node from the queue, and, if it has not been visited, visit it and push all the unvisited nodes on its adjacency list, continuing until the queue is empty.

```
void traverse(int k, void (*visit)(int))
{
    link t;
    QUEUEinit(V); QUEUEput(k);
    while (!QUEUEempty())
        if (visited[k = QUEUEget()] == 0)
        {
            (*visit)(k); visited[k] = 1;
            for (t = adj[k]; t != NULL; t = t->next)
                if (visited[t->v] == 0) QUEUEput(t->v);
        }
}
```

The visit-the-top-node-and-push-all-its-neighbors algorithm is a simple formulation of depth-first search, but it is clear from Figure 5.34 that it suffers the disadvantage of possibly leaving multiple copies of each node on the stack. It does so even if we test whether each node that is about to go on the stack has been visited and refrain from putting the node in the stack if it has been. To avoid this problem, we can use a stack implementation that disallows duplicates by using a forget-the-old-item policy, because the copy nearest the top of the stack is always the first one visited, so the others are simply popped.

The stack dynamics for depth-first search that are illustrated in Figure 5.34 depend on the nodes on each adjacency list ending up on the stack in the same order that they appear in the list. To get this ordering for a given adjacency list when pushing one node at a time, we would have to push the last node first, then the next-to-last node, and so forth. Moreover, to limit the stack size to the number of vertices while at the same time visiting the nodes in the same order as in depth-first search, we need to use a stack discipline with a forget-the-old-item policy. If visiting the nodes in the same order as depth-first search is not important to us, we can avoid both of these complications and directly formulate a nonrecursive stack-based graph-traversal method: With

Figure 5.35
Breadth-first-search queue dynamics

We start with 0 on the queue, then get 0, visit it, and put the nodes on its adjacency list 7 5 2 1 6, in that order onto the queue. Then we get 7, visit it, and put the nodes on its adjacency list, and so forth. With duplicates disallowed with an ignore-the-new-item policy (right), we get the same result without any extraneous queue entries.

0	0
0 7 5 2 1 6	0 7 5 2 1 6
7 5 2 1 6 1 2 4	7 5 2 1 6 4
5 2 1 6 1 2 4 4 3	5 2 1 6 4 3
2 1 6 1 2 4 4 3	2 1 6 4 3
1 6 1 2 4 4 3	1 6 4 3
6 1 2 4 4 3 4	6 4 3
2 4 4 3 4	4 3
4 4 3 4	3
3 4 3	
3	

the stack initialized to the start node, we enter into a loop where we visit the node at the top of the stack, then proceed through its adjacency list, pushing each node onto the stack (if the node has not been visited already), using a stack implementation that disallows duplicates with an ignore-the-new-item policy. This algorithm visits all the nodes in the graph in a manner similar to depth-first-search, but it is not recursive.

The algorithm in the previous paragraph is noteworthy because we could use any generalized queue ADT, and still visit each of the nodes in the graph (and generate a spanning tree). For example, if we use a queue instead of a stack, then we have *breadth-first search*, which is analogous to level-order traversal in a tree. Program 5.22 is an implementation of this method (assuming that we use a queue implementation like Program 4.12); an example of the algorithm in operation is depicted in Figure 5.35. In Part 6, we shall examine numerous graph algorithms based on more sophisticated generalized queue ADTs.

Breadth-first search and depth-first search both visit all the nodes in a graph, but their manner of doing so is dramatically different, as illustrated in Figure 5.36. Breadth-first search amounts to an army of searchers fanning out to cover the territory; depth-first search corresponds to a single searcher probing unknown territory as deeply as possible, retreating only when hitting dead ends. These are basic problem-solving paradigms of significance in many areas of computer science beyond graph searching.

Exercises

- 5.92 Show how recursive depth-first search visits the nodes in the graph built for the edge sequence 0-2, 1-4, 2-5, 3-6, 0-4, 6-0, and 1-3 (see Exercise 3.70), by giving diagrams corresponding to Figures 5.33 (left) and 5.34 (right).
- 5.93 Show how stack-based depth-first search visits the nodes in the graph built for the edge sequence 0-2, 1-4, 2-5, 3-6, 0-4, 6-0, and 1-3, by giving diagrams corresponding to Figures 5.33 (left) and 5.34 (right).
- 5.94 Show how (queue-based) breadth-first search visits the nodes in the graph built for the edge sequence 0-2, 1-4, 2-5, 3-6, 0-4, 6-0, and 1-3, by giving diagrams corresponding to Figures 5.33 (right) and 5.35 (left).
- 5.95 Why is the running time in Property 5.10 quoted as $V + E$ and not simply E ?
 - 5.96 Show how stack-based depth-first search visits the nodes in the example graph in the text (Figure 3.15) when using a forget-the-old-item policy, by giving diagrams corresponding to Figures 5.33 (left) and 5.35 (right).
 - 5.97 Show how stack-based depth-first search visits the nodes in the example graph in the text (Figure 3.15) when using an ignore-the-new-item policy, by giving diagrams corresponding to Figures 5.33 (left) and 5.35 (right).
- ▷ 5.98 Implement a stack-based depth-first search for graphs that are represented with adjacency lists.
- 5.99 Implement a recursive depth-first search for graphs that are represented with adjacency lists.

5.9 Perspective

Recursion lies at the heart of early theoretical studies into the nature of computation. Recursive functions and programs play a central role in mathematical studies that attempt to separate problems that can be solved by a computer from problems that cannot be.

It is certainly impossible to do justice to topics as far-reaching as trees and recursion in so brief a discussion. Many of the best examples of recursive programs will be our focus throughout the book—divide-and-conquer algorithms and recursive data structures that have been applied successfully to solve a wide variety of problems. For many applications, there is no reason to go beyond a simple, direct recursive implementation; for others, we will consider the derivation of alternate nonrecursive and bottom-up implementations.

In this book, our interest lies in the practical aspects of recursive programs and data structures. Our goal is to exploit recursion to

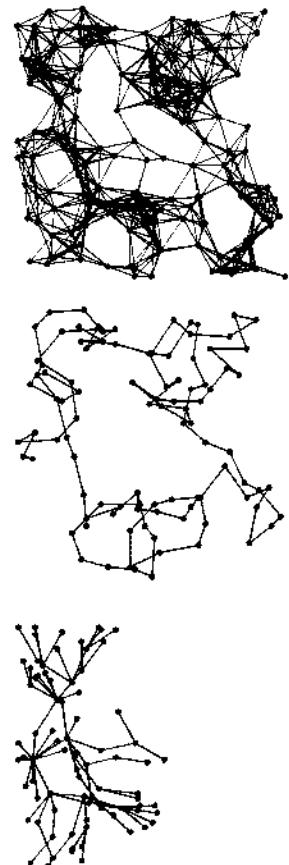


Figure 5.36
Graph-traversal trees

This diagram shows depth-first search (center) and breadth-first search (bottom), halfway through searching in a large graph (top). Depth-first search meanders from one node to the next, so most nodes are connected to just two others. By contrast, breadth-first search sweeps through the graph, visiting all the nodes connected to a given node before moving on, so several nodes are connected to many others.

produce elegant and efficient implementations. To meet that goal, we need to have particular respect for the dangers of simple programs that lead to an exponential number of function calls or impossibly deep nesting. Despite this pitfall, recursive programs and data structures are attractive because they often provide us with inductive arguments that can convince us that our programs are correct and efficient.

We use trees throughout the book, both to help us understand the dynamic properties of programs, and as dynamic data structures. Chapters 12 through 15 in particular are largely devoted to the manipulation of explicit tree structures. The properties described in this chapter provide us with the basic information that we need if we are to use explicit tree structures effectively.

Despite its central role in algorithm design, recursion is not a panacea. As we discovered in our study of tree- and graph-traversal algorithms, stack-based (inherently recursive) algorithms are not the only option when we have multiple computational tasks to manage. An effective algorithm-design technique for many problems is the use of generalized queue implementations other than stacks to give us the freedom to choose the next task according to some more subjective criteria than simply choosing the most recent. Data structures and algorithms that efficiently support such operations are a prime topic of Chapter 9, and we shall encounter many examples of their application when we consider graph algorithms in Part 7.

References for Part Two

There are numerous introductory textbooks on data structures. For example, the book by Standish covers linked structures, data abstraction, stacks and queues, memory allocation, and software engineering concepts at a more leisurely pace than here. Summit's book (and its source on the web) is an invaluable source of detailed information about C implementations, as is, of course, the Kernighan and Ritchie classic. The book by Plauger is a thorough explanation of C library functions.

The designers of PostScript perhaps did not anticipate that their language would be of interest to people learning basic algorithms and data structures. However, the language is not difficult to learn, and the reference manual is both thorough and accessible.

The technique for implementing ADTs with pointers to structures that are not specified was taught by Appel in the systems programming course at Princeton in the mid 1980s. It is described in full detail, with numerous examples, in the book by Hanson. The Hanson and Summit books are both outstanding references for programmers who want to write bugfree and portable code for large systems.

Knuth's books, particularly Volumes 1 and 3, remain the authoritative source on properties of elementary data structures. Baeza-Yates and Gonnet have more up-to-date information, backed by an extensive bibliography. Sedgewick and Flajolet cover mathematical properties of trees in detail.

- Adobe Systems Incorporated, *PostScript Language Reference Manual, second edition*, Addison-Wesley, Reading, MA, 1990.
- R. Baeza-Yates and G. H. Gonnet, *Handbook of Algorithms and Data Structures*, second edition, Addison-Wesley, Reading, MA, 1984.
- D. R. Hanson, *C Interfaces and Implementations: Techniques for Creating Reusable Software*, Addison-Wesley, 1997.
- B. W. Kernighan and D. M. Ritchie, *The C Programming Language, second edition*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, second edition, Addison-Wesley, Reading, MA, 1973; *Volume 2: Seminumerical Algorithms*, second edition, Addison-Wesley, Reading, MA, 1981; *Volume 3: Sorting*

- and Searching*, second printing, Addison-Wesley, Reading, MA, 1975.
- P. J. Plauger, *The Standard C Library*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1996.
- T. A. Standish, *Data Structures, Algorithms, and Software Principles in C*, Addison-Wesley, 1995.
- S. Summit, *C Programming FAQs*, Addison-Wesley, 1996.

P A R T
T H R E E

Sorting

CHAPTER SIX

Elementary Sorting Methods

FOR OUR FIRST excursion into the area of sorting algorithms, we shall study several elementary methods that are appropriate for small files, or for files that have a special structure. There are several reasons for studying these simple sorting algorithms in detail. First, they provide context in which we can learn terminology and basic mechanisms for sorting algorithms, and thus allow us to develop an adequate background for studying the more sophisticated algorithms. Second, these simple methods are actually more effective than the more powerful general-purpose methods in many applications of sorting. Third, several of the simple methods extend to better general-purpose methods or are useful in improving the efficiency of more sophisticated methods.

Our purpose in this chapter is not just to introduce the elementary methods, but also to develop a framework within which we can study sorting in later chapters. We shall look at a variety of situations that may be important in applying sorting algorithms, examine different kinds of input files, and look at other ways of comparing sorting methods and learning their properties.

