2 Getting Started

This chapter will familiarize you with the framework we shall use throughout the book to think about the design and analysis of algorithms. It is self-contained, but it does include several references to material that will be introduced in Chapters 3 and 4. (It also contains several summations, which Appendix A shows how to solve.)

We begin by examining the insertion sort algorithm to solve the sorting problem introduced in Chapter 1. We define a "pseudocode" that should be familiar to readers who have done computer programming and use it to show how we shall specify our algorithms. Having specified the algorithm, we then argue that it correctly sorts and we analyze its running time. The analysis introduces a notation that focuses on how that time increases with the number of items to be sorted. Following our discussion of insertion sort, we introduce the divide-and-conquer approach to the design of algorithms and use it to develop an algorithm called merge sort. We end with an analysis of merge sort's running time.

2.1 Insertion sort

Our first algorithm, insertion sort, solves the ***sorting problem*** introduced in Chapter 1:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

The numbers that we wish to sort are also known as the ***keys***.

In this book, we shall typically describe algorithms as programs written in a ***pseudocode*** that is similar in many respects to C, Pascal, or Java. If you have been introduced to any of these languages, you should have little trouble reading our algorithms. What separates pseudocode from "real" code is that in pseudocode, we

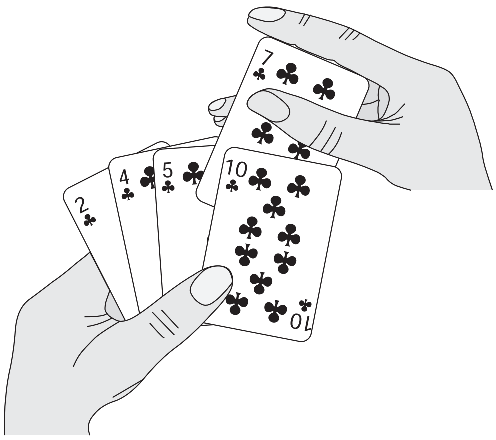


Figure 2.1 Sorting a hand of cards using insertion sort.

employ whatever expressive method is most clear and concise to specify a given algorithm. Sometimes, the clearest method is English, so do not be surprised if you come across an English phrase or sentence embedded within a section of "real" code. Another difference between pseudocode and real code is that pseudocode is not typically concerned with issues of software engineering. Issues of data abstraction, modularity, and error handling are often ignored in order to convey the essence of the algorithm more concisely.

We start with ***insertion sort***, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left, as illustrated in Figure 2.1. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

Our pseudocode for insertion sort is presented as a procedure called INSERTION-SORT, which takes as a parameter an array $A[1 \, . . \, n]$ containing a sequence of length n that is to be sorted. (In the code, the number n of elements in A is denoted by $length[A]$.) The input numbers are ***sorted in place***: the numbers are rearranged within the array A, with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when INSERTION-SORT is finished.

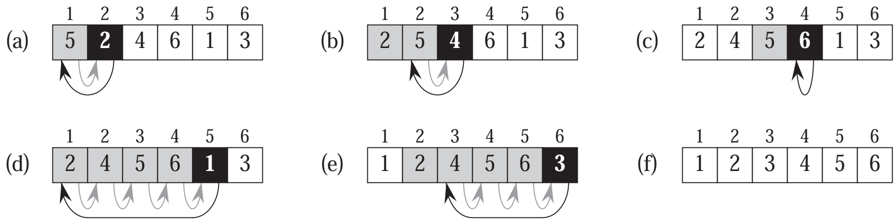


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key is moved to in line 8. **(f)** The final sorted array.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6              do A[i + 1] ← A[i]
7                 i ← i − 1
8          A[i + 1] ← key
```

Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. The index j indicates the "current card" being inserted into the hand. At the beginning of each iteration of the "outer" **for** loop, which is indexed by j, the subarray consisting of elements $A[1 \mathinner{..} j-1]$ constitute the currently sorted hand, and elements $A[j+1 \mathinner{..} n]$ correspond to the pile of cards still on the table. In fact, elements $A[1 \mathinner{..} j-1]$ are the elements *originally* in positions 1 through $j-1$, but now in sorted order. We state these properties of $A[1 \mathinner{..} j-1]$ formally as a ***loop invariant***:

> At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1 \mathinner{..} j-1]$ consists of the elements originally in $A[1 \mathinner{..} j-1]$ but in sorted order.

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

When the first two properties hold, the loop invariant is true prior to every iteration of the loop. Note the similarity to mathematical induction, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration is like the base case, and showing that the invariant holds from iteration to iteration is like the inductive step.

The third property is perhaps the most important one, since we are using the loop invariant to show correctness. It also differs from the usual use of mathematical induction, in which the inductive step is used infinitely; here, we stop the "induction" when the loop terminates.

Let us see how these properties hold for insertion sort.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$.[1] The subarray $A[1 \mathinner{..} j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the outer **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until the proper position for $A[j]$ is found (lines 4–7), at which point the value of $A[j]$ is inserted (line 8). A more formal treatment of the second property would require us to state and show a loop invariant for the "inner" **while** loop. At this point, however, we prefer not to get bogged down in such formalism, and so we rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine what happens when the loop terminates. For insertion sort, the outer **for** loop ends when j exceeds n, i.e., when $j = n + 1$. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1 \mathinner{..} n]$ consists of the elements originally in $A[1 \mathinner{..} n]$, but in sorted

---

[1] When the loop is a **for** loop, the moment at which we check the loop invariant just prior to the first iteration is immediately after the initial assignment to the loop-counter variable and just before the first test in the loop header. In the case of INSERTION-SORT, this time is after assigning 2 to the variable j but before the first test of whether $j \le \mathit{length}[A]$.

order. But the subarray $A[1 \mathinner{..} n]$ is the entire array! Hence, the entire array is sorted, which means that the algorithm is correct.

We shall use this method of loop invariants to show correctness later in this chapter and in other chapters as well.

Pseudocode conventions

We use the following conventions in our pseudocode.

1. Indentation indicates block structure. For example, the body of the **for** loop that begins on line 1 consists of lines 2–8, and the body of the **while** loop that begins on line 5 contains lines 6–7 but not line 8. Our indentation style applies to **if**-**then**-**else** statements as well. Using indentation instead of conventional indicators of block structure, such as **begin** and **end** statements, greatly reduces clutter while preserving, or even enhancing, clarity.[2]
2. The looping constructs **while**, **for**, and **repeat** and the conditional constructs **if**, **then**, and **else** have interpretations similar to those in Pascal.[3] There is one subtle difference with respect to **for** loops, however: in Pascal, the value of the loop-counter variable is undefined upon exiting the loop, but in this book, the loop counter retains its value after exiting the loop. Thus, immediately after a **for** loop, the loop counter's value is the value that first exceeded the **for** loop bound. We used this property in our correctness argument for insertion sort. The **for** loop header in line 1 is **for** $j \leftarrow 2$ **to** $length[A]$, and so when this loop terminates, $j = length[A]+1$ (or, equivalently, $j = n+1$, since $n = length[A]$).
3. The symbol "$\triangleright$" indicates that the remainder of the line is a comment.
4. A multiple assignment of the form $i \leftarrow j \leftarrow e$ assigns to both variables i and j the value of expression e; it should be treated as equivalent to the assignment $j \leftarrow e$ followed by the assignment $i \leftarrow j$.
5. Variables (such as i, j, and key) are local to the given procedure. We shall not use global variables without explicit indication.
6. Array elements are accessed by specifying the array name followed by the index in square brackets. For example, $A[i]$ indicates the ith element of the array A. The notation "$\mathinner{..}$" is used to indicate a range of values within an ar-

[2] In real programming languages, it is generally not advisable to use indentation alone to indicate block structure, since levels of indentation are hard to determine when code is split across pages.

[3] Most block-structured languages have equivalent constructs, though the exact syntax may differ from that of Pascal.

ray. Thus, $A[1\,..\,j]$ indicates the subarray of A consisting of the j elements $A[1], A[2], \ldots, A[j]$.

7. Compound data are typically organized into ***objects***, which are composed of ***attributes*** or ***fields***. A particular field is accessed using the field name followed by the name of its object in square brackets. For example, we treat an array as an object with the attribute *length* indicating how many elements it contains. To specify the number of elements in an array A, we write $length[A]$. Although we use square brackets for both array indexing and object attributes, it will usually be clear from the context which interpretation is intended.

   A variable representing an array or object is treated as a pointer to the data representing the array or object. For all fields f of an object x, setting $y \leftarrow x$ causes $f[y] = f[x]$. Moreover, if we now set $f[x] \leftarrow 3$, then afterward not only is $f[x] = 3$, but $f[y] = 3$ as well. In other words, x and y point to ("are") the same object after the assignment $y \leftarrow x$.

   Sometimes, a pointer will refer to no object at all. In this case, we give it the special value NIL.

8. Parameters are passed to a procedure ***by value***: the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is *not* seen by the calling procedure. When objects are passed, the pointer to the data representing the object is copied, but the object's fields are not. For example, if x is a parameter of a called procedure, the assignment $x \leftarrow y$ within the called procedure is not visible to the calling procedure. The assignment $f[x] \leftarrow 3$, however, is visible.

9. The boolean operators "and" and "or" are ***short circuiting***. That is, when we evaluate the expression "x and y" we first evaluate x. If x evaluates to FALSE, then the entire expression cannot evaluate to TRUE, and so we do not evaluate y. If, on the other hand, x evaluates to TRUE, we must evaluate y to determine the value of the entire expression. Similarly, in the expression "x or y" we evaluate the expression y only if x evaluates to FALSE. Short-circuiting operators allow us to write boolean expressions such as "$x \neq$ NIL and $f[x] = y$" without worrying about what happens when we try to evaluate $f[x]$ when x is NIL.

Exercises

2.1-1

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2
Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.

2.1-3
Consider the ***searching problem***:

Input: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value v.

Output: An index i such that $v = A[i]$ or the special value NIL if v does not appear in A.

Write pseudocode for ***linear search***, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

2.1-4
Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $(n + 1)$-element array C. State the problem formally and write pseudocode for adding the two integers.

2.2 Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, a most efficient one can be easily identified. Such analysis may indicate more than one viable candidate, but several inferior algorithms are usually discarded in the process.

Before we can analyze an algorithm, we must have a model of the implementation technology that will be used, including a model for the resources of that technology and their costs. For most of this book, we shall assume a generic one-processor, ***random-access machine*** (***RAM***) model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations. In later chapters, however, we shall have occasion to investigate models for digital hardware.

Strictly speaking, one should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then we could sort in just one instruction. Such a RAM would be unrealistic, since real computers do not have such instructions. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time.

The data types in the RAM model are integer and floating point. Although we typically do not concern ourselves with precision in this book, in some applications precision is crucial. We also assume a limit on the size of each word of data. For example, when working with inputs of size n, we typically assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. We require $c \geq 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily. (If the word size could grow arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time—clearly an unrealistic scenario.)

Real computers contain instructions not listed above, and such instructions represent a gray area in the RAM model. For example, is exponentiation a constant-time instruction? In the general case, no; it takes several instructions to compute x^y when x and y are real numbers. In restricted situations, however, exponentiation is a constant-time operation. Many computers have a "shift left" instruction, which in constant time shifts the bits of an integer by k positions to the left. In most computers, shifting the bits of an integer by one position to the left is equivalent to multiplication by 2. Shifting the bits by k positions to the left is equivalent to multiplication by 2^k. Therefore, such computers can compute 2^k in one constant-time instruction by shifting the integer 1 by k positions to the left, as long as k is no more than the number of bits in a computer word. We will endeavor to avoid such gray areas in the RAM model, but we will treat computation of 2^k as a constant-time operation when k is a small enough positive integer.