We begin by looking at a simple driver program for testing sorting methods, which provides a context for us to consider the conventions that we shall follow. We also consider the basic properties of sorting methods that are important for us to know when we are evaluating the utility of algorithms for particular applications. Then, we look closely at implementations of three elementary methods: selection sort, insertion sort, and bubble sort. Following that, we examine

the performance characteristics of these algorithms in detail. Next, we look at shellsort, which is perhaps not properly characterized as elementary, but is easy to implement and is closely related to insertion sort. After a digression into the mathematical properties of shellsort, we delve into the subject of developing data type interfaces and implementations, along the lines that we have discussed in Chapters 3 and 4, for extending our algorithms to sort the kinds of data files that arise in practice. We then consider sorting methods that refer indirectly to the data and linked-list sorting. The chapter concludes with a discussion of a specialized method that is appropriate when the key values are known to be restricted to a small range.

In numerous sorting applications, a simple algorithm may be the method of choice. First, we often use a sorting program only once, or just a few times. Once we have “solved” a sort problem for a set of data, we may not need the sort program again in the application manipulating those data. If an elementary sort is no slower than some other part of processing the data—for example reading them in or printing them out—then there may be no point in looking for a faster way. If the number of items to be sorted is not too large (say, less than a few hundred elements), we might just choose to implement and run a simple method, rather than bothering with the interface to a system sort or with implementing and debugging a complicated method. Second, elementary methods are always suitable for small files (say, less than a few dozen elements)—sophisticated algorithms generally incur overhead that makes them slower than elementary ones for small files. This issue is not worth considering unless we wish to sort a huge number of small files, but applications with such a requirement are not unusual. Other types of files that are relatively easy to sort are ones that are already almost sorted (or already are sorted!) or ones that contain large numbers of duplicate keys. We shall see that several of the simple methods are particularly efficient when sorting such well-structured files.

As a rule, the elementary methods that we discuss here take time proportional to N^2 to sort N randomly arranged items. If N is small, this running time may be perfectly adequate. As just mentioned, the methods are likely to be even faster than more sophisticated methods for tiny files and in other special situations. But the methods that we discuss in this chapter are *not* suitable for large, randomly arranged

files, because the running time will become excessive even on the fastest computers. A notable exception is shellsort (see Section 6.6), which takes many fewer than N^2 steps for large N , and is arguably the sorting method of choice for midsize files and for a few other special applications.

6.1 Rules of the Game

Before considering specific algorithms, we will find it useful to discuss general terminology and basic assumptions for sorting algorithms. We shall be considering methods of sorting *files* of *items* containing *keys*. All these concepts are natural abstractions in modern programming environments. The keys, which are only part (often a small part) of the items, are used to control the sort. The objective of the sorting method is to rearrange the items such that their keys are ordered according to some well-defined ordering rule (usually numerical or alphabetical order). Specific characteristics of the keys and the items can vary widely across applications, but the abstract notion of putting keys and associated information into order is what characterizes the sorting problem.

If the file to be sorted will fit into memory, then the sorting method is called *internal*. Sorting files from tape or disk is called *external* sorting. The main difference between the two is that an internal sort can access any item easily whereas an external sort must access items sequentially, or at least in large blocks. We shall look at a few external sorts in Chapter 11, but most of the algorithms that we consider are internal sorts.

We shall consider both arrays and linked lists. The problem of sorting arrays and the problem of sorting linked lists are both of interest: during the development of our algorithms, we shall also encounter some basic tasks that are best suited for sequential allocation, and other tasks that are best suited for linked allocation. Some of the classical methods are sufficiently abstract that they can be implemented efficiently for either arrays or linked lists; others are particularly well suited to one or the other. Other types of access restrictions are also sometimes of interest.

We begin by focusing on array sorting. Program 6.1 illustrates many of the conventions that we shall use in our implementations. It

Program 6.1 Example of array sort with driver program

This program illustrates our conventions for implementing basic array sorts. The `main` function is a driver that initializes an array of integers (either with random values or from standard input), calls a `sort` function to sort that array, then prints out the ordered result.

The `sort` function, which is a version of insertion sort (see Section 6.3 for a detailed description, an example, and an improved implementation), assumes that the data type of the items being sorted is `Item`, and that the operations `less` (compare two keys), `exch` (exchange two items), and `compexch` (compare two items and exchange them if necessary to make the second not less than the first) are defined for `Item`. We implement `Item` for integers (as needed by `main`) with `typedef` and simple macros in this code. Use of other data types is the topic of Section 6.7, and does not affect `sort`.

```
#include <stdio.h>
#include <stdlib.h>
typedef int Item;
#define key(A) (A)
#define less(A, B) (key(A) < key(B))
#define exch(A, B) { Item t = A; A = B; B = t; }
#define compexch(A, B) if (less(B, A)) exch(A, B)
void sort(Item a[], int l, int r)
{ int i, j;
    for (i = l+1; i <= r; i++)
        for (j = i; j > l; j--)
            compexch(a[j-1], a[j]);
}
main(int argc, char *argv[])
{ int i, N = atoi(argv[1]), sw = atoi(argv[2]);
    int *a = malloc(N*sizeof(int));
    if (sw)
        for (i = 0; i < N; i++)
            a[i] = 1000*(1.0*rand()/RAND_MAX);
    else
        while (scanf("%d", &a[N]) == 1) N++;
    sort(a, 0, N-1);
    for (i = 0; i < N; i++) printf("%3d ", a[i]);
    printf("\n");
}
```

consists of a driver program that fills an array by reading integers from standard input or generating random ones (as dictated by an integer argument); then calls a sort function to put the integers in the array in order; then prints out the sorted result.

As we know from Chapters 3 and 4, there are numerous mechanisms available to us to arrange for our sort implementations to be useful for other types of data. We shall discuss the use of such mechanisms in detail in Section 6.7. The `sort` function in Program 6.1 uses a simple inline data type like the one discussed in Section 4.1, referring to the items being sorted only through its arguments and a few simple operations on the data. As usual, this approach allows us to use the same code to sort other types of items. For example, if the code for generating, storing, and printing random keys in the function `main` in Program 6.1 were changed to process floating-point numbers instead of integers, the only change that we would have to make outside of `main` is to change the `typedef` for `Item` from `int` to `float` (and we would not have to change `sort` at all). To provide such flexibility (while at the same time explicitly identifying those variables that hold items) our sort implementations will leave the data type of the items to be sorted unspecified as `Item`. For the moment, we can think of `Item` as `int` or `float`; in Section 6.7, we shall consider in detail data-type implementations that allow us to use our sort implementations for arbitrary items with floating-point numbers, strings, and other different types of keys, using mechanisms discussed in Chapters 3 and 4.

We can substitute for `sort` any of the array-sort implementations from this chapter, or from Chapters 7 through 10. They all assume that items of type `Item` are to be sorted, and they all take three arguments: the array, and the left and right bounds of the subarray to be sorted. They also all use `less` to compare keys in items and `exch` to exchange items (or the `compexch` combination). To differentiate sorting methods, we give our various sort routines different names. It is a simple matter to rename one of them, to change the driver, or to use function pointers to switch algorithms in a client program such as Program 6.1 without having to change any code in the sort implementation.

These conventions will allow us to examine natural and concise implementations of many array-sorting algorithms. In Sections 6.7 and 6.8, we shall consider a driver that illustrates how to use the implementations in more general contexts, and numerous data type

implementations. Although we are ever mindful of such packaging considerations, our focus will be on algorithmic issues, to which we now turn.

The example sort function in Program 6.1 is a variant of *insertion sort*, which we shall consider in detail in Section 6.3. Because it uses only compare-exchange operations, it is an example of a *nonadaptive* sort: The sequence of operations that it performs is independent of the order of the data. By contrast, an *adaptive* sort is one that performs different sequences of operations, depending on the outcomes of comparisons (less operations). Nonadaptive sorts are interesting because they are well suited for hardware implementation (see Chapter 11), but most of the general-purpose sorts that we consider are adaptive.

As usual, the primary performance parameter of interest is the running time of our sorting algorithms. The selection-sort, insertion-sort, and bubble-sort methods that we discuss in Sections 6.2 through 6.4 all require time proportional to N^2 to sort N items, as discussed in Section 6.5. The more advanced methods that we discuss in Chapters 7 through 10 can sort N items in time proportional to $N \log N$, but they are not always as good as the methods considered here for small N and in certain other special situations. In Section 6.6, we shall look at a more advanced method (shellsort) that can run in time proportional to $N^{3/2}$ or less, and, in Section 6.10, we shall see a specialized method (key-indexed sorting) that runs in time proportional to N for certain types of keys.

The analytic results described in the previous paragraph all follow from enumerating the basic operations (comparisons and exchanges) that the algorithms perform. As discussed in Section 2.2, we also must consider the costs of the operations, and we generally find it worthwhile to focus on the most frequently executed operations (the inner loop of the algorithm). Our goal is to develop efficient and reasonable implementations of efficient algorithms. In pursuit of this goal, we will not just avoid gratuitous additions to inner loops, but also look for ways to remove instructions from inner loops when possible. Generally, the best way to reduce costs in an application is to switch to a more efficient algorithm; the second best way is to tighten the inner loop. We shall consider both options in detail for sorting algorithms.

The amount of extra memory used by a sorting algorithm is the second important factor that we shall consider. Basically, the methods divide into three types: those that sort in place and use no extra memory except perhaps for a small stack or table; those that use a linked-list representation or otherwise refer to data through pointers or array indices, and so need extra memory for N pointers or indices; and those that need enough extra memory to hold another copy of the array to be sorted.

We frequently use sorting methods for items with multiple keys—we may even need to sort one set of items using different keys at different times. In such cases, it may be important for us to be aware whether or not the sorting method that we use has the following property:

Definition 6.1 *A sorting method is said to be stable if it preserves the relative order of items with duplicated keys in the file.*

For example, if an alphabetized list of students and their year of graduation is sorted by year, a stable method produces a list in which people in the same class are still in alphabetical order, but a nonstable method is likely to produce a list with no vestige of the original alphabetic order. Figure 6.1 shows an example. Often, people who are unfamiliar with stability are surprised by the way an unstable algorithm seems to scramble the data when they first encounter the situation.

Several (but not all) of the simple sorting methods that we consider in this chapter are stable. On the other hand, many (but not all) of the sophisticated algorithms that we consider in the next several chapters are not. If stability is vital, we can force it by appending a small index to each key before sorting or by lengthening the sort key in some other way. Doing this extra work is tantamount to using both keys for the sort in Figure 6.1; using a stable algorithm would be preferable. It is easy to take stability for granted; actually, few of the sophisticated methods that we see in later chapters achieve stability without using significant extra time or space.

As we have mentioned, sorting programs normally access items in one of two ways: either keys are accessed for comparison, or entire items are accessed to be moved. If the items to be sorted are large, it is wise to avoid shuffling them around by doing an *indirect sort*: we rearrange not the items themselves, but rather an array of pointers (or indices) such that the first pointer points to the smallest item, the

Adams	1
Black	2
Brown	4
Jackson	2
Jones	4
Smith	1
Thompson	4
Washington	2
White	3
Wilson	3
Adams	1
Smith	1
Washington	2
Jackson	2
Black	2
White	3
Wilson	3
Thompson	4
Brown	4
Jones	4
Adams	1
Smith	1
Black	2
Jackson	2
Washington	2
White	3
Wilson	3
Brown	4
Jones	4
Thompson	4

Figure 6.1
Stable-sort example

A sort of these records might be appropriate on either key. Suppose that they are sorted initially by the first key (top). A nonstable sort on the second key does not preserve the order in records with duplicate keys (center), but a stable sort does preserve the order (bottom).

second pointer points to the next smallest item, and so forth. We can keep keys either with the items (if the keys are large) or with the pointers (if the keys are small). We could rearrange the items after the sort, but that is often unnecessary, because we do have the capability to refer to them in sorted order (indirectly). We shall consider indirect sorting in Section 6.8.

Exercises

- ▷ 6.1 A child's sorting toy has i cards that fit on a peg in position i for i from 1 to 5. Write down the method that you use to put the cards on the pegs, assuming that you cannot tell from the card whether it fits on a peg (you have to try fitting it on).
- 6.2 A card trick requires that you put a deck of cards in order by suit (in the order spades, hearts, clubs, diamonds) and by rank within each suit. Ask a few friends to do this task (shuffling in between!) and write down the method(s) that they use.
- 6.3 Explain how you would sort a deck of cards with the restriction that the cards must be laid out face down in a row, and the only allowed operations are to check the values of two cards and (optionally) to exchange them.
- 6.4 Explain how you would sort a deck of cards with the restriction that the cards must be kept stacked in the deck, and the only allowed operations are to look at the value of the top two cards, to exchange the top two cards, and to move the top card to the bottom of the deck.
- 6.5 Give all sequences of three compare-exchange operations that will sort three elements (see Program 6.1).
- 6.6 Give a sequence of five compare-exchange operations that will sort four elements.
- 6.7 Write a client program that checks whether the sort routine being used is stable.
- 6.8 Checking that the array is sorted after `sort` provides no guarantee that the sort works. Why not?
- 6.9 Write a performance driver client program that runs `sort` multiple times on files of various sizes, measures the time taken for each run, and prints out or plots the average running times.
- 6.10 Write an exercise driver client program that runs `sort` on difficult or pathological cases that might turn up in practical applications. Examples include files that are already in order, files in reverse order, files where all keys are the same, files consisting of only two distinct values, and files of size 0 or 1.

6.2 Selection Sort

One of the simplest sorting algorithms works as follows. First, find the smallest element in the array, and exchange it with the element in the first position. Then, find the second smallest element and exchange it with the element in the second position. Continue in this way until the entire array is sorted. This method is called *selection sort* because it works by repeatedly selecting the smallest remaining element. Figure 6.2 shows the method in operation on a sample file.

Program 6.2 is an implementation of selection sort that adheres to our conventions. The inner loop is just a comparison to test a current element against the smallest element found so far (plus the code necessary to increment the index of the current element and to check that it does not exceed the array bounds); it could hardly be simpler. The work of moving the items around falls outside the inner loop: each exchange puts an element into its final position, so the number of exchanges is $N - 1$ (no exchange is needed for the final element). Thus the running time is dominated by the number of comparisons. In Section 6.5, we show this number to be proportional to N^2 , and examine more closely how to predict the total running time and how to compare selection sort with other elementary sorts.

A disadvantage of selection sort is that its running time depends only slightly on the amount of order already in the file. The process of finding the minimum element on one pass through the file does not seem to give much information about where the minimum might be on the next pass through the file. For example, the user of the sort might be surprised to realize that it takes about as long to run selection sort for a file that is already in order, or for a file with all keys equal, as it does for a randomly ordered file! As we shall see, other methods are better able to take advantage of order in the input file.

Despite its simplicity and evident brute-force approach, selection sort outperforms more sophisticated methods in one important application: it is the method of choice for sorting files with huge items and small keys. For such applications, the cost of moving the data dominates the cost of making comparisons, and no algorithm can sort a file with substantially less data movement than selection sort (see Property 6.5 in Section 6.5).

```

@ SORTING EXAMPLE
A S O R T I N G E X A M P L E
A A O R T I N G E X S M P L E
A A E R T I N G O X S M P L E
A A E E G I N T O X S M P L R
A A E E G I N T O X S M P L R
A A E E G I L T O X S M P N R
A A E E G I L M O X S T P N R
A A E E G I L M N X S T P O R
A A E E G I L M N O S T P X R
A A E E G I L M N O P T S X R
A A E E G I L M N O P R S X T
A A E E G I L M N O P R S X T
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X

```

Figure 6.2
Selection sort example

The first pass has no effect in this example, because there is no element in the array smaller than the A at the left. On the second pass, the other A is the smallest remaining element, so it is exchanged with the S in the second position. Then, the E near the middle is exchanged with the O in the third position on the third pass; then, the other E is exchanged with the R in the fourth position on the fourth pass; and so forth.

Program 6.2 Selection sort

For each i from 1 to $r-1$, exchange $a[i]$ with the minimum element in $a[i], \dots, a[r]$. As the index i travels from left to right, the elements to its left are in their final position in the array (and will not be touched again), so the array is fully sorted when i reaches the right end.

```
void selection(Item a[], int l, int r)
{ int i, j;
  for (i = l; i < r; i++)
    { int min = i;
      for (j = i+1; j <= r; j++)
        if (less(a[j], a[min])) min = j;
      exch(a[i], a[min]);
    }
}
```

Exercises

► 6.11 Show, in the style of Figure 6.2, how selection sort sorts the sample file **EASYQUESTION**.

6.12 What is the maximum number of exchanges involving any particular element during selection sort? What is the average number of exchanges involving an element?

6.13 Give an example of a file of N elements that maximizes the number of times the test `less(a[j], a[min])` fails (and, therefore, `min` gets updated) during the operation of selection sort.

○ 6.14 Is selection sort stable?

6.3 Insertion Sort

The method that people often use to sort bridge hands is to consider the elements one at a time, inserting each into its proper place among those already considered (keeping them sorted). In a computer implementation, we need to make space for the element being inserted by moving larger elements one position to the right, and then inserting the element into the vacated position. The `sort` function in Program 6.1 is an implementation of this method, which is called *insertion sort*.

As in selection sort, the elements to the left of the current index are in sorted order during the sort, but they are not in their final

position, as they may have to be moved to make room for smaller elements encountered later. The array is, however, fully sorted when the index reaches the right end. Figure 6.3 shows the method in operation on a sample file.

The implementation of insertion sort in Program 6.1 is straightforward, but inefficient. We shall now consider three ways to improve it, to illustrate a recurrent theme throughout many of our implementations: We want code to be succinct, clear, and efficient, but these goals sometimes conflict, so we must often strike a balance. We do so by developing a natural implementation, then seeking to improve it by a sequence of transformations, checking the effectiveness (and correctness) of each transformation.

First, we can stop doing complex operations when we encounter a key that is not larger than the key in the item being inserted, because the subarray to the left is sorted. Specifically, we can break out of the inner for loop in `sort` in Program 6.1 when the condition `less(a[j-1], a[j])` is true. This modification changes the implementation into an adaptive sort, and speeds up the program by about a factor of 2 for randomly ordered keys (see Property 6.2).

With the improvement described in the previous paragraph, we have two conditions that terminate the inner loop—we could recode it as a while loop to reflect that fact explicitly. A more subtle improvement of the implementation follows from noting that the test $j > 1$ is usually extraneous: indeed, it succeeds only when the element inserted is the smallest seen so far and reaches the beginning of the array. A commonly used alternative is to keep the keys to be sorted in $a[1]$ to $a[N]$, and to put a *sentinel* key in $a[0]$, making it at least as small as the smallest key in the array. Then, the test whether a smaller key has been encountered simultaneously tests both conditions of interest, making the inner loop smaller and the program faster.

Sentinels are sometimes inconvenient to use: perhaps the smallest possible key is not easily defined, or perhaps the calling routine has no room to include an extra key. Program 6.3 illustrates one way around these two problems for insertion sort: We make an explicit first pass over the array that puts the item with the smallest key in the first position. Then, we sort the rest of the array, with that first and smallest item now serving as sentinel. We generally shall avoid sentinels in our code, because it is often easier to understand code with

```

A S O R T I N G E X A M P L E
A SO R T I N G E X A M P L E
A OS R T I N G E X A M P L E
A O R ST I N G E X A M P L E
A O R S TI N G E X A M P L E
A I O R S T N G E X A M P L E
A I N O R S T G E X A M P L E
A G I N O R S T E X A M P L E
A E G I N O R S T X A M P L E
A E G I N O R S T X M P L E
A A E G I N O R S T X P L E
A A E G I M N O P R S T X L E
A A E G I L M N O P R S T X E
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X

```

Figure 6.3
Insertion sort example

During the first pass of insertion sort, the S in the second position is larger than the A, so it does not have to be moved. On the second pass, when the O in the third position is encountered, it is exchanged with the S to put A O S in sorted order, and so forth. Unshaded elements that are not circled are those that were moved one position to the right.

Program 6.3 Insertion sort

This code is an improvement over the implementation of `sort` in Program 6.1 because (i) it first puts the smallest element in the array into the first position, so that that element can serve as a sentinel; (ii) it does a single assignment, rather than an exchange, in the inner loop; and (iii) it terminates the inner loop when the element being inserted is in position. For each i , it sorts the elements $a[1], \dots, a[i]$ by moving one position to the right elements in the sorted list $a[1], \dots, a[i-1]$ that are larger than $a[i]$, then putting $a[i]$ into its proper position.

```
void insertion(Item a[], int l, int r)
{
    int i;
    for (i = r; i > l; i--) compexch(a[i-1], a[i]);
    for (i = l+2; i <= r; i++)
    {
        int j = i; Item v = a[i];
        while (less(v, a[j-1]))
        {
            a[j] = a[j-1]; j--;
        }
        a[j] = v;
    }
}
```

explicit tests, but we shall note situations where sentinels might be useful in making programs both simpler and more efficient.

The third improvement that we shall consider also involves removing extraneous instructions from the inner loop. It follows from noting that successive exchanges involving the same element are inefficient. If there are two or more exchanges, we have

$$t = a[j]; a[j] = a[j-1]; a[j-1] = t;$$

followed by

$$t = a[j-1]; a[j-1] = a[j-2]; a[j-2] = t;$$

and so forth. The value of t does not change between these two sequences, and we waste time storing it, then reloading it for the next exchange. Program 6.3 moves larger elements one position to the right instead of using exchanges, and thus avoids wasting time in this way.

Program 6.3 is an implementation of insertion sort that is more efficient than the one given in Program 6.1 (in Section 6.5, we shall see that it is nearly twice as fast). In this book, we are interested *both* in elegant and efficient algorithms *and* in elegant and efficient imple-

mentations of them. In this case, the underlying algorithms do differ slightly—we should properly refer to the `sort` function in Program 6.1 as a *nonadaptive insertion sort*. A good understanding of the properties of an algorithm is the best guide to developing an implementation that can be used effectively in an application.

Unlike that of selection sort, the running time of insertion sort primarily depends on the initial order of the keys in the input. For example, if the file is large and the keys are already in order (or even are nearly in order), then insertion sort is quick and selection sort is slow. We compare the algorithms in more detail in Section 6.5.

Exercises

▷ 6.15 Show, in the style of Figure 6.3, how insertion sort sorts the sample file `EASYQUESTION`.

6.16 Give an implementation of insertion sort with the inner loop coded as a `while` loop that terminates on one of two conditions, as described in the text.

6.17 For each of the conditions in the `while` loop in Exercise 6.16, describe a file of N elements where that condition is always false when the loop terminates.

○ 6.18 Is insertion sort stable?

6.19 Give a nonadaptive implementation of selection sort based on finding the minimum element with code like the first `for` loop in Program 6.3.

6.4 Bubble Sort

The first sort that many people learn, because it is so simple, is *bubble sort*: Keep passing through the file, exchanging adjacent elements that are out of order, continuing until the file is sorted. Bubble sort's prime virtue is that it is easy to implement, but whether it is actually easier to implement than insertion or selection sort is arguable. Bubble sort generally will be slower than the other two methods, but we consider it briefly for the sake of completeness.