In the RAM model, we do not attempt to model the memory hierarchy that is common in contemporary computers. That is, we do not model caches or virtual memory (which is most often implemented with demand paging). Several computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. A handful of problems in this book examine memory-hierarchy effects, but for the most part, the analyses in this book will not consider them. Models that include the memory hierarchy are quite a bit more complex than the RAM model, so that they can be difficult to work with. Moreover, RAM-model analyses are usually excellent predictors of performance on actual machines.

Analyzing even a simple algorithm in the RAM model can be a challenge. The mathematical tools required may include combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because the behavior of an algorithm may be different for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Even though we typically select only one machine model to analyze a given algorithm, we still face many choices in deciding how to express our analysis. We would like a way that is simple to write and manipulate, shows the important characteristics of an algorithm's resource requirements, and suppresses tedious details.

Analysis of insertion sort

The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers. Moreover, INSERTION-SORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are. In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. To do so, we need to define the terms "running time" and "size of input" more carefully.

The best notion for ***input size*** depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the *number of items in the input*—for example, the array size n for sorting. For many other problems, such as multiplying two integers, the best measure of input size is the *total number of bits* needed to represent the input in ordinary binary notation. Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. We shall indicate which input size measure is being used with each problem we study.

The ***running time*** of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time c_i, where c_i is a constant. This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.[4]

---

[4]There are some subtleties here. Computational steps that we specify in English are often variants of a procedure that requires more than just a constant amount of time. For example, later in this book we might say "sort the points by x-coordinate," which, as we shall see, takes more than a constant amount of time. Also, note that a statement that calls a subroutine takes constant time, though the subroutine, once invoked, may take more. That is, we separate the process of ***calling*** the subroutine—passing parameters to it, etc.—from the process of ***executing*** the subroutine.

In the following discussion, our expression for the running time of INSERTION-SORT will evolve from a messy formula that uses all the statement costs c_i to a much simpler notation that is more concise and more easily manipulated. This simpler notation will also make it easy to determine whether one algorithm is more efficient than another.

We start by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed. For each $j = 2, 3, \ldots, n$, where $n = \mathit{length}[A]$, we let t_j be the number of times the **while** loop test in line 5 is executed for that value of j. When a **for** or **while** loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j \leftarrow 2$ **to** $\mathit{length}[A]$	c_1	n
2 **do** $\mathit{key} \leftarrow A[j]$	c_2	$n-1$
3 $\triangleright$ Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > \mathit{key}$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow \mathit{key}$	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and is executed n times will contribute $c_i n$ to the total running time.[5] To compute $T(n)$, the running time of INSERTION-SORT, we sum the products of the *cost* and *times* columns, obtaining

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \,. \end{aligned}$$

Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. For example, in INSERTION-SORT, the best

[5] This characteristic does not necessarily hold for a resource such as memory. A statement that references m words of memory and is executed n times does not necessarily consume mn words of memory in total.

case occurs if the array is already sorted. For each $j = 2, 3, \ldots, n$, we then find that $A[i] \leq key$ in line 5 when i has its initial value of $j - 1$. Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

This running time can be expressed as $an + b$ for *constants* a and b that depend on the statement costs c_i; it is thus a ***linear function*** of n.

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 \,..\, j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots, n$. Noting that

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(see Appendix A for a review of how to solve these summations), we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

This worst-case running time can be expressed as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i; it is thus a ***quadratic function*** of n.

Typically, as in insertion sort, the running time of an algorithm is fixed for a given input, although in later chapters we shall see some interesting "randomized" algorithms whose behavior can vary even for a fixed input.

Worst-case and average-case analysis

In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. For the remainder of this book, though, we shall usually concentrate on finding only the ***worst-case running time***, that is, the longest running time for *any* input of size n. We give three reasons for this orientation.

- The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.
- For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm's worst case will often occur when the information is not present in the database. In some searching applications, searches for absent information may be frequent.
- The "average case" is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray $A[1 \mathinner{..} j-1]$ to insert element $A[j]$? On average, half the elements in $A[1 \mathinner{..} j-1]$ are less than $A[j]$, and half the elements are greater. On average, therefore, we check half of the subarray $A[1 \mathinner{..} j-1]$, so $t_j = j/2$. If we work out the resulting average-case running time, it turns out to be a quadratic function of the input size, just like the worst-case running time.

In some particular cases, we shall be interested in the ***average-case*** or ***expected*** running time of an algorithm; in Chapter 5, we shall see the technique of ***probabilistic analysis***, by which we determine expected running times. One problem with performing an average-case analysis, however, is that it may not be apparent what constitutes an "average" input for a particular problem. Often, we shall assume that all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a ***randomized algorithm***, which makes random choices, to allow a probabilistic analysis.

Order of growth

We used some simplifying abstractions to ease our analysis of the INSERTION-SORT procedure. First, we ignored the actual cost of each statement, using the constants c_i to represent these costs. Then, we observed that even these constants give us more detail than we really need: the worst-case running time is an^2+bn+c for some constants a, b, and c that depend on the statement costs c_i. We thus ignored not only the actual statement costs, but also the abstract costs c_i.

We shall now make one more simplifying abstraction. It is the ***rate of growth***, or ***order of growth***, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large n. We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. Thus, we write that insertion sort, for example, has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared"). We shall use Θ-notation informally in this chapter; it will be defined precisely in Chapter 3.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth. Due to constant factors and lower-order terms, this evaluation may be in error for small inputs. But for large enough inputs, a $\Theta(n^2)$ algorithm, for example, will run more quickly in the worst case than a $\Theta(n^3)$ algorithm.

Exercises

2.2-1
Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

2.2-2
Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

2.2-3
Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.

2.2-4
How can we modify almost any algorithm to have a good best-case running time?

2.3 Designing algorithms

There are many ways to design algorithms. Insertion sort uses an ***incremental*** approach: having sorted the subarray $A[1\,..\,j-1]$, we insert the single element $A[j]$ into its proper place, yielding the sorted subarray $A[1\,..\,j]$.

In this section, we examine an alternative design approach, known as "divide-and-conquer." We shall use divide-and-conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort. One advantage of divide-and-conquer algorithms is that their running times are often easily determined using techniques that will be introduced in Chapter 4.

2.3.1 The divide-and-conquer approach

Many useful algorithms are ***recursive*** in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a ***divide-and-conquer*** approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of subproblems.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, we use an auxiliary procedure MERGE(A, p, q, r), where A is an array and p, q, and r are indices numbering elements of the array such that $p \le q < r$. The procedure assumes that the subarrays $A[p \mathinner{..} q]$ and $A[q+1 \mathinner{..} r]$ are in sorted order. It ***merges*** them to form a single sorted subarray that replaces the current subarray $A[p \mathinner{..} r]$.

Our MERGE procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the number of elements being merged, and it works as follows. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are checking just two top cards. Since we perform at most n basic steps, merging takes $\Theta(n)$ time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. The idea is to put on the bottom of each pile a ***sentinel*** card, which contains a special value that we use to simplify our code. Here, we use ∞ as the sentinel value, so that whenever a card with ∞ is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14             then A[k] ← L[i]
15                  i ← i + 1
16             else A[k] ← R[j]
17                  j ← j + 1
```

In detail, the MERGE procedure works as follows. Line 1 computes the length n_1 of the subarray $A[p..q]$, and line 2 computes the length n_2 of the subarray $A[q+1..r]$. We create arrays L and R ("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively, in line 3. The **for** loop of lines 4–5 copies the subar-

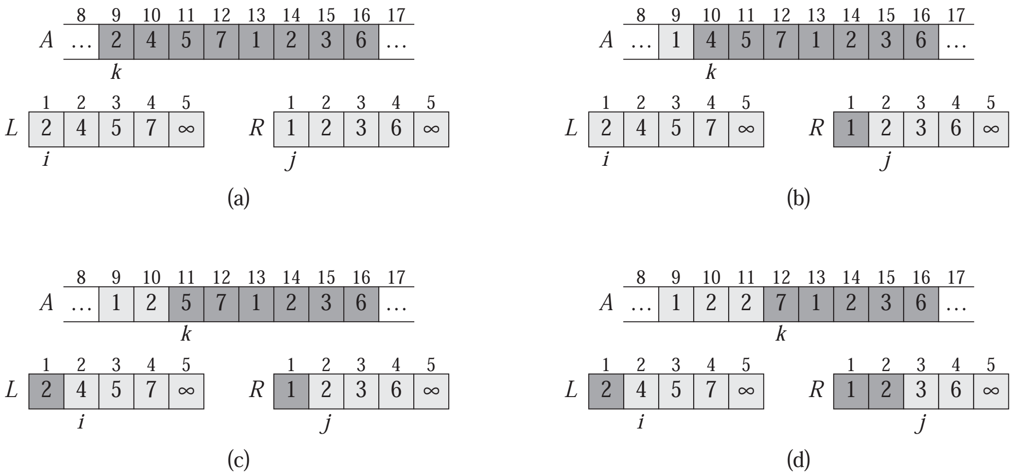


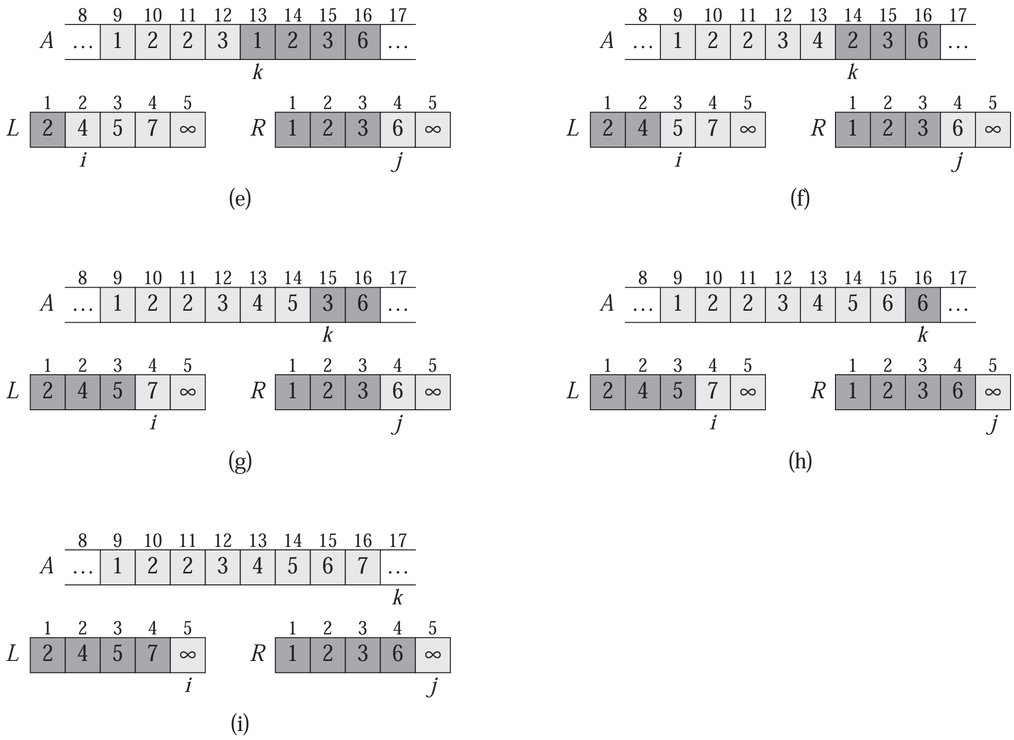
Figure 2.3 The operation of lines 10–17 in the call MERGE$(A, 9, 12, 16)$, when the subarray $A[9..16]$ contains the sequence $\langle 2, 4, 5, 7, 1, 2, 3, 6 \rangle$. After copying and inserting sentinels, the array L contains $\langle 2, 4, 5, 7, \infty \rangle$, and the array R contains $\langle 1, 2, 3, 6, \infty \rangle$. Lightly shaded positions in A contain their final values, and lightly shaded positions in L and R contain values that have yet to be copied back into A. Taken together, the lightly shaded positions always comprise the values originally in $A[9..16]$, along with the two sentinels. Heavily shaded positions in A contain values that will be copied over, and heavily shaded positions in L and R contain values that have already been copied back into A. **(a)–(h)** The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12–17. **(i)** The arrays and indices at termination. At this point, the subarray in $A[9..16]$ is sorted, and the two sentinels in L and R are the only two elements in these arrays that have not been copied into A.

ray $A[p..q]$ into $L[1..n_1]$, and the **for** loop of lines 6–7 copies the subarray $A[q+1..r]$ into $R[1..n_2]$. Lines 8–9 put the sentinels at the ends of the arrays L and R. Lines 10–17, illustrated in Figure 2.3, perform the $r - p + 1$ basic steps by maintaining the following loop invariant:

> At the start of each iteration of the **for** loop of lines 12–17, the subarray $A[p..k-1]$ contains the $k - p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

We must show that this loop invariant holds prior to the first iteration of the **for** loop of lines 12–17, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p..k-1]$ is empty. This empty subarray contains the $k - p = 0$



smallest elements of L and R, and since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \le R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p \mathinner{.\,.} k-1]$ contains the $k - p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p \mathinner{.\,.} k]$ will contain the $k - p + 1$ smallest elements. Incrementing k (in the **for** loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, then lines 16–17 perform the appropriate action to maintain the loop invariant.

Termination: At termination, $k = r + 1$. By the loop invariant, the subarray $A[p \mathinner{.\,.} k-1]$, which is $A[p \mathinner{.\,.} r]$, contains the $k - p = r - p + 1$ smallest elements of $L[1 \mathinner{.\,.} n_1 + 1]$ and $R[1 \mathinner{.\,.} n_2 + 1]$, in sorted order. The arrays L and R together contain $n_1 + n_2 + 2 = r - p + 3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

To see that the MERGE procedure runs in $\Theta(n)$ time, where $n = r - p + 1$, observe that each of lines 1–3 and 8–11 takes constant time, the **for** loops of lines 4–7 take $\Theta(n_1 + n_2) = \Theta(n)$ time,[6] and there are n iterations of the **for** loop of lines 12–17, each of which takes constant time.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT(A, p, r) sorts the elements in the subarray $A[p \mathinner{..} r]$. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p \mathinner{..} r]$ into two subarrays: $A[p \mathinner{..} q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1 \mathinner{..} r]$, containing $\lfloor n/2 \rfloor$ elements.[7]

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

To sort the entire sequence $A = \langle A[1], A[2], \ldots, A[n] \rangle$, we make the initial call MERGE-SORT$(A, 1, \mathit{length}[A])$, where once again $\mathit{length}[A] = n$. Figure 2.4 illustrates the operation of the procedure bottom-up when n is a power of 2. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n/2$ are merged to form the final sorted sequence of length n.

2.3.2 Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call to itself, its running time can often be described by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

A recurrence for the running time of a divide-and-conquer algorithm is based on the three steps of the basic paradigm. As before, we let $T(n)$ be the running time on a problem of size n. If the problem size is small enough, say $n \leq c$

---

[6] We shall see in Chapter 3 how to formally interpret equations containing Θ-notation.

[7] The expression $\lceil x \rceil$ denotes the least integer greater than or equal to x, and $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x. These notations are defined in Chapter 3. The easiest way to verify that setting q to $\lfloor (p+r)/2 \rfloor$ yields subarrays $A[p \mathinner{..} q]$ and $A[q+1 \mathinner{..} r]$ of sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively, is to examine the four cases that arise depending on whether each of p and r is odd or even.

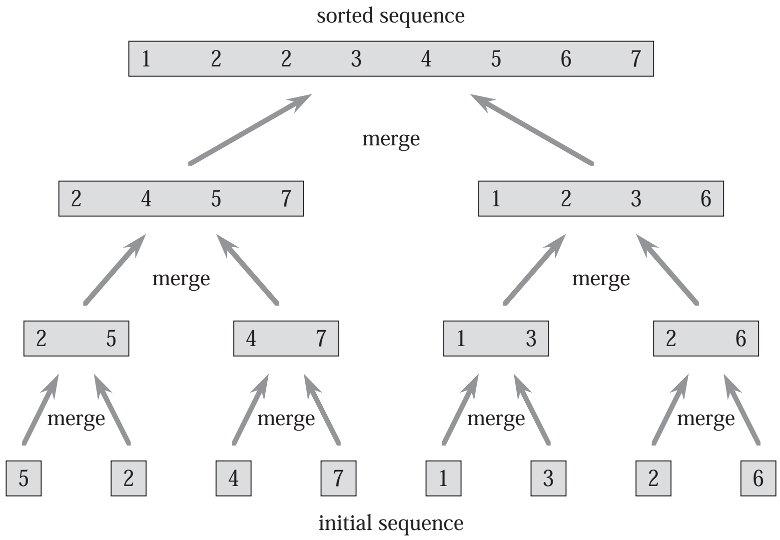


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

for some constant c, the straightforward solution takes constant time, which we write as $\Theta(1)$. Suppose that our division of the problem yields a subproblems, each of which is $1/b$ the size of the original. (For merge sort, both a and b are 2, but we shall see many divide-and-conquer algorithms in which $a \neq b$.) If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c , \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

In Chapter 4, we shall see how to solve common recurrences of this form.

Analysis of merge sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly $n/2$. In Chapter 4, we shall see that this assumption does not affect the order of growth of the solution to the recurrence.

We reason as follows to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine: We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, so $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases} \tag{2.1}$$

In Chapter 4, we shall see the "master theorem," which we can use to show that $T(n)$ is $\Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its $\Theta(n \lg n)$ running time, outperforms insertion sort, whose running time is $\Theta(n^2)$, in the worst case.