Suppose that we always move from right to left through the file. Whenever the minimum element is encountered during the first pass, we exchange it with each of the elements to its left, eventually putting it into position at the left end of the array. Then on the second pass, the second smallest element will be put into position, and so forth. Thus, N passes suffice, and bubble sort operates as a type of selection

Program 6.4 Bubble sort

For each i from 1 to $r-1$, the inner (j) loop puts the minimum element among the elements in $a[i], \dots, a[r]$ into $a[i]$ by passing from right to left through the elements, compare–exchanging successive elements. The smallest one moves on all such comparisons, so it “bubbles” to the beginning. As in selection sort, as the index i travels from left to right through the file, the elements to its left are in their final position in the array.

```

void bubble(Item a[], int l, int r)
{ int i, j;
  for (i = l; i < r; i++)
    for (j = r; j > i; j--)
      compexch(a[j-1], a[j]);
}

```

A S O R T I N G E X A M P L E
A A (S) O R T I N G E X (E) M P L
A A E (E) S O R T I N G E X (L) M P
A A E E (E) S O R T I N G L X (M) P
A A E E E (G) S O R T I N G L X (M) P
A A E E E G (I) S O R T L (N) M P X
A A E E E G I L (S) O R T (M) N P X
A A E E E G I L M (S) O R T (N) P X
A A E E E G I L M N (S) O R T P X
A A E E E G I L M N O (S) P R T X
A A E E E G I L M N O P (S) R T X
A A E E E G I L M N O P (R) H S T X
A A E E E G I L M N O P R (S) T X
A A E E E G I L M N O P R S T (T) X
A A E E E G I L M N O P R S T X

Figure 6.4
Bubble sort example

Small keys percolate over to the left in bubble sort. As the sort moves from right to left, each key is exchanged with the one on its left until a smaller one is encountered. On the first pass, the E is exchanged with the L, the P, and the M before stopping at the A on the right; then the A moves to the beginning of the file, stopping at the other A, which is already in position. The i th smallest key reaches its final position after the i th pass, just as in selection sort, but other keys are moved closer to their final position, as well.

sort, although it does more work to get each element into position. Program 6.4 is an implementation, and Figure 6.4 shows an example of the algorithm in operation.

We can speed up Program 6.4 by carefully implementing the inner loop, in much the same way as we did in Section 6.3 for insertion sort (see Exercise 6.25). Indeed, comparing the code, Program 6.4 appears to be virtually identical to the nonadaptive insertion sort in Program 6.1. The difference between the two is that the inner for loop moves through the left (sorted) part of the array for insertion sort and through the right (not necessarily sorted) part of the array for bubble sort.

Program 6.4 uses only complex instructions and is therefore nonadaptive, but we can improve it to run more efficiently when the file is nearly in order by testing whether no exchanges at all are performed on one of the passes (and therefore the file is in sorted order, so we can break out of the outer loop). Adding this improvement will make bubble sort faster on some types of files, but it is generally not as effective as is changing insertion sort to break out of the inner loop, as discussed in detail in Section 6.5.

Exercises

► 6.20 Show, in the style of Figure 6.4, how bubble sort sorts the sample file EASYQUESTION.

6.21 Give an example of a file for which the number of exchanges done by bubble sort is maximized.

o 6.22 Is bubble sort stable?

6.23 Explain how bubble sort is preferable to the nonadaptive version of selection sort described in Exercise 6.19.

• 6.24 Do experiments to determine how many passes are saved, for random files of N elements, when you add to bubble sort a test to terminate when the file is sorted.

6.25 Develop an efficient implementation of bubble sort, with as few instructions as possible in the inner loop. Make sure that your “improvements” do not slow down the program!

6.5 Performance Characteristics of Elementary Sorts

Selection sort, insertion sort, and bubble sort are all quadratic-time algorithms both in the worst and in the average case, and none requires extra memory. Their running times thus differ by only a constant factor, but they operate quite differently, as illustrated in Figures 6.5 through 6.7.

Generally, the running time of a sorting algorithm is proportional to the number of comparisons that the algorithm uses, to the number of times that items are moved or exchanged, or to both. For random input, comparing the methods involves studying constant-factor differences in the numbers of comparisons and exchanges and constant-factor differences in the lengths of the inner loops. For input with special characteristics, the running times of the methods may differ by more than a constant factor. In this section, we look closely at the analytic results in support of this conclusion.

Property 6.1 *Selection sort uses about $N^2/2$ comparisons and N exchanges.*

We can verify this property easily by examining the sample run in Figure 6.2, which is an N -by- N table in which unshaded letters correspond to comparisons. About one-half of the elements in the table are unshaded—those above the diagonal. The $N - 1$ (not the final one) elements on the diagonal each correspond to an exchange. More precisely, examination of the code reveals that, for each i from 1 to $N - 1$, there is one exchange and $N - i$ comparisons, so there is a total

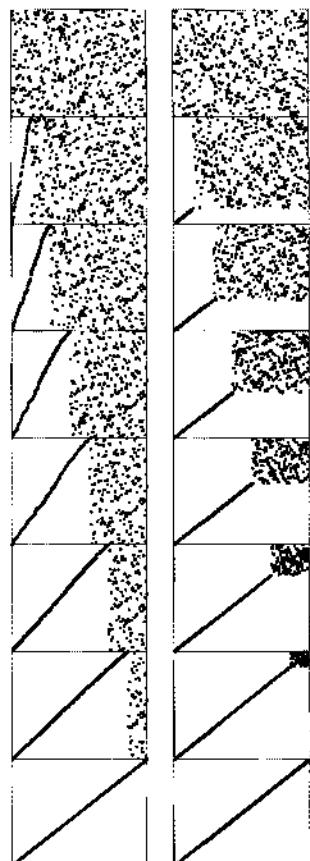


Figure 6.5
Dynamic characteristics of insertion and selection sorts

These snapshots of insertion sort (left) and selection sort (right) in action on a random permutation illustrate how each method progresses through the sort. We represent an array being sorted by plotting i vs. $a[i]$ for each i . Before the sort, the plot is uniformly random; after the sort, it is a diagonal line from bottom left to top right. Insertion sort never looks ahead of its current position in the array; selection sort never looks back.

of $N - 1$ exchanges and $(N - 1) + (N - 2) + \dots + 2 + 1 = N(N - 1)/2$ comparisons. These observations hold no matter what the input data are; the only part of selection sort that does depend on the input is the number of times that `min` is updated. In the worst case, this quantity could also be quadratic; in the average case, however, it is just $O(N \log N)$ (see reference section), so we can expect the running time of selection sort to be insensitive to the input. ■

Property 6.2 *Insertion sort uses about $N^2/4$ comparisons and $N^2/4$ half-exchanges (moves) on the average, and twice that many at worst.*

As implemented in Program 6.3, the number of comparisons and of moves is the same. Just as for Property 6.1, this quantity is easy to visualize in the N -by- N diagram in Figure 6.3 that gives the details of the operation of the algorithm. Here, the elements below the diagonal are counted—all of them, in the worst case. For random input, we expect each element to go about halfway back, on the average, so one-half of the elements below the diagonal should be counted. ■

Property 6.3 *Bubble sort uses about $N^2/2$ comparisons and $N^2/2$ exchanges on the average and in the worst case.*

The i th bubble sort pass requires $N - i$ compare-exchange operations, so the proof goes as for selection sort. When the algorithm is modified to terminate when it discovers that the file is sorted, the running time depends on the input. Just one pass is required if the file is already in order, but the i th pass requires $N - i$ comparisons and exchanges if the file is in reverse order. The average-case performance is not significantly better than the worst case, as stated, although the analysis that demonstrates this fact is complicated (see reference section). ■

Although the concept of a partially sorted file is necessarily rather imprecise, insertion sort and bubble sort work well for certain types of nonrandom files that often arise in practice. General-purpose sorts are commonly misused for such applications. For example, consider the operation of insertion sort on a file that is already sorted. Each element is immediately determined to be in its proper place in the file, and the total running time is linear. The same is true for bubble sort, but selection sort is still quadratic.

Definition 6.2 *An inversion is a pair of keys that are out of order in the file.*

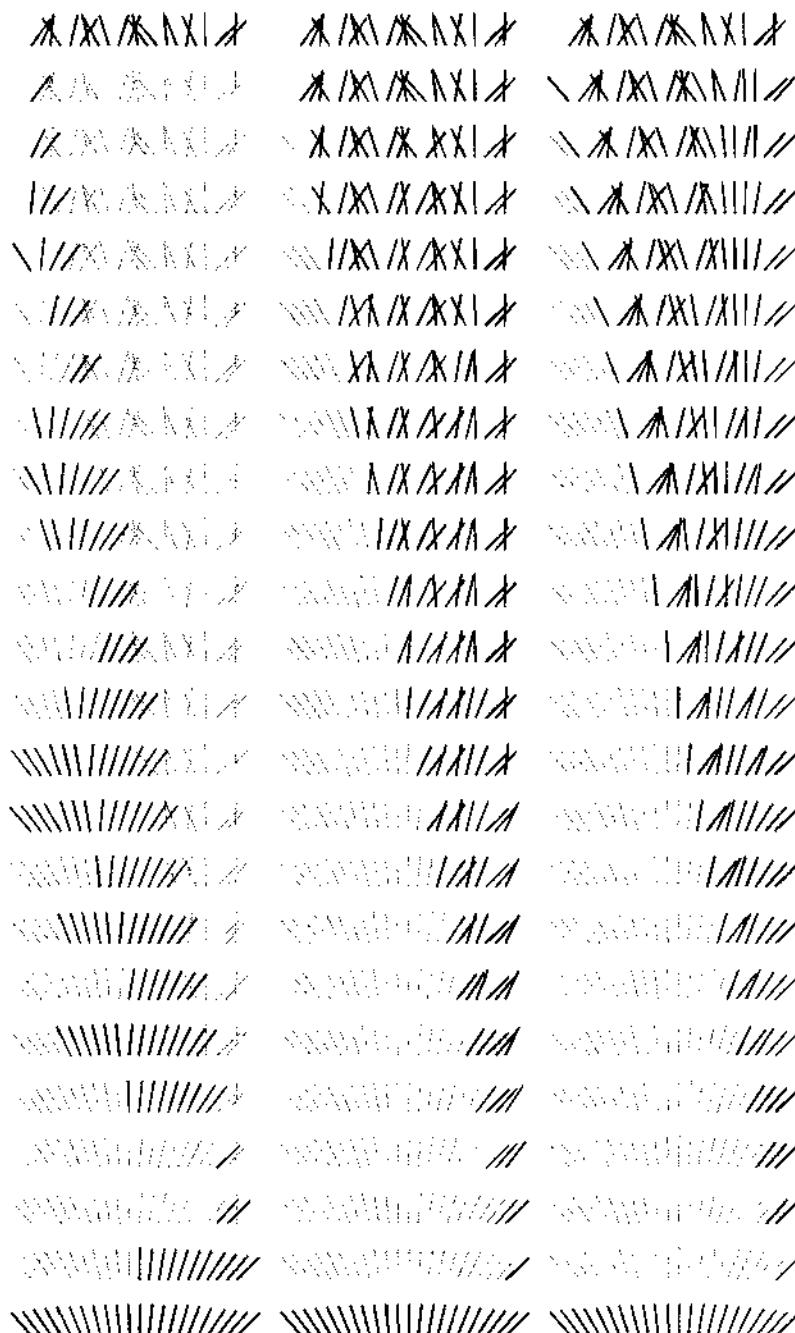


Figure 6.6
Comparisons and exchanges
in elementary sorts

This diagram highlights the differences in the way that insertion sort, selection sort, and bubble sort bring a file into order. The file to be sorted is represented by lines that are to be sorted according to their angles. Black lines correspond to the items accessed during each pass of the sort; gray lines correspond to items not touched. For insertion sort (left), the element to be inserted goes about halfway back through the sorted part of the file on each pass. Selection sort (center) and bubble sort (right) both go through the entire unsorted part of the array to find the next smallest element there for each pass; the difference between the methods is that bubble sort exchanges any adjacent out-of-order elements that it encounters, whereas selection sort just exchanges the minimum into position. The effect of this difference is that the unsorted part of the array becomes more nearly sorted as bubble sort progresses.