We do not need the master theorem to intuitively understand why the solution to the recurrence (2.1) is $T(n) = \Theta(n \lg n)$. Let us rewrite recurrence (2.1) as

$$T(n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases} \tag{2.2}$$

where the constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps.[8]

Figure 2.5 shows how we can solve the recurrence (2.2). For convenience, we assume that n is an exact power of 2. Part (a) of the figure shows $T(n)$, which in part (b) has been expanded into an equivalent tree representing the recurrence. The cn term is the root (the cost at the top level of recursion), and the two subtrees

[8]It is unlikely that the same constant exactly represents both the time to solve problems of size 1 and the time per array element of the divide and combine steps. We can get around this problem by letting c be the larger of these times and understanding that our recurrence gives an upper bound on the running time, or by letting c be the lesser of these times and understanding that our recurrence gives a lower bound on the running time. Both bounds will be on the order of $n \lg n$ and, taken together, give a $\Theta(n \lg n)$ running time.

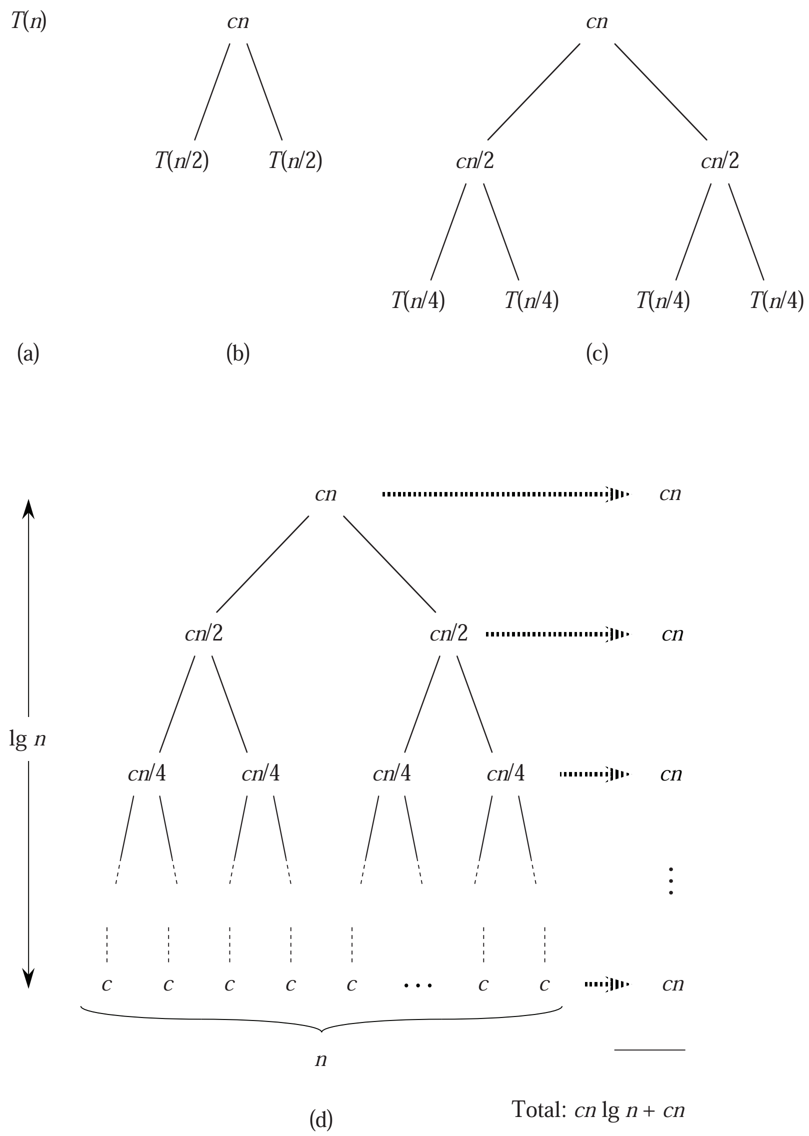


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

of the root are the two smaller recurrences $T(n/2)$. Part (c) shows this process carried one step further by expanding $T(n/2)$. The cost for each of the two subnodes at the second level of recursion is $cn/2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of c. Part (d) shows the resulting tree.

Next, we add the costs across each level of the tree. The top level has total cost cn, the next level down has total cost $c(n/2) + c(n/2) = cn$, the level after that has total cost $c(n/4) + c(n/4) + c(n/4) + c(n/4) = cn$, and so on. In general, the level i below the top has 2^i nodes, each contributing a cost of $c(n/2^i)$, so that the ith level below the top has total cost $2^i\, c(n/2^i) = cn$. At the bottom level, there are n nodes, each contributing a cost of c, for a total cost of cn.

The total number of levels of the "recursion tree" in Figure 2.5 is $\lg n + 1$. This fact is easily seen by an informal inductive argument. The base case occurs when $n = 1$, in which case there is only one level. Since $\lg 1 = 0$, we have that $\lg n + 1$ gives the correct number of levels. Now assume as an inductive hypothesis that the number of levels of a recursion tree for 2^i nodes is $\lg 2^i + 1 = i + 1$ (since for any value of i, we have that $\lg 2^i = i$). Because we are assuming that the original input size is a power of 2, the next input size to consider is 2^{i+1}. A tree with 2^{i+1} nodes has one more level than a tree of 2^i nodes, and so the total number of levels is $(i + 1) + 1 = \lg 2^{i+1} + 1$.

To compute the total cost represented by the recurrence (2.2), we simply add up the costs of all the levels. There are $\lg n + 1$ levels, each costing cn, for a total cost of $cn(\lg n + 1) = cn \lg n + cn$. Ignoring the low-order term and the constant c gives the desired result of $\Theta(n \lg n)$.

Exercises

2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

2.3-2

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

2.3-3

Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2 , \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

2.3-4

Insertion sort can be expressed as a recursive procedure as follows. In order to sort $A[1 \mathinner{..} n]$, we recursively sort $A[1 \mathinner{..} n-1]$ and then insert $A[n]$ into the sorted array $A[1 \mathinner{..} n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

2.3-5

Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. ***Binary search*** is an algorithm that repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

2.3-6

Observe that the **while** loop of lines 5 – 7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1 \mathinner{..} j-1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

2.3-7 ★

Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Problems

2-1 Insertion sort on small arrays in merge sort

Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort make it faster for small n. Thus, it makes sense to use insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

a. Show that the n/k sublists, each of length k, can be sorted by insertion sort in $\Theta(nk)$ worst-case time.

b. Show that the sublists can be merged in $\Theta(n \lg(n/k))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest asymptotic (Θ-notation) value of k as a function of n for which the modified algorithm has the same asymptotic running time as standard merge sort?

d. How should k be chosen in practice?

2-2 Correctness of bubblesort

Bubblesort is a popular sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

BUBBLESORT(A)

```
1  for i ← 1 to length[A]
2      do for j ← length[A] downto i + 1
3             do if A[j] < A[j − 1]
4                   then exchange A[j] ↔ A[j − 1]
```

a. Let A' denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$$A'[1] \le A'[2] \le \cdots \le A'[n] , \tag{2.3}$$

where $n = \mathit{length}[A]$. What else must be proved to show that BUBBLESORT actually sorts?

The next two parts will prove inequality (2.3).

b. State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

d. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

2-3 Correctness of Horner's rule

The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots)) , \end{aligned}$$

given the coefficients $a_0, a_1, \ldots, a_n$ and a value for x:

```
1  y ← 0
2  i ← n
3  while i ≥ 0
4      do y ← a_i + x · y
5         i ← i − 1
```

a. What is the asymptotic running time of this code fragment for Horner's rule?

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

c. Prove that the following is a loop invariant for the **while** loop in lines 3 –5.

> At the start of each iteration of the **while** loop of lines 3–5,
>
> $$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k .$$

Interpret a summation with no terms as equaling 0. Your proof should follow the structure of the loop invariant proof presented in this chapter and should show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$.

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

2-4 Inversions

Let $A[1 \,..\, n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

b. What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

d. Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

Chapter notes

In 1968, Knuth published the first of three volumes with the general title *The Art of Computer Programming* [182, 183, 185]. The first volume ushered in the modern study of computer algorithms with a focus on the analysis of running time, and the full series remains an engaging and worthwhile reference for many of the topics presented here. According to Knuth, the word "algorithm" is derived from the name "al-Khowârizmî," a ninth-century Persian mathematician.

Aho, Hopcroft, and Ullman [5] advocated the asymptotic analysis of algorithms as a means of comparing relative performance. They also popularized the use of recurrence relations to describe the running times of recursive algorithms.

Knuth [185] provides an encyclopedic treatment of many sorting algorithms. His comparison of sorting algorithms (page 381) includes exact step-counting analyses, like the one we performed here for insertion sort. Knuth's discussion of insertion sort encompasses several variations of the algorithm. The most important of these is Shell's sort, introduced by D. L. Shell, which uses insertion sort on periodic subsequences of the input to produce a faster sorting algorithm.

Merge sort is also described by Knuth. He mentions that a mechanical collator capable of merging two decks of punched cards in a single pass was invented in 1938. J. von Neumann, one of the pioneers of computer science, apparently wrote a program for merge sort on the EDVAC computer in 1945.

The early history of proving programs correct is described by Gries [133], who credits P. Naur with the first article in this field. Gries attributes loop invariants to R. W. Floyd. The textbook by Mitchell [222] describes more recent progress in proving programs correct.

6 Heapsort

In this chapter, we introduce another sorting algorithm. Like merge sort, but unlike insertion sort, heapsort's running time is $O(n \lg n)$. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Heapsort also introduces another algorithm design technique: the use of a data structure, in this case one we call a "heap," to manage information during the execution of the algorithm. Not only is the heap data structure useful for heapsort, but it also makes an efficient priority queue. The heap data structure will reappear in algorithms in later chapters.

We note that the term "heap" was originally coined in the context of heapsort, but it has since come to refer to "garbage-collected storage," such as the programming languages Lisp and Java provide. Our heap data structure is *not* garbage-collected storage, and whenever we refer to heaps in this book, we shall mean the structure defined in this chapter.