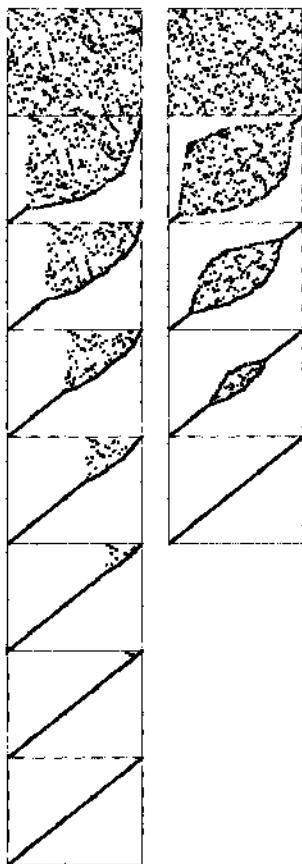


Figure 6.7
Dynamic characteristics of
two bubble sorts

Standard bubble sort (left) operates in a manner similar to selection sort in that each pass brings one element into position, but it also brings some order into other parts of the array, in an asymmetric manner. Changing the scan through the array to alternate between beginning to end and end to beginning gives a version of bubble sort called shaker sort (right), which finishes more quickly (see Exercise 6.30).

To count the number of inversions in a file, we can add up, for each element, the number of elements to its left that are greater (we refer to this quantity as the number of inversions corresponding to the element). But this count is precisely the distance that the elements have to move when inserted into the file during insertion sort. A file that has some order will have fewer inversions than will one that is arbitrarily scrambled.

In one type of partially sorted file, each item is close to its final position in the file. For example, some people sort their hand in a card game by first organizing the cards by suit, to put their cards close to their final position, then considering the cards one by one. We shall be considering a number of sorting methods that work in much the same way—they bring elements close to final positions in early stages to produce a partially sorted file with every element not far from where it ultimately must go. Insertion sort and bubble sort (but not selection sort) are efficient methods for sorting such files.

Property 6.4 *Insertion sort and bubble sort use a linear number of comparisons and exchanges for files with at most a constant number of inversions corresponding to each element.*

As just mentioned, the running time of insertion sort is directly proportional to the number of inversions in the file. For bubble sort (here, we are referring to Program 6.4, modified to terminate when the file is sorted), the proof is more subtle (see Exercise 6.29). Each bubble sort pass reduces the number of smaller elements to the right of any element by precisely 1 (unless the number was already 0), so bubble sort uses at most a constant number of passes for the types of files under consideration, and therefore does at most a linear number of comparisons and exchanges. ■

In another type of partially sorted file, we perhaps have appended a few elements to a sorted file or have edited a few elements in a sorted file to change their keys. This kind of file is prevalent in sorting applications. Insertion sort is an efficient method for such files; bubble sort and selection sort are not.

Property 6.5 *Insertion sort uses a linear number of comparisons and exchanges for files with at most a constant number of elements having more than a constant number of corresponding inversions.*

Table 6.1 Empirical study of elementary sorting algorithms

Insertion sort and selection sort are about twice as fast as bubble sort for small files, but running times grow quadratically (when the file size grows by a factor of 2, the running time grows by a factor of 4). None of the methods are useful for large randomly ordered files—for example, the numbers corresponding to those in this table are less than 2 for the shellsort algorithm in Section 6.6. When comparisons are expensive—for example, when the keys are strings—then insertion sort is much faster than the other two because it uses many fewer comparisons. Not included here is the case where exchanges are expensive; then selection sort is best.

N	32-bit integer keys					string keys		
	S	I*	I	B	B*	S	I	B
1000	5	7	4	11	8	13	8	19
2000	21	29	15	45	34	56	31	78
4000	85	119	62	182	138	228	126	321

Key:

- S Selection sort (Program 6.2)
- I* Insertion sort, exchange-based (Program 6.1)
- I Insertion sort (Program 6.3)
- B Bubble sort (Program 6.4)
- B* Shaker sort (Exercise 6.30)

The running time of insertion sort depends on the total number of inversions in the file, and does not depend on the way in which the inversions are distributed. ■

To draw conclusions about running time from Properties 6.1 through 6.5, we need to analyze the relative cost of comparisons and exchanges, a factor that in turn depends on the size of the items and keys (see Table 6.1). For example, if the items are one-word keys, then an exchange (four array accesses) should be about twice as expensive as a comparison. In such a situation, the running times of selection and insertion sort are roughly comparable, but bubble sort is slower. But if the items are large in comparison to the keys, then selection sort will be best.

Property 6.6 *Selection sort runs in linear time for files with large items and small keys.*

Let M be the ratio of the size of the item to the size of the key. Then we can assume the cost of a comparison to be 1 time unit and the cost of an exchange to be M time units. Selection sort takes about $N^2/2$ time units for comparisons and about NM time units for exchanges. If M is larger than a constant multiple of N , then the NM term dominates the N^2 term, so the running time is proportional to NM , which is proportional to the amount of time that would be required to move all the data. ■

For example, if we have to sort 1000 items that consist of 1-word keys and 1000 words of data each, and we actually have to rearrange the items, then we cannot do better than selection sort, since the running time will be dominated by the cost of moving all 1 million words of data. In Section 6.8, we shall see alternatives to rearranging the data.

Exercises

- ▷ 6.26 Which of the three elementary methods (selection sort, insertion sort, or bubble sort) runs fastest for a file with all keys identical?
- 6.27 Which of the three elementary methods runs fastest for a file in reverse order?
- 6.28 Give an example of a file of 10 elements (use the keys A through J) for which bubble sort uses fewer comparisons than insertion sort, or prove that no such file exists.
- 6.29 Show that each bubble sort pass reduces by precisely 1 the number of elements to the left of each element that are greater (unless that number was already 0).
- 6.30 Implement a version of bubble sort that alternates left-to-right and right-to-left passes through the data. This (faster but more complicated) algorithm is called *shaker sort* (see Figure 6.7).
- 6.31 Show that Property 6.5 does not hold for shaker sort (see Exercise 6.30).
- 6.32 Implement selection sort in PostScript (see Section 4.3), and use your implementation to draw figures like Figures 6.5 through 6.7. You may try a recursive implementation, or read the manual to learn about loops and arrays in PostScript.

6.6 Shellsort

Insertion sort is slow because the only exchanges it does involve adjacent items, so items can move through the array only one place at a time. For example, if the item with the smallest key happens to be at the end of the array, N steps are needed to get it where it belongs. *Shellsort* is a simple extension of insertion sort that gains speed by allowing exchanges of elements that are far apart.

The idea is to rearrange the file to give it the property that taking every h th element (starting anywhere) yields a sorted file. Such a file is said to be *h -sorted*. Put another way, an h -sorted file is h independent sorted files, interleaved together. By h -sorting for some large values of h , we can move elements in the array long distances and thus make it easier to h -sort for smaller values of h . Using such a procedure for any sequence of values of h that ends in 1 will produce a sorted file: that is the essence of shellsort.

One way to implement shellsort would be, for each h , to use insertion sort independently on each of the h subfiles. Despite the apparent simplicity of this process, we can use an even simpler approach, precisely because the subfiles are independent. When h -sorting the file, we simply insert it among the previous elements in its h -subfile by moving larger elements to the right (see Figure 6.8). We accomplish this task by using the insertion-sort code, but modified to increment or decrement by h instead of 1 when moving through the file. This observation reduces the shellsort implementation to nothing more than an insertion-sort-like pass through the file for each increment, as in Program 6.5. The operation of this program is illustrated in Figure 6.9.

How do we decide what increment sequence to use? In general, this question is a difficult one to answer. Properties of many different increment sequences have been studied in the literature, and some have been found that work well in practice, but no provably best sequence has been found. In practice, we generally use sequences that decrease roughly geometrically, so the number of increments is logarithmic in the size of the file. For example, if each increment is about one-half of the previous, then we need only about 20 increments to sort a file of 1 million elements; if the ratio is about one-quarter, then 10 increments will suffice. Using as few increments as possible is an important consideration that is easy to respect—we also need to

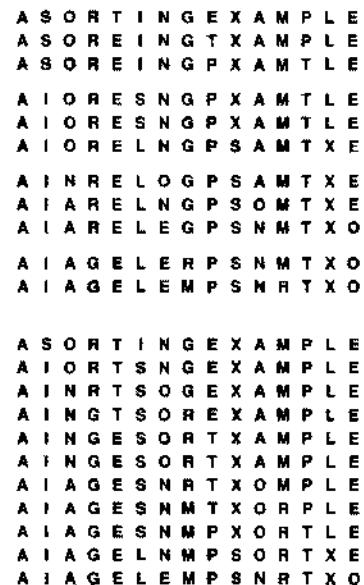


Figure 6.8
Interleaving 4-sorts

The top part of this diagram shows the process of 4-sorting a file of 16 elements by first insertion sorting the subfile at positions 0, 4, 8, 12, then insertion sorting the subfile at positions 1, 5, 9, 13, then insertion sorting the subfile at positions 2, 6, 10, 14, then insertion sorting the subfile at positions 3, 7, 11. But the four subfiles are independent, so we can achieve the same result by inserting each element into position into its subfile, going back four at a time (bottom). Taking the first row in each section of the top diagram, then the second row in each section, and so forth, gives the bottom diagram.

```

A S O R T I N G E X A M P L E
A S O R T I N G E X A M P L E
A E O R T I N G E X A M P L S

A E O R T I N G E X A M P L S
A E O R T I N G E X A M P L S
A E N R T I O G E X A M P L S
A E N G T I O R E X A M P L S
A E N G E I O R T X A M P L S
A E N G E I O R T X A M P L S
A E A G E I N R T X O M P L S
A E A G E I N M T X O M P L S
A E A G E I N M P X O R T L S
A E A G E I N M P L O R T X S
A E A G E I N M P L O R T X S

A E A G E I N M P L O R T X S
A A E G E I N M P L O R T X S
A A E E G I N M P L O R T X S
A A E E G I N M P L O R T X S
A A E E G I N M P L O R T X S
A A E E G I N M P L O R T X S
A A E E G I M N P L O R T X S
A A E E G I M N P L O R T X S
A A E E G I L M N P O R T X S
A A E E G I L M N O P R T X S
A A E E G I L M N O P R T X S
A A E E G I L M N O P R T X S
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X

```

Figure 6.9
Shellsort example

Sorting a file by 13-sorting (top), then 4-sorting (center), then 1-sorting (bottom) does not involve many comparisons (as indicated by the unshaded elements). The final pass is just insertion sort, but no element has to move far because of the order in the file due to the first two passes.