6.1 Heaps

The ***(binary) heap*** data structure is an array object that can be viewed as a nearly complete binary tree (see Section B.5.3), as shown in Figure 6.1. Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: $\mathit{length}[A]$, which is the number of elements in the array, and $\mathit{heap\text{-}size}[A]$, the number of elements in the heap stored within array A. That is, although $A[1 \,.\,.\, \mathit{length}[A]]$ may contain valid numbers, no element past $A[\mathit{heap\text{-}size}[A]]$, where $\mathit{heap\text{-}size}[A] \le \mathit{length}[A]$, is an element of the heap.

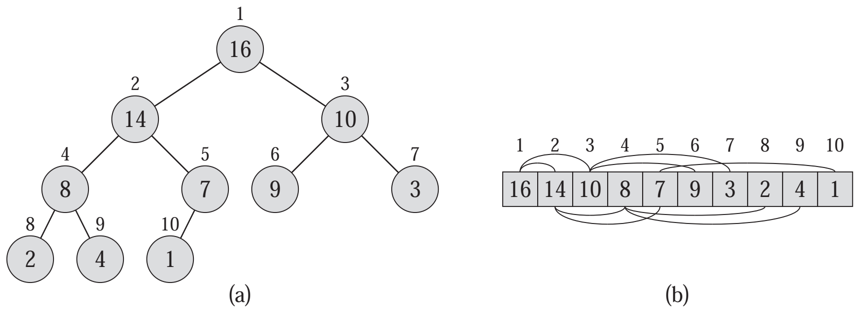


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

The root of the tree is $A[1]$, and given the index i of a node, the indices of its parent PARENT(i), left child LEFT(i), and right child RIGHT(i) can be computed simply:

PARENT(i)
    return $\lfloor i/2 \rfloor$

LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

On most computers, the LEFT procedure can compute $2i$ in one instruction by simply shifting the binary representation of i left one bit position. Similarly, the RIGHT procedure can quickly compute $2i+1$ by shifting the binary representation of i left one bit position and adding in a 1 as the low-order bit. The PARENT procedure can compute $\lfloor i/2 \rfloor$ by shifting i right one bit position. In a good implementation of heapsort, these three procedures are often implemented as "macros" or "in-line" procedures.

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a ***heap property***, the specifics of which depend on the kind of heap. In a ***max-heap***, the ***max-heap property*** is that for every node i other than the root,

$$A[\text{PARENT}(i)] \geq A[i] ,$$

that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself. A ***min-heap*** is organized in the opposite way; the ***min-heap property*** is that for every node i other than the root,

$$A[\text{PARENT}(i)] \le A[i] .$$

The smallest element in a min-heap is at the root.

For the heapsort algorithm, we use max-heaps. Min-heaps are commonly used in priority queues, which we discuss in Section 6.5. We shall be precise in specifying whether we need a max-heap or a min-heap for any particular application, and when properties apply to either max-heaps or min-heaps, we just use the term "heap."

Viewing a heap as a tree, we define the ***height*** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\Theta(\lg n)$ (see Exercise 6.1-2). We shall see that the basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(\lg n)$ time. The remainder of this chapter presents five basic procedures and shows how they are used in a sorting algorithm and a priority-queue data structure.

- The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM procedures, which run in $O(\lg n)$ time, allow the heap data structure to be used as a priority queue.

Exercises

6.1-1
What are the minimum and maximum numbers of elements in a heap of height h?

6.1-2
Show that an n-element heap has height $\lfloor \lg n \rfloor$.

6.1-3
Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

6.1-4
Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

6.1-5
Is an array that is in sorted order a min-heap?

6.1-6
Is the sequence $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a max-heap?

6.1-7
Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

6.2 Maintaining the heap property

MAX-HEAPIFY is an important subroutine for manipulating max-heaps. Its inputs are an array A and an index i into the array. When MAX-HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that $A[i]$ may be smaller than its children, thus violating the max-heap property. The function of MAX-HEAPIFY is to let the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i becomes a max-heap.

```
MAX-HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
   then largest ← l
   else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
   then largest ← r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
        MAX-HEAPIFY(A, largest)
```

Figure 6.2 illustrates the action of MAX-HEAPIFY. At each step, the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$ is determined, and its index is

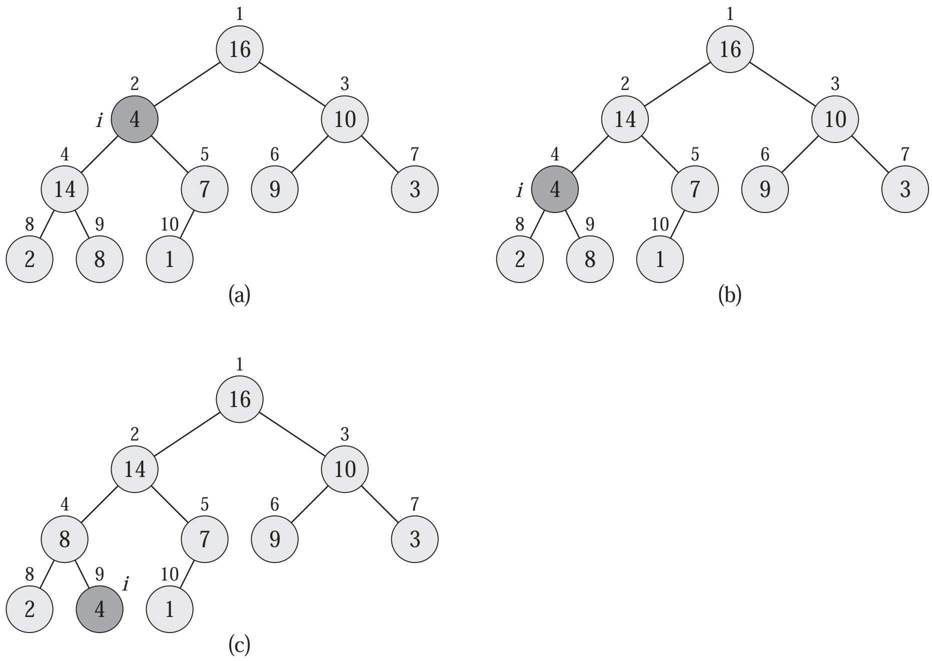


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where *heap-size*$[A] = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

stored in *largest*. If $A[i]$ is largest, then the subtree rooted at node i is a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and $A[i]$ is swapped with $A[\mathit{largest}]$, which causes node i and its children to satisfy the max-heap property. The node indexed by *largest*, however, now has the original value $A[i]$, and thus the subtree rooted at *largest* may violate the max-heap property. Consequently, MAX-HEAPIFY must be called recursively on that subtree.

The running time of MAX-HEAPIFY on a subtree of size n rooted at given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$, plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i. The children's subtrees each have size at most $2n/3$—the worst case occurs when the last row of the tree is exactly half full—and the running time of MAX-HEAPIFY can therefore be described by the recurrence

$$T(n) \le T(2n/3) + \Theta(1) .$$

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1), is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$.

Exercises

6.2-1
Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

6.2-2
Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

6.2-3
What is the effect of calling MAX-HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

6.2-4
What is the effect of calling MAX-HEAPIFY(A, i) for $i > \textit{heap-size}[A]/2$?

6.2-5
The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

6.2-6
Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (*Hint:* For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a path from the root down to a leaf.)

6.3 Building a heap

We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array $A[1 \,.\,.\, n]$, where $n = \textit{length}[A]$, into a max-heap. By Exercise 6.1-7, the elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \,..\, n]$ are all leaves of the tree, and so each is a 1-element heap to begin with. The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```

Figure 6.3 shows an example of the action of BUILD-MAX-HEAP.

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–3, each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2, \ldots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.

We can compute a simple upper bound on the running time of BUILD-MAX-HEAP as follows. Each call to MAX-HEAPIFY costs $O(\lg n)$ time, and there are $O(n)$ such calls. Thus, the running time is $O(n \lg n)$. This upper bound, though correct, is not asymptotically tight.

We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \lg n \rfloor$ (see Exercise 6.1-2) and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h (see Exercise 6.3-3).

The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, so we can express the total cost of BUILD-MAX-HEAP as

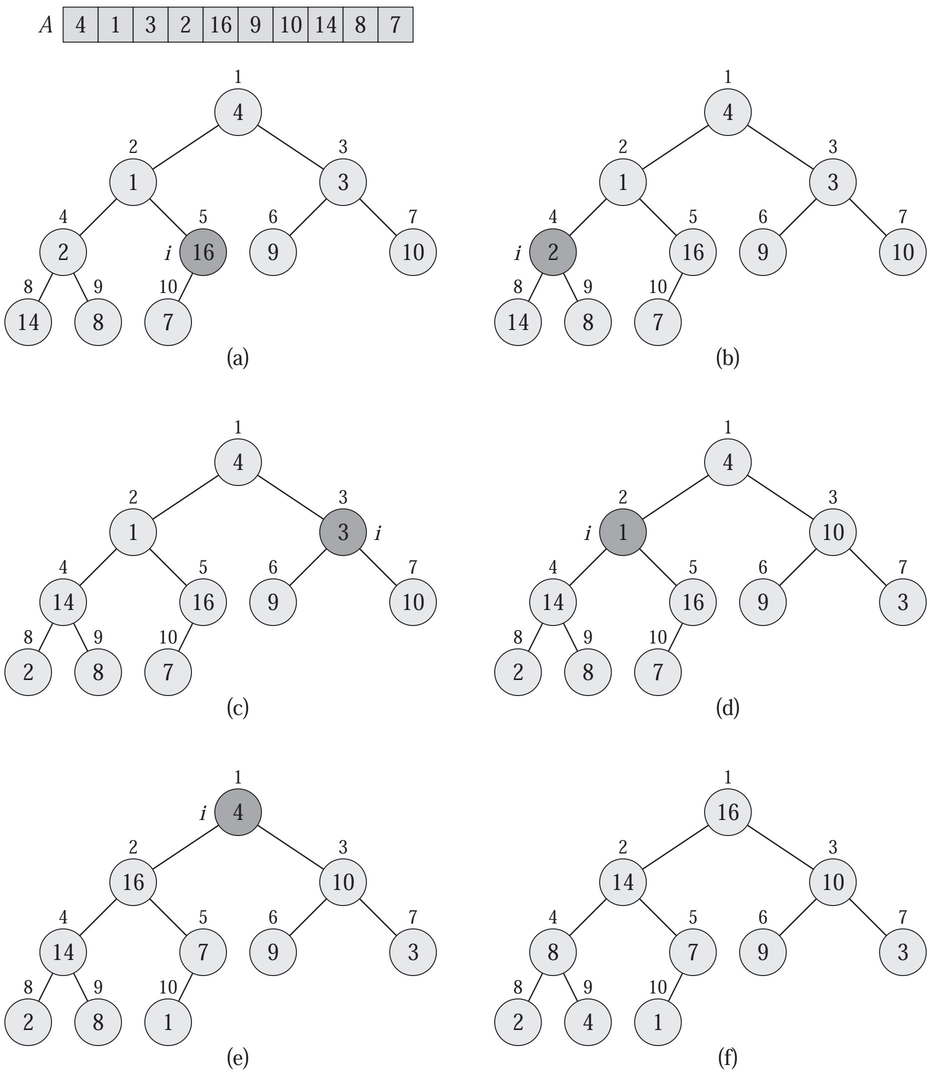


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) .$$

The last summation can be evaluated by substituting $x = 1/2$ in the formula (A.8), which yields