Program 6.5 Shellsort

If we do not use sentinels and then replace every occurrence of “1” by “h” in insertion sort, the resulting program *h*-sorts the file. Adding an outer loop to change the increments leads to this compact shellsort implementation, which uses the increment sequence 1 4 13 40 121 364 1093 3280 9841 ...

```

void shellsort(Item a[], int l, int r)
{
    int i, j, h;
    for (h = 1; h <= (r-1)/9; h = 3*h+1);
    for ( ; h > 0; h /= 3)
        for (i = 1+h; i <= r; i++)
            { int j = i; Item v = a[i];
              while (j >= 1+h && less(v, a[j-h]))
                  { a[j] = a[j-h]; j -= h; }
              a[j] = v;
            }
}

```

consider arithmetical interactions among the increments such as the size of their common divisors and other properties.

The practical effect of finding a good increment sequence is limited to perhaps a 25% speedup, but the problem presents an intriguing puzzle that provides a good example of the inherent complexity in an apparently simple algorithm.

The increment sequence 1 4 13 40 121 364 1093 3280 9841 ... that is used in Program 6.5, with a ratio between increments of about one-third, was recommended by Knuth in 1969 (*see reference section*). It is easy to compute (start with 1, generate the next increment by multiplying by 3 and adding 1) and leads to a relatively efficient sort, even for moderately large files, as illustrated in Figure 6.10.

Many other increment sequences lead to a more efficient sort but it is difficult to beat the sequence in Program 6.5 by more than 20% even for relatively large *N*. One increment sequence that does so is 1 8 23 77 281 1073 4193 16577 ..., the sequence $4^{i+1} + 3 \cdot 2^i + 1$ for $i > 0$, which has provably faster worst-case behavior (see Property 6.10). Figure 6.12 shows that this sequence and Knuth’s sequence—and many other sequences—have similar dynamic characteristics for large files.

The possibility that even better increment sequences exist is still real. A few ideas on improved increment sequences are explored in the exercises.

On the other hand, there are some bad increment sequences: for example 1 2 4 8 16 32 64 128 256 512 1024 2048 . . . (the original sequence suggested by Shell when he proposed the algorithm in 1959 (see reference section)) is likely to lead to bad performance because elements in odd positions are not compared against elements in even positions until the final pass. The effect is noticeable for random files, and is catastrophic in the worst case: The method degenerates to require quadratic running time if, for example, the half of the elements with the smallest values are in even positions and the half of the elements with the largest values are in the odd positions (See Exercise 6.36.)

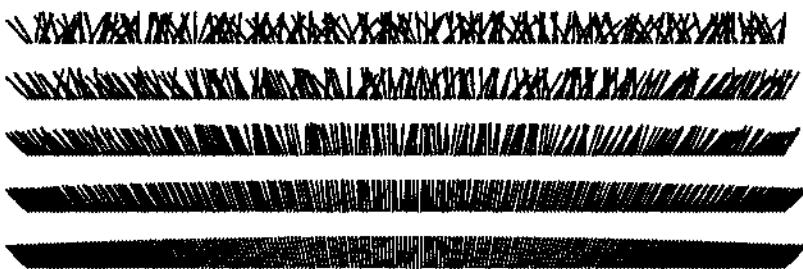
Program 6.5 computes the next increment by dividing the current one by 3, after initializing to ensure that the same sequence is always used. Another option is just to start with $h = N/3$ or with some other function of N . It is best to avoid such strategies, because bad sequences of the type described in the previous paragraph are likely to turn up for some values of N .

Our description of the efficiency of shellsort is necessarily imprecise, because no one has been able to analyze the algorithm. This gap in our knowledge makes it difficult not only to evaluate different increment sequences, but also to compare shellsort with other methods analytically. Not even the functional form of the running time for shellsort is known (furthermore, the form depends on the increment sequence). Knuth found that the functional forms $N(\log N)^2$ and $N^{1.25}$ both fit the data reasonably well, and later research suggests that a more complicated function of the form $N^{1+1/\sqrt{\lg N}}$ is involved for some sequences.

We conclude this section by digressing into a discussion of several facts about the analysis of shellsort that *are known*. Our primary purpose in doing so is to illustrate that even algorithms that are apparently simple can have complex properties, and that the analysis of algorithms is not just of practical importance but also can be intellectually challenging. Readers intrigued by the idea of finding a new and improved shellsort increment sequence may find the information that follows useful; other readers may wish to skip to Section 6.7.

Figure 6.10
Shellsorting a random permutation

The effect of each of the passes in Shellsort is to bring the file as a whole closer to sorted order. The file is first 40-sorted, then 13-sorted, then 4-sorted, then 1-sorted. Each pass brings the file closer to sorted order.



Property 6.7 *The result of h -sorting a file that is k -ordered is a file that is both h - and k -ordered.*

This fact seems obvious, but is tricky to prove (see Exercise 6.47). ■

Property 6.8 *Shellsort does less than $N(h - 1)(k - 1)/g$ comparisons to g -sort a file that is h - and k -ordered, provided that h and k are relatively prime.*

The basis for this fact is illustrated in Figure 6.11. No element farther than $(h - 1)(k - 1)$ positions to the left of any given element x can be greater than x , if h and k are relatively prime (see Exercise 6.43). When g -sorting, we examine at most one out of every g of those elements. ■

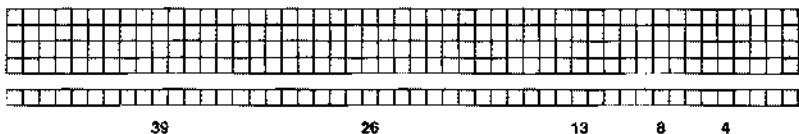
Property 6.9 *Shellsort does less than $O(N^{3/2})$ comparisons for the increments $1\ 4\ 13\ 40\ 121\ 364\ 1093\ 3280\ 9841\ \dots$*

For large increments, there are h subfiles of size about N/h , for a worst-case cost about N^2/h . For small increments, Property 6.8 implies that the cost is about Nh . The result follows if we use the better of these bounds for each increment. It holds for any relatively prime sequence that grows exponentially. ■

Property 6.10 *Shellsort does less than $O(N^{4/3})$ comparisons for the increments $1\ 8\ 23\ 77\ 281\ 1073\ 4193\ 16577\ \dots$*

The proof of this property is along the lines of the proof of Property 6.9. The property analogous to Property 6.8 implies that the cost for small increments is about $Nh^{1/2}$. Proof of this property requires number theory that is beyond the scope of this book (see reference section). ■

The increment sequences that we have discussed to this point are effective because successive elements are relatively prime. Another



family of increment sequences is effective precisely because successive elements are *not* relatively prime.

In particular, the proof of Property 6.8 implies that, in a file that is 2-ordered and 3-ordered, each element moves at most one position during the final insertion sort. That is, such a file can be sorted with one bubble-sort pass (the extra loop in insertion sort is not needed). Now, if a file is 4-ordered and 6-ordered, then it also follows that each element moves at most one position when we are 2-sorting it (because each subfile is 2-ordered and 3-ordered); and if a file is 6-ordered and 9-ordered, each element moves at most one position when we are 3-sorting it. Continuing this line of reasoning, we are led to the following idea, which was developed by Pratt in 1971 (see reference section).

Property 6.11 *Shellsort does less than $O(N(\log N)^2)$ comparisons for the increments 1 2 3 4 6 9 8 12 18 27 16 24 36 54 81*

Consider the following triangle of increments, where each number in the triangle is two times the number above and to the right of it and also three times the number above and to the left of it.

1
2 3
4 6 9
8 12 18 27
16 24 36 54 81
32 48 72 108 162 243
64 96 144 216 324 486 729

If we use these numbers from bottom to top and right to left as a shellsort increment sequence, then every increment x after the bottom row is preceded by $2x$ and $3x$, so every subfile is 2-ordered and 3-ordered, and no element moves more than one position during the

Figure 6.11
A 4- and 13- ordered file.

The bottom row depicts an array, with shaded boxes depicting those items that must be smaller than or equal to the item at the far right, if the array is both 4- and 13-ordered. The four rows at top depict the origin of the pattern. If the item at right is at array position i , then 4-ordering means that items at array positions $i - 4, i - 8, i - 12, \dots$ are smaller or equal (top); 13-ordering means that the item at $i - 13$, and, therefore, because of 4-ordering, the items at $i - 17, i - 21, i - 25, \dots$ are smaller or equal (second from top); also, the item at $i - 26$, and, therefore, because of 4-ordering, the items at $i - 30, i - 34, i - 38, \dots$ are smaller or equal (third from top); and so forth. The white squares remaining are those that could be larger than the item at left; there are at most 18 such items (and the one that is farthest away is at $i - 36$). Thus, at most $18N$ comparisons are required for an insertion sort of a 13-ordered and 4-ordered file of size N .

Table 6.2 Empirical study of shellsort increment sequences

Shellsort is many times faster than the other elementary methods even when the increments are powers of 2, but some increment sequences can speed it up by another factor of 5 or more. The three best sequences in this table are totally different in design. Shellsort is a practical method even for large files, particularly by contrast with selection sort, insertion sort, and bubble sort (see Table 6.1).

<i>N</i>	O	K	G	S	P	I
12500	16	6	6	5	6	6
25000	37	13	11	12	15	10
50000	102	31	30	27	38	26
100000	303	77	60	63	81	58
200000	817	178	137	139	180	126

Key:

O 1 2 4 8 16 32 64 128 256 512 1024 2048 ...

K 1 4 13 40 121 364 1093 3280 9841 ... (Property 6.9)

G 1 2 4 10 23 51 113 249 548 1207 2655 5843 ... (Exercise 6.40)

S 1 8 23 77 281 1073 4193 16577 ... (Property 6.10)

P 1 7 8 49 56 64 343 392 448 512 2401 2744 ... (Exercise 6.44)

I 1 5 19 41 109 209 505 929 2161 3905 ... (Exercise 6.45)

entire sort! The number of increments in the triangle that are less than N is certainly less than $(\log_2 N)^2$. ■

Pratt's increments tend not to work as well as the others in practice, because there are too many of them. We can use the same principle to build an increment sequence from *any* two relatively prime numbers h and k . Such sequences do well in practice because the worst-case bounds corresponding to Property 6.11 overestimate the cost for random files.

The problem of designing good increment sequences for shellsort provides an excellent example of the complex behavior of a simple algorithm. We certainly will not be able to focus at this level of detail on all the algorithms that we encounter (not only do we not have the space, but also, as we did with shellsort, we might encounter mathematical analysis beyond the scope of this book, or even open research prob-

lems). However, many of the algorithms in this book are the product of extensive analytic and empirical studies by many researchers over the past several decades, and we can benefit from this work. This research illustrates that the quest for improved performance can be both intellectually challenging and practically rewarding, even for simple algorithms. Table 6.2 gives empirical results that show that several approaches to designing increment sequences work well in practice; the relatively short sequence 1 8 23 77 281 1073 4193 16577 ... is among the simplest to use in a shellsort implementation.

Figure 6.13 shows that shellsort performs reasonably well on a variety of kinds of files, rather than just on random ones. Indeed, constructing a file for which shellsort runs slowly for a given increment sequence is a challenging exercise (see Exercise 6.42). As we have mentioned, there are some bad increment sequences for which shellsort may require a quadratic number of comparisons in the worst case (see Exercise 6.36), but much lower bounds have been shown to hold for a wide variety of sequences.

Shellsort is the method of choice for many sorting applications because it has acceptable running time even for moderately large files and requires a small amount of code that is easy to get working. In the next few chapters, we shall see methods that are more efficient, but they are perhaps only twice as fast (if that much) except for large N , and they are significantly more complicated. In short, if you need a quick solution to a sorting problem, and do not want to bother with interfacing to a system sort, you can *use shellsort*, then determine sometime later whether the extra work required to replace it with a more sophisticated method will be worthwhile.

Exercises

▷ 6.33 Is shellsort stable?

6.34 Show how to implement a shellsort with the increments 1 8 23 77 281 1073 4193 16577 ..., with direct calculations to get successive increments in a manner similar to the code given for Knuth's increments.

▷ 6.35 Give diagrams corresponding to Figures 6.8 and 6.9 for the keys E A S Y Q U E S T I O N.

6.36 Find the running time when you use shellsort with the increments 1 2 4 8 16 32 64 128 256 512 1024 2048 ... to sort a file consisting of the integers 1, 2, ..., N in the odd positions and $N+1, N+2, \dots, 2N$ in the even positions.

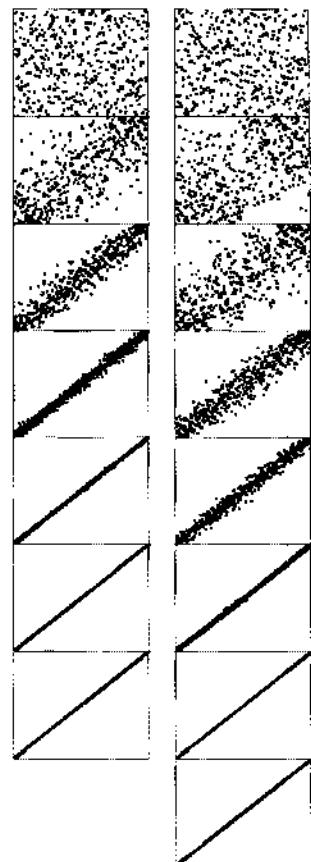
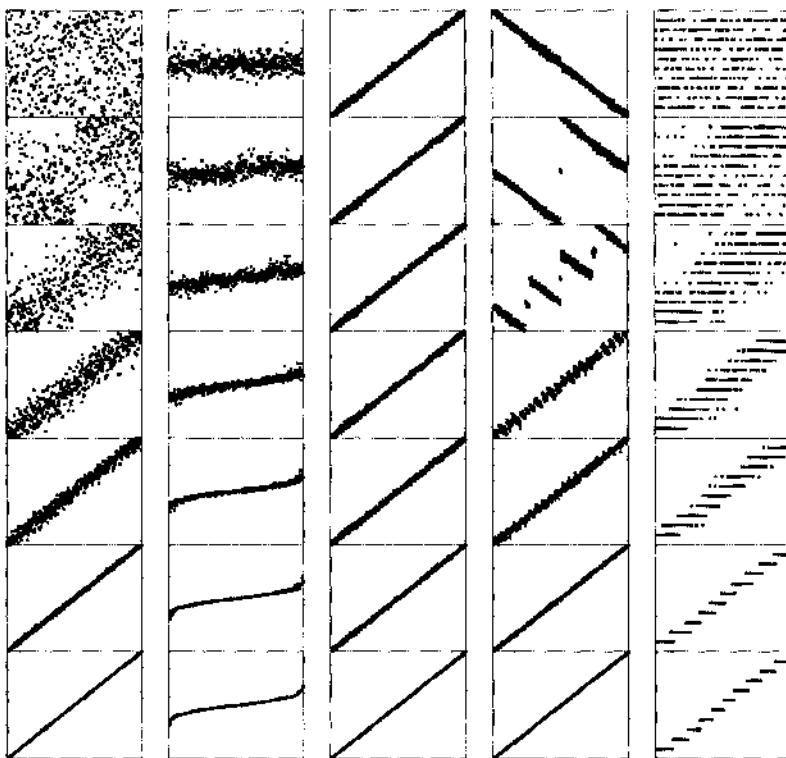


Figure 6.12
Dynamic characteristics of shellsort (two different increment sequences)

In this representation of shellsort in operation, it appears as though a rubber band, anchored at the corners, is pulling the points toward the diagonal. Two increment sequences are depicted: 121 40 13 4 1 (left) and 209 109 41 19 5 1 (right). The second requires one more pass than the first, but is faster because each pass is more efficient.

Figure 6.13
Dynamic characteristics of shellsort for various types of files

These diagrams show shellsort, with the increments 209 109 41 19 5 1, in operation on files that are random, Gaussian, nearly ordered, nearly reverse-ordered, and randomly ordered with 10 distinct key values (left to right, on the top). The running time for each pass depends on how well ordered the file is when the pass begins. After a few passes, these files are similarly ordered; thus, the running time is not particularly sensitive to the input.



6.37 Write a driver program to compare increment sequences for shellsort. Read the sequences from standard input, one per line, then use them all to sort 10 random files of size N for $N = 100$, 1000, and 10000. Count comparisons, or measure actual running times.

- **6.38** Run experiments to determine whether adding or deleting an increment can improve the increment sequence 1 8 23 77 281 1073 4193 16577 ... for $N = 10000$.
- **6.39** Run experiments to determine the value of x that leads to the lowest running time for random files when the 13 is replaced by x in the increment sequence 1 4 13 40 121 364 1093 3280 9841 ... used for $N = 10000$.
- **6.40** Run experiments to determine the value of α that leads to the lowest running time for random files for the increment sequence 1, $\lfloor \alpha \rfloor$, $\lfloor \alpha^2 \rfloor$, $\lfloor \alpha^3 \rfloor$, $\lfloor \alpha^4 \rfloor$, ...; for $N = 10000$.
- **6.41** Find the three-increment sequence that uses as small a number of comparisons as you can find for random files of 1000 elements.

- 6.42 Construct a file of 100 elements for which shellsort, with the increments 1 8 23 77, uses as large a number of comparisons as you can find.
- 6.43 Prove that any number greater than or equal to $(h - 1)(k - 1)$ can be expressed as a linear combination (with nonnegative coefficients) of h and k , if h and k are relatively prime. *Hint:* Show that, if any two of the first $h - 1$ multiples of k have the same remainder when divided by h , then h and k must have a common factor.
- 6.44 Run experiments to determine the values of h and k that lead to the lowest running times for random files when a Pratt-like sequence based on h and k is used for sorting 10000 elements.
- 6.45 The increment sequence 1 5 19 41 109 209 505 929 2161 3905 ... is based on merging the sequences $9 \cdot 4^i - 9 \cdot 2^i + 1$ and $4^i - 3 \cdot 2^i + 1$ for $i > 0$. Compare the results of using these sequences individually and using the merged result, for sorting 10000 elements.
- 6.46 We derive the increment sequence 1 3 7 21 48 112 336 861 1968 4592 13776 ... by starting with a base sequence of relatively prime numbers, say 1 3 7 16 41 101, then building a triangle, as in Pratt's sequence, this time generating the i th row in the triangle by multiplying the first element in the $i - 1$ st row by the i th element in the base sequence; and multiplying every element in the $i - 1$ st row by the $i + 1$ st element in the base sequence. Run experiments to find a base sequence that improves on the one given for sorting 10000 elements.
- 6.47 Complete the proofs of Properties 6.7 and 6.8.
- 6.48 Implement a shellsort that is based on the shaker sort algorithm of Exercise 6.30, and compare with the standard algorithm. *Note:* Your increment sequences should be substantially different from those for the standard algorithm.

6.7 Sorting of Other Types of Data

Although it is reasonable to learn most algorithms by thinking of them as simply sorting arrays of numbers into numerical order or characters into alphabetical order, it is also worthwhile to recognize that the algorithms are largely independent of the type of items being sorted, and that is not difficult to move to a more general setting. We have talked in detail about breaking our programs into independent modules to implement data types, and abstract data types (see Chapters 3 and 4); in this section, we consider ways in which we can apply the concepts discussed there to make our sorting implementations useful for various types of data.

Program 6.6 Sort driver for arrays

This driver for basic array sorts uses two explicit interfaces: one for the functions that initialize and print (and sort!) arrays, and the other for a data type that encapsulates the operations that we perform on generic items. The first allows us to compile the functions for arrays separately and perhaps to use them in other drivers; the second allows us to sort other types of data with the same sort code.

```
#include <stdlib.h>
#include "Item.h"
#include "Array.h"
main(int argc, char *argv[])
{ int i, N = atoi(argv[1]), sw = atoi(argv[2]);
  Item *a = malloc(N*sizeof(Item));
  if (sw) randinit(a, N); else scaninit(a, &N);
  sort(a, 0, N-1);
  show(a, 0, N-1);
}
```

Specifically, we consider implementations, interfaces, and client programs for:

- *Items*, or generic objects to be sorted
- *Arrays* of items

The item data type provides us with a way to use our sort code for any type of data for which certain basic operations are defined. The approach is effective both for simple data types and for abstract data types, and we shall consider numerous implementations. The array interface is less critical to our mission; we include it to give us an example of a multiple-module program that uses multiple data types. We consider just one (straightforward) implementation of the array interface.

Program 6.6 is a client program with the same general functionality of the main program in Program 6.1, but with the code for manipulating arrays and items encapsulated in separate modules, which gives us, in particular, the ability to test various sort programs on various different types of data, by substituting various different modules, but without changing the client program at all. To complete the imple-

Program 6.7 Interface for array data type

This `Array.h` interface defines high-level functions for arrays of abstract items: initialize random values, initialize values read from standard input, print the contents, and sort the contents. The item types are defined in a separate interface (see Program 6.9).

```
void randinit(Item [], int);
void scaninit(Item [], int *);
void show(Item [], int, int);
void sort(Item [], int, int);
```

mentation, we need to define the *array* and *item* data type interfaces precisely, then provide implementations.

The interface in Program 6.7 defines examples of high-level operations that we might want to perform on arrays. We want to be able to initialize an array with key values, either randomly or from the standard input; we want to be able to sort the entries (of course!); and we want to be able to print out the contents. These are but a few examples; in a particular application, we might want to define various other operations. With this interface, we can substitute different implementations of the various operations without having to change the client program that uses the interface—`main` in Program 6.6, in this case. The various sort implementations that we are studying can serve as implementations for the `sort` function. Program 6.8 has simple implementations for the other functions. Again, we might wish to substitute other implementations, depending on the application. For example, we might use an implementation of `show` that prints out only part of the array when testing sorts on huge arrays.

In a similar manner, to work with particular types of items and keys, we define their types and declare all the relevant operations on them in an explicit interface, then provide implementations of the operations defined in the *item* interface. Program 6.9 is an example of such an interface for floating point keys. This code defines the operations that we have been using to compare keys and to exchange items, as well as functions to generate a random key, to read a key from standard input, and to print out the value of a key. Program 6.10 has implementations of these functions for this simple example. Some of the operations are defined as macros in the interface, which approach

Program 6.8 Implementation of array data type

This code provides implementations of the functions defined in Program 6.7, again using the item types and basic functions for processing them that are defined in a separate interface (see Program 6.9).

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#include "Array.h"

void randinit(Item a[], int N)
{ int i;
    for (i = 0; i < N; i++) a[i] = ITEMrand();
}

void scaninit(Item a[], int *N)
{ int i = 0;
    for (i = 0; i < *N; i++)
        if (ITEMscan(&a[i]) == EOF) break;
    *N = i;
}

void show(itemType a[], int l, int r)
{ int i;
    for (i = l; i <= r; i++) ITEMshow(a[i]);
    printf("\n");
}
```

is generally more efficient; others are C code in the implementation, which approach is generally more flexible.

Programs 6.6 through 6.10 together with any of the sorting routines *as is* in Sections 6.2 through 6.6 provide a test of the sort for floating-point numbers. By providing similar interfaces and implementations for other types of data, we can put our sorts to use for a variety of data—such as long integers (see Exercise 6.49), complex numbers (see Exercise 6.50), or vectors (see Exercise 6.55)—without changing the sort code at all. For more complicated types of items, the interfaces and implementations have to be more complicated, but this implementation work is completely separated from the algorithm-design questions that we have been considering. We can use these same mechanisms with most of the sorting methods that we consider in this

Program 6.9 Sample interface for item data type

The file `Item.h` that is included in Programs 6.6 and 6.8 defines the data representation and associated operations for the items to be sorted. In this example, the items are floating-point keys. We use macros for the `key`, `less`, `exch`, and `compexch` data type operations for use by our sorting programs; we could also define them as functions to be implemented separately, like the three functions `ITEMrand` (return a random key), `ITEMscan` (read a key from standard input) and `ITEMshow` (print the value of a key).

```
typedef double Item;
#define key(A) (A)
#define less(A, B) (key(A) < key(B))
#define exch(A, B) { Item t = A; A = B; B = t; }
#define compexch(A, B) if (less(B, A)) exch(A, B)
Item ITEMrand(void);
int ITEMscan(Item *);
void ITEMshow(Item);
```

chapter and with those that we shall study in Chapters 7 through 9, as well. We consider in detail one important exception in Section 6.10—it leads to a whole family of important sorting algorithms that have to be packaged differently, the subject of Chapter 10.

The approach that we have discussed in this section is a middle road between Program 6.1 and an industrial-strength fully abstract set of implementations complete with error checking, memory management, and even more general capabilities. Packaging issues of this sort are of increasing importance in some modern programming and applications environments. We will necessarily leave some questions unanswered. Our primary purpose is to demonstrate, through the relatively simple mechanisms that we have examined, that the sorting implementations that we are studying are widely applicable.

Exercises

6.49 Write an interface and implementation for the generic item data type (similar to Programs 6.9 and 6.10) to support having the sorting methods sort long integers.

6.50 Write an interface and implementation for the generic item data type to support having the sorting methods sort complex numbers $x + iy$ using the

Program 6.10 Sample implementation for item data type

This code implements the three functions `ITEMrand`, `ITEMscan`, and `ITEMshow` that are declared in Program 6.9. In this code, we refer to the type of the data directly with `double`, use explicit floating-point options in `scanf` and `printf`, and so forth. We include the interface file `Item.h` so that we will discover at compile time any discrepancies between interface and implementation.

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
double ITEMrand(void)
{ return 1.0*rand()/RAND_MAX; }
int ITEMscan(double *x)
{ return scanf("%f", x); }
void ITEMshow(double x)
{ printf("%7.5f ", x); }
```

magnitude $\sqrt{x^2 + y^2}$ for the key. Note: Ignoring the square root is likely to improve efficiency.

- 6.51 Write an interface that defines a first-class *abstract* data type for generic items (see Section 4.8), and provide an implementation where the items are floating point numbers. Test your program with Programs 6.3 and 6.6.
- ▷ 6.52 Add a function `check` to the array data type in Programs 6.8 and 6.7, which tests whether or not the array is in sorted order.
- 6.53 Add a function `testinit` to the array data type in Programs 6.8 and 6.7, which generates test data according to distributions similar to those illustrated in Figure 6.13. Provide an integer argument for the client to use to specify the distribution.
- 6.54 Change Programs 6.7 and 6.8 to implement an *abstract* data type. (Your implementation should allocate and maintain the array, as in our implementations for stacks and queues in Chapter 3.)
- 6.55 Write an interface and implementation for the generic item data type for use in having the sorting methods sort multidimensional vectors of d integers, putting the vectors in order by first component, those with equal first component in order by second component, those with equal first and second components in order by third component, and so forth.

6.8 Index and Pointer Sorting

The development of a string data type implementation similar to Programs 6.9 and 6.10 is of particular interest, because character strings are widely used as sort keys. Using the C library string-comparison function, we can change the first three lines in Program 6.9 to

```
typedef char *Item;
#define key(A) (A)
#define less(A, B) (strcmp(key(A), key(B)) < 0)
```

to convert it to an interface for strings.

The implementation is more challenging than Program 6.10 because, when working with strings in C, we must be aware of the allocation of memory for the strings. Program 6.11 uses the method that we examined in Chapter 3 (Program 3.17), maintaining a buffer in the data-type implementation. Other options are to allocate mem-

Program 6.11 Data-type implementation for string items

This implementation allows us to use our sorting programs to sort strings. A string is a pointer to a character, so a sort will process an array of pointers to characters, rearranging them so the indicated strings are in alphanumeric order. We statically allocate the storage buffer containing the string characters in this module; dynamic allocation is perhaps more appropriate. The ITEM_{def} and implementation is omitted.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Item.h"
static char buf[100000];
static int cnt = 0;
int ITEMscan(char **x)
{ int t;
  *x = &buf[cnt];
  t = scanf("%s", *x); cnt += strlen(*x)+1;
  return t;
}
void ITEMshow(char *x)
{ printf("%s ", x); }
```

ory dynamically for each string, or to keep the buffer in the client program. We can use this code (along with the interface described in the previous paragraph) to sort strings of characters, using any of the sort implementations that we have been considering. Because strings are represented as pointers to arrays of characters in C, this program is an example of a *pointer sort*, which we shall consider shortly.

We are faced with memory-management choices of this kind any time that we modularize a program. Who should be responsible for managing the memory corresponding to the concrete realization of some type of object: the client, the data-type implementation, or the system? There is no hard-and-fast answer to this question (some programming-language designers become evangelical when the question is raised). Some modern programming systems (not C) have general mechanisms for dealing with memory management automatically. We will revisit this issue in Chapter 9, when we discuss the implementation of a more sophisticated abstract data type.

One simple approach for sorting without (intermediate) moves of items is to maintain an *index array* with keys in the items accessed only for comparisons. Suppose that the items to be sorted are in an array $\text{data}[0], \dots, \text{data}[N-1]$, and that we do not wish to move them around, for some reason (perhaps they are huge). To get the effect of sorting, we use a *second* array a of item indices. We begin by initializing $a[i]$ to i for $i = 0, \dots, N-1$. That is, we begin with $a[0]$ having the index of the first data item, $a[1]$ having the index of the second data item, and so on. The goal of the sort is to rearrange the index array a such that $a[0]$ gives the index of the data item with the smallest key, $a[1]$ gives the index of the data item with the second smallest key, and so on. Then we can achieve the effect of sorting by accessing the keys through the indices—for example, we could print out the array in sorted order in this way.

Now, we take advantage of the fact that our sort routines access data only through `less` and `exch`. In the item-type interface definition, we specify the type of the items to be sorted to be integers (the indices in a) with `typedef int Item;` and leave the exchange as before, but we change `less` to refer to the data through the indices:

```
#define less(A, B) (data[A] < data[B]).
```

For simplicity, this discussion assumes that the data are keys, rather than full items. We can use the same principle for larger, more compli-

cated items, by modifying `less` to access specific keys in the items. The sort routines rearrange the indices in `a`, which carry the information that we need to access the keys. An example of this arrangement, with the same items sorted by two different keys, is shown in Figure 6.14.

This index-array approach to indirection will work in any programming language that supports arrays. Another possibility, especially attractive in C, is to use pointers. For example, defining the data type

```
typedef dataType *Item;
```

and then initializing `a` with

```
for (i = 0; i < N; i++) a[i] = &data[i];
```

and doing comparisons indirectly with

```
#define less(A, B)  (*A < *B)
```

is equivalent to using the strategy described in the preceding paragraph. This arrangement is known as a *pointer sort*. The string data-type implementation that we just considered (Program 6.11) is an example of a pointer sort. For sorting an array of fixed-size items, a pointer sort is essentially equivalent to an index sort, but with the address of the array added to each index. But a pointer sort is much more general, because the pointers could point anywhere, and the items being sorted do not need to be fixed in size. As is true in index sorting, if `a` is an array of pointers to keys, then a call to `sort` will result in the pointers being rearranged such that accessing them sequentially will access the keys in order. We implement comparisons by following pointers; we implement exchanges by exchanging the pointers.

The standard C library sort function `qsort` is a pointer sort (see Program 3.17). The function takes four arguments: the array; the number of items to be sorted; the size of the items; and a pointer to a function that compares two items, given pointers to them. For example, if `Item` is `char*`, then the following code implements a string sort that adheres to our conventions:

```
int compare(void *i, void *j)
{ return strcmp(*(Item *)i, *(Item *)j); }
void sort(Item a[], int l, int r)
{ qsort(a, r-l+1, sizeof(Item), compare); }
```

The underlying algorithm is not specified in the interface, but quicksort (see Chapter 7) is widely used. In Chapter 7 we shall consider many of

0	10	9	Wilson	63
1	4	2	Johnson	86
2	5	1	Jones	87
3	6	0	Smith	90
4	8	4	Washington	84
5	7	8	Thompson	65
6	2	3	Brown	82
7	3	10	Jackson	61
8	9	6	White	76
9	0	5	Adams	86
10	1	7	Black	71

Figure 6.14
Index sorting example

By manipulating indices, rather than the records themselves, we can sort an array simultaneously on several keys. For this sample data that might represent students' names and grades, the second column is the result of an index sort on the name, and the third column is the result of an index sort on the grade. For example, Wilson is last in alphabetic order and has the tenth highest grade, while Adams is first in alphabetic order and has the sixth highest grade.

A rearrangement of the N distinct nonnegative integers less than N is called a *permutation* in mathematics: an index sort computes a permutation. In mathematics, permutations are normally defined as rearrangements of the integers 1 through N ; we shall use 0 through $N - 1$ to emphasize the direct relationship between permutations and C array indices.

Program 6.12 Data-type interface for record items

The records have two keys: a string key (for example, a name) in the first field, and an integer key (for example, a grade) in the second field. The comparison `less` is defined as a function, rather than as a macro, so we can change sort keys by changing implementations,

```
struct record { char name[30]; int num; };
typedef struct record* Item;
#define exch(A, B) { Item t = A; A = B; B = t; }
#define compexch(A, B) if (less(B, A)) exch(A, B);
int less(Item, Item);
Item ITEMrand();
int ITEMscan(Item *);
void ITEMshow(Item);
```

the reasons why this is true. We also, in this chapter and in Chapters 7 through 11, develop an understanding of why other methods might be more appropriate for some specific applications, and we explore approaches for speeding up the computation when the sort time is a critical factor in an application.

The primary reason to use indices or pointers is to avoid intruding on the data being sorted. We can “sort” a file even if read-only access is all that is available. Moreover, with multiple index or pointer arrays, we can sort one file on multiple keys (see Figure 6.14). This flexibility to manipulate the data without actually changing them is useful in many applications.

A second reason for manipulating indices is that we can avoid the cost of moving full records. The cost savings is significant for files with large records (and small keys), because the comparison needs to access just a small part of the record, and most of the record is not even touched during the sort. The indirect approach makes the cost of an exchange roughly equal to the cost of a comparison for general situations involving arbitrarily large records (at the cost of the extra space for the indices or pointers). Indeed, if the keys are long, the exchanges might even wind up being less costly than the comparisons. When we estimate the running times of methods that sort files of integers, we are often making the assumption that the costs of