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

Thus, the running time of BUILD-MAX-HEAP can be bounded as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n) . \end{aligned}$$

Hence, we can build a max-heap from an unordered array in linear time.

We can build a min-heap by the procedure BUILD-MIN-HEAP, which is the same as BUILD-MAX-HEAP but with the call to MAX-HEAPIFY in line 3 replaced by a call to MIN-HEAPIFY (see Exercise 6.2-2). BUILD-MIN-HEAP produces a min-heap from an unordered linear array in linear time.

Exercises

6.3-1
Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2
Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor \mathit{length}[A]/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor \mathit{length}[A]/2 \rfloor$?

6.3-3
Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

6.4 The heapsort algorithm

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array $A[1 \, . . \, n]$, where $n = \mathit{length}[A]$. Since the maximum element of the array is stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$. If we now "discard" node n from the heap (by decrementing $heap\text{-}size[A]$), we observe that $A[1 \,..\, (n-1)]$ can easily be made into a max-heap. The children of the root remain max-heaps, but the new root element may violate the max-heap property. All that is needed to restore the max-heap property, however, is one call to MAX-HEAPIFY$(A, 1)$, which leaves a max-heap in $A[1 \,..\, (n-1)]$. The heapsort algorithm then repeats this process for the max-heap of size $n-1$ down to a heap of size 2. (See Exercise 6.4-2 for a precise loop invariant.)

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3       do exchange A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          MAX-HEAPIFY(A, 1)
```

Figure 6.4 shows an example of the operation of heapsort after the max-heap is initially built. Each max-heap is shown at the beginning of an iteration of the **for** loop of lines 2–5.

The HEAPSORT procedure takes time $O(n \lg n)$, since the call to BUILD-MAX-HEAP takes time $O(n)$ and each of the $n-1$ calls to MAX-HEAPIFY takes time $O(\lg n)$.

Exercises

6.4-1

Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4-2

Argue the correctness of HEAPSORT using the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–5, the subarray $A[1 \,..\, i]$ is a max-heap containing the i smallest elements of $A[1 \,..\, n]$, and the subarray $A[i+1 \,..\, n]$ contains the $n-i$ largest elements of $A[1 \,..\, n]$, sorted.

6.4-3

What is the running time of heapsort on an array A of length n that is already sorted in increasing order? What about decreasing order?

6.4-4

Show that the worst-case running time of heapsort is $\Omega(n \lg n)$.

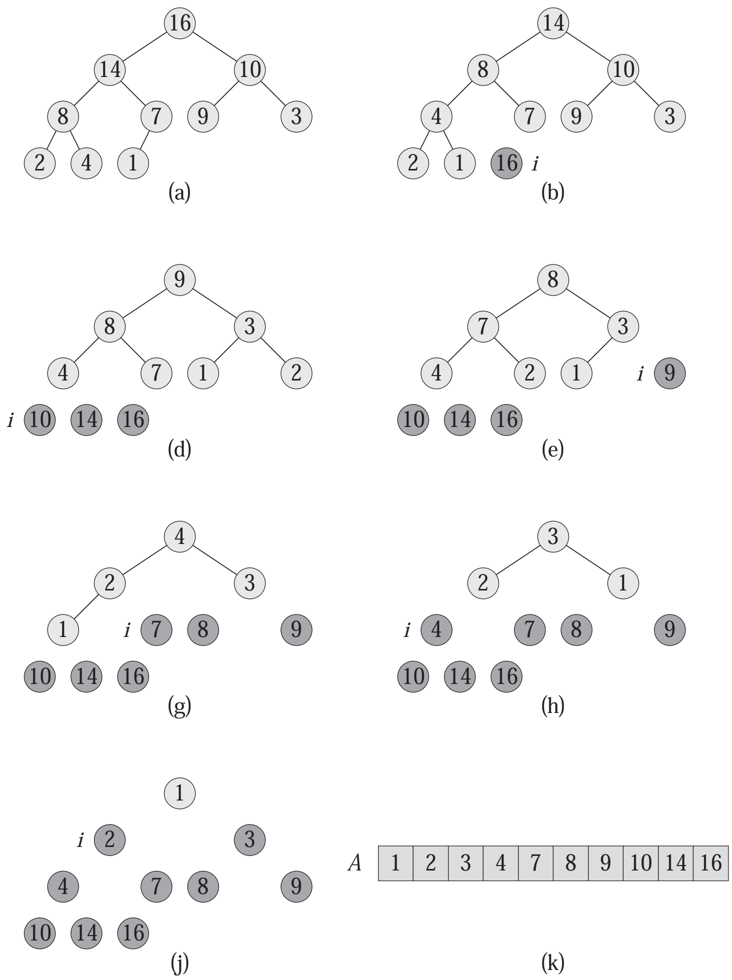


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after it has been built by BUILD-MAX-HEAP. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5. The value of i at that time is shown. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

6.4-5 ⋆
Show that when all elements are distinct, the best-case running time of heapsort is $\Omega(n \lg n)$.

6.5 Priority queues

Heapsort is an excellent algorithm, but a good implementation of quicksort, presented in Chapter 7, usually beats it in practice. Nevertheless, the heap data structure itself has enormous utility. In this section, we present one of the most popular applications of a heap: its use as an efficient priority queue. As with heaps, there are two kinds of priority queues: max-priority queues and min-priority queues. We will focus here on how to implement max-priority queues, which are in turn based on max-heaps; Exercise 6.5-3 asks you to write the procedures for min-priority queues.

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations.

INSERT(S, x) inserts the element x into the set S. This operation could be written as $S \leftarrow S \cup \{x\}$.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

One application of max-priority queues is to schedule jobs on a shared computer. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest-priority job is selected from those pending using EXTRACT-MAX. A new job can be added to the queue at any time using INSERT.

Alternatively, a ***min-priority queue*** supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY. A min-priority queue can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. The simulation program uses EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, they are inserted into the min-priority queue using INSERT. We shall see other uses for min-priority queues, highlighting the DECREASE-KEY operation, in Chapters 23 and 24.

Not surprisingly, we can use a heap to implement a priority queue. In a given application, such as job scheduling or event-driven simulation, elements of a priority queue correspond to objects in the application. It is often necessary to determine which application object corresponds to a given priority-queue element, and vice-versa. When a heap is used to implement a priority queue, therefore, we often need to store a ***handle*** to the corresponding application object in each heap element. The exact makeup of the handle (i.e., a pointer, an integer, etc.) depends on the application. Similarly, we need to store a handle to the corresponding heap element in each application object. Here, the handle would typically be an array index. Because heap elements change locations within the array during heap operations, an actual implementation, upon relocating a heap element, would also have to update the array index in the corresponding application object. Because the details of accessing application objects depend heavily on the application and its implementation, we shall not pursue them here, other than noting that in practice, these handles do need to be correctly maintained.

Now we discuss how to implement the operations of a max-priority queue. The procedure HEAP-MAXIMUM implements the MAXIMUM operation in $\Theta(1)$ time.

HEAP-MAXIMUM(A)

return $A[1]$

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation. It is similar to the **for** loop body (lines 3–5) of the HEAPSORT procedure.

HEAP-EXTRACT-MAX(A)

if $\mathit{heap\text{-}size}[A] < 1$
    then error "heap underflow"
$\mathit{max} \leftarrow A[1]$
$A[1] \leftarrow A[\mathit{heap\text{-}size}[A]]$
$\mathit{heap\text{-}size}[A] \leftarrow \mathit{heap\text{-}size}[A] - 1$
MAX-HEAPIFY$(A, 1)$
return *max*

The running time of HEAP-EXTRACT-MAX is $O(\lg n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for MAX-HEAPIFY.

The procedure HEAP-INCREASE-KEY implements the INCREASE-KEY operation. The priority-queue element whose key is to be increased is identified by an index i into the array. The procedure first updates the key of element $A[i]$ to its new value. Because increasing the key of $A[i]$ may violate the max-heap property, the procedure then, in a manner reminiscent of the insertion loop (lines 5–7) of INSERTION-SORT from Section 2.1, traverses a path from this node toward the root to find a proper place for the newly increased key. During this traversal, it repeatedly compares an element to its parent, exchanging their keys and continuing if the element's key is larger, and terminating if the element's key is smaller, since the max-heap property now holds. (See Exercise 6.5-5 for a precise loop invariant.)

HEAP-INCREASE-KEY(A, i, key)

1 **if** $key < A[i]$
2   **then error** "new key is smaller than current key"
3 $A[i] \leftarrow key$
4 **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5   **do** exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
6          $i \leftarrow \text{PARENT}(i)$

Figure 6.5 shows an example of a HEAP-INCREASE-KEY operation. The running time of HEAP-INCREASE-KEY on an n-element heap is $O(\lg n)$, since the path traced from the node updated in line 3 to the root has length $O(\lg n)$.

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes as an input the key of the new element to be inserted into max-heap A. The procedure first expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property.

MAX-HEAP-INSERT(A, key)

$heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
$A[heap\text{-}size[A]] \leftarrow -\infty$
HEAP-INCREASE-KEY$(A, heap\text{-}size[A], key)$

The running time of MAX-HEAP-INSERT on an n-element heap is $O(\lg n)$.

In summary, a heap can support any priority-queue operation on a set of size n in $O(\lg n)$ time.

Exercises

6.5-1
Illustrate the operation of HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-2
Illustrate the operation of MAX-HEAP-INSERT$(A, 10)$ on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$. Use the heap of Figure 6.5 as a model for the HEAP-INCREASE-KEY call.

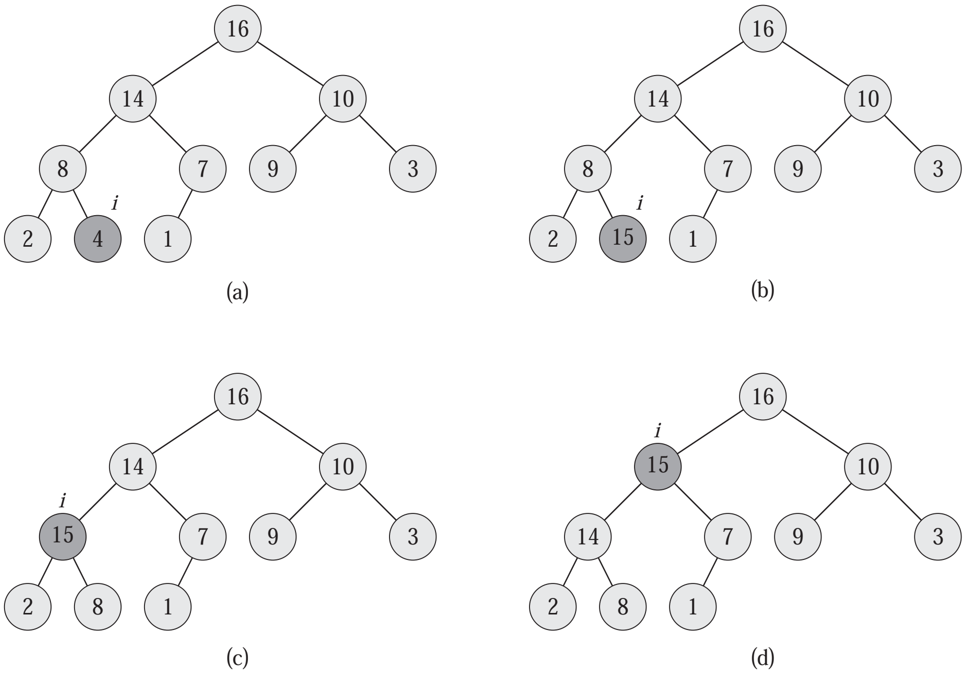


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

6.5-3

Write pseudocode for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.

6.5-4

Why do we bother setting the key of the inserted node to $-\infty$ in line 2 of MAX-HEAP-INSERT when the next thing we do is increase its key to the desired value?

6.5-5

Argue the correctness of HEAP-INCREASE-KEY using the following loop invariant:

> At the start of each iteration of the **while** loop of lines 4–6, the array $A[1 \,.\,.\, \mathit{heap\text{-}size}[A]]$ satisfies the max-heap property, except that there may be one violation: $A[i]$ may be larger than $A[\text{PARENT}(i)]$.

6.5-6

Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (Queues and stacks are defined in Section 10.1.)

6.5-7

The operation HEAP-DELETE(A, i) deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in $O(\lg n)$ time for an n-element max-heap.

6.5-8

Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (*Hint:* Use a min-heap for k-way merging.)

Problems

6-1 Building a heap using insertion

The procedure BUILD-MAX-HEAP in Section 6.3 can be implemented by repeatedly using MAX-HEAP-INSERT to insert the elements into the heap. Consider the following implementation:

```
BUILD-MAX-HEAP′(A)
heap-size[A] ← 1
for i ← 2 to length[A]
     do MAX-HEAP-INSERT(A, A[i])
```

a. Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

b. Show that in the worst case, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build an n-element heap.

6-2 Analysis of d-ary heaps

A ***d*-ary heap** is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.

a. How would you represent a d-ary heap in an array?

b. What is the height of a d-ary heap of n elements in terms of n and d?

c. Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

d. Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

e. Give an efficient implementation of INCREASE-KEY(A, i, k), which first sets $A[i] \leftarrow \max(A[i], k)$ and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.

6-3 Young tableaus

An $m \times n$ ***Young tableau*** is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus, a Young tableau can be used to hold $r \leq mn$ finite numbers.

a. Draw a 4×4 Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Argue that an $m \times n$ Young tableau Y is empty if $Y[1, 1] = \infty$. Argue that Y is full (contains mn elements) if $Y[m, n] < \infty$.

c. Give an algorithm to implement EXTRACT-MIN on a nonempty $m \times n$ Young tableau that runs in $O(m + n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m - 1) \times n$ or an $m \times (n - 1)$ subproblem. (*Hint:* Think about MAX-HEAPIFY.) Define $T(p)$, where $p = m + n$, to be the maximum running time of EXTRACT-MIN on any $m \times n$ Young tableau. Give and solve a recurrence for $T(p)$ that yields the $O(m + n)$ time bound.

d. Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m + n)$ time.

e. Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.

f. Give an $O(m+n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

Chapter notes

The heapsort algorithm was invented by Williams [316], who also described how to implement a priority queue with a heap. The BUILD-MAX-HEAP procedure was suggested by Floyd [90].

We use min-heaps to implement min-priority queues in Chapters 16, 23, and 24. We also give an implementation with improved time bounds for certain operations in Chapters 19 and 20.

Faster implementations of priority queues are possible for integer data. A data structure invented by van Emde Boas [301] supports the operations MINIMUM, MAXIMUM, INSERT, DELETE, SEARCH, EXTRACT-MIN, EXTRACT-MAX, PREDECESSOR, and SUCCESSOR in worst-case time $O(\lg \lg C)$, subject to the restriction that the universe of keys is the set $\{1, 2, \ldots, C\}$. If the data are b-bit integers, and the computer memory consists of addressable b-bit words, Fredman and Willard [99] showed how to implement MINIMUM in $O(1)$ time and INSERT and EXTRACT-MIN in $O(\sqrt{\lg n})$ time. Thorup [299] has improved the $O(\sqrt{\lg n})$ bound to $O((\lg \lg n)^2)$ time. This bound uses an amount of space unbounded in n, but it can be implemented in linear space by using randomized hashing.

An important special case of priority queues occurs when the sequence of EXTRACT-MIN operations is ***monotone***, that is, the values returned by successive EXTRACT-MIN operations are monotonically increasing over time. This case arises in several important applications, such as Dijkstra's single-source shortest-paths algorithm, which is discussed in Chapter 24, and in discrete-event simulation. For Dijkstra's algorithm it is particularly important that the DECREASE-KEY operation be implemented efficiently. For the monotone case, if the data are integers in the range $1, 2, \ldots, C$, Ahuja, Melhorn, Orlin, and Tarjan [8] describe how to implement EXTRACT-MIN and INSERT in $O(\lg C)$ amortized time (see Chapter 17 for more on amortized analysis) and DECREASE-KEY in $O(1)$ time, using a data structure called a radix heap. The $O(\lg C)$ bound can be improved to $O(\sqrt{\lg C})$ using Fibonacci heaps (see Chapter 20) in conjunction with radix heaps. The bound was further improved to $O(\lg^{1/3+\epsilon} C)$ expected time by Cherkassky, Goldberg, and Silverstein [58], who combine the multilevel bucketing structure of Denardo and Fox [72] with the heap of Thorup mentioned above. Raman [256] further improved these results to obtain a bound of $O(\min(\lg^{1/4+\epsilon} C, \lg^{1/3+\epsilon} n))$, for any fixed $\epsilon > 0$. More detailed discussions of these results can be found in papers by Raman [256] and Thorup [299].

7 Quicksort

Quicksort is a sorting algorithm whose worst-case running time is $\Theta(n^2)$ on an input array of n numbers. In spite of this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is $\Theta(n \lg n)$, and the constant factors hidden in the $\Theta(n \lg n)$ notation are quite small. It also has the advantage of sorting in place (see page 16), and it works well even in virtual memory environments.

Section 7.1 describes the algorithm and an important subroutine used by quicksort for partitioning. Because the behavior of quicksort is complex, we start with an intuitive discussion of its performance in Section 7.2 and postpone its precise analysis to the end of the chapter. Section 7.3 presents a version of quicksort that uses random sampling. This algorithm has a good average-case running time, and no particular input elicits its worst-case behavior. The randomized algorithm is analyzed in Section 7.4, where it is shown to run in $\Theta(n^2)$ time in the worst case and in $O(n \lg n)$ time on average.

7.1 Description of quicksort

Quicksort, like merge sort, is based on the divide-and-conquer paradigm introduced in Section 2.3.1. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p \mathinner{.\,.} r]$.

Divide: Partition (rearrange) the array $A[p \mathinner{.\,.} r]$ into two (possibly empty) subarrays $A[p \mathinner{.\,.} q-1]$ and $A[q+1 \mathinner{.\,.} r]$ such that each element of $A[p \mathinner{.\,.} q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1 \mathinner{.\,.} r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p \mathinner{.\,.} q-1]$ and $A[q+1 \mathinner{.\,.} r]$ by recursive calls to quicksort.

Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[p \mathinner{.\,.} r]$ is now sorted.

The following procedure implements quicksort.

QUICKSORT(A, p, r)

1 **if** $p < r$
2 **then** $q \leftarrow$ PARTITION(A, p, r)
3 QUICKSORT$(A, p, q - 1)$
4 QUICKSORT$(A, q + 1, r)$

To sort an entire array A, the initial call is QUICKSORT$(A, 1, \mathit{length}[A])$.

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \,.\,.\, r]$ in place.

PARTITION(A, p, r)

1 $x \leftarrow A[r]$
2 $i \leftarrow p - 1$
3 **for** $j \leftarrow p$ **to** $r - 1$
4 **do if** $A[j] \le x$
5 **then** $i \leftarrow i + 1$
6 exchange $A[i] \leftrightarrow A[j]$
7 exchange $A[i + 1] \leftrightarrow A[r]$
8 **return** $i + 1$

Figure 7.1 shows the operation of PARTITION on an 8-element array. PARTITION always selects an element $x = A[r]$ as a ***pivot*** element around which to partition the subarray $A[p \,.\,.\, r]$. As the procedure runs, the array is partitioned into four (possibly empty) regions. At the start of each iteration of the **for** loop in lines 3–6, each region satisfies certain properties, which we can state as a loop invariant:

> At the beginning of each iteration of the loop of lines 3–6, for any array index k,
>
> 1. If $p \le k \le i$, then $A[k] \le x$.
> 2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
> 3. If $k = r$, then $A[k] = x$.

Figure 7.2 summarizes this structure. The indices between j and $r - 1$ are not covered by any of the three cases, and the values in these entries have no particular relationship to the pivot x.

We need to show that this loop invariant is true prior to the first iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

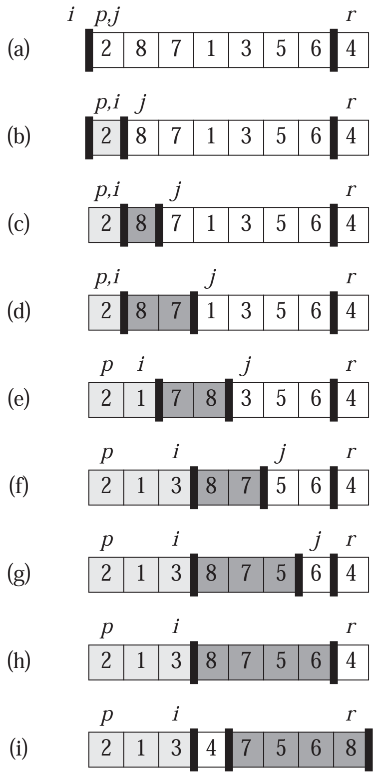


Figure 7.1 The operation of PARTITION on a sample array. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 8 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6 and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

Initialization: Prior to the first iteration of the loop, $i = p - 1$, and $j = p$. There are no values between p and i, and no values between $i + 1$ and $j - 1$, so the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

Maintenance: As Figure 7.3 shows, there are two cases to consider, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when

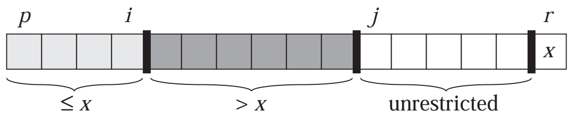


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p \,..\, r]$. The values in $A[p \,..\, i]$ are all less than or equal to x, the values in $A[i+1 \,..\, j-1]$ are all greater than x, and $A[r] = x$. The values in $A[j \,..\, r-1]$ can take on any values.

$A[j] > x$; the only action in the loop is to increment j. After j is incremented, condition 2 holds for $A[j-1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \leq x$; i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Because of the swap, we now have that $A[i] \leq x$, and condition 1 is satisfied. Similarly, we also have that $A[j-1] > x$, since the item that was swapped into $A[j-1]$ is, by the loop invariant, greater than x.

Termination: At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x.

The final two lines of PARTITION move the pivot element into its place in the middle of the array by swapping it with the leftmost element that is greater than x. The output of PARTITION now satisfies the specifications given for the divide step.

The running time of PARTITION on the subarray $A[p \,..\, r]$ is $\Theta(n)$, where $n = r - p + 1$ (see Exercise 7.1-3).

Exercises

7.1-1
Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$.

7.1-2
What value of q does PARTITION return when all elements in the array $A[p \,..\, r]$ have the same value? Modify PARTITION so that $q = (p+r)/2$ when all elements in the array $A[p \,..\, r]$ have the same value.

7.1-3
Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

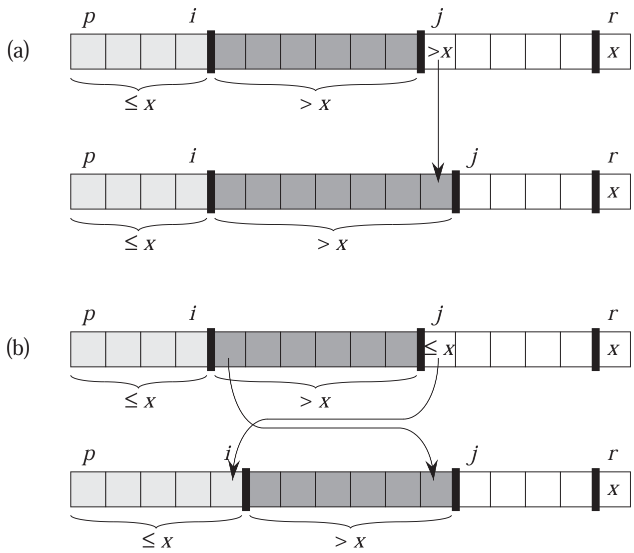


Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \le x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

7.1-4

How would you modify QUICKSORT to sort into nonincreasing order?

7.2 Performance of quicksort

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, and this in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort. In this section, we shall informally investigate how quicksort performs under the assumptions of balanced versus unbalanced partitioning.

Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n-1$ elements and one with 0 elements. (This claim is proved in Section 7.4.1.) Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\Theta(n)$ time. Since the recursive call on an array of size 0 just returns, $T(0) = \Theta(1)$, and the recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

Intuitively, if we sum the costs incurred at each level of the recursion, we get an arithmetic series (equation (A.2)), which evaluates to $\Theta(n^2)$. Indeed, it is straightforward to use the substitution method to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$. (See Exercise 7.2-1.)

Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. Therefore the worst-case running time of quicksort is no better than that of insertion sort. Moreover, the $\Theta(n^2)$ running time occurs when the input array is already completely sorted—a common situation in which insertion sort runs in $O(n)$ time.

Best-case partitioning

In the most even possible split, PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$. In this case, quicksort runs much faster. The recurrence for the running time is then

$$T(n) \le 2T(n/2) + \Theta(n) ,$$

which by case 2 of the master theorem (Theorem 4.1) has the solution $T(n) = O(n \lg n)$. Thus, the equal balancing of the two sides of the partition at every level of the recursion produces an asymptotically faster algorithm.

Balanced partitioning

The average-case running time of quicksort is much closer to the best case than to the worst case, as the analyses in Section 7.4 will show. The key to understanding why is to understand how the balance of the partitioning is reflected in the recurrence that describes the running time.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

$$T(n) \le T(9n/10) + T(n/10) + cn$$

on the running time of quicksort, where we have explicitly included the constant c hidden in the $\Theta(n)$ term. Figure 7.4 shows the recursion tree for this recurrence. Notice that every level of the tree has cost cn, until a boundary condition is reached at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost at most cn. The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$. The total cost of quicksort is

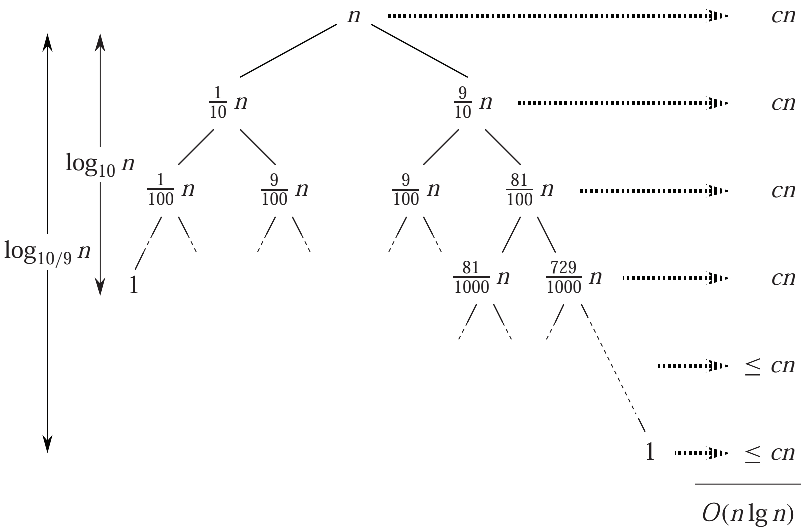


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

therefore $O(n \lg n)$. Thus, with a 9-to-1 proportional split at every level of recursion, which intuitively seems quite unbalanced, quicksort runs in $O(n \lg n)$ time—asymptotically the same as if the split were right down the middle. In fact, even a 99-to-1 split yields an $O(n \lg n)$ running time. The reason is that any split of *constant* proportionality yields a recursion tree of depth $\Theta(\lg n)$, where the cost at each level is $O(n)$. The running time is therefore $O(n \lg n)$ whenever the split has constant proportionality.

Intuition for the average case

To develop a clear notion of the average case for quicksort, we must make an assumption about how frequently we expect to encounter the various inputs. The behavior of quicksort is determined by the relative ordering of the values in the array elements given as the input, and not by the particular values in the array. As in our probabilistic analysis of the hiring problem in Section 5.2, we will assume for now that all permutations of the input numbers are equally likely.

When we run quicksort on a random input array, it is unlikely that the partitioning always happens in the same way at every level, as our informal analysis has assumed. We expect that some of the splits will be reasonably well balanced and that some will be fairly unbalanced. For example, Exercise 7.2-6 asks you to show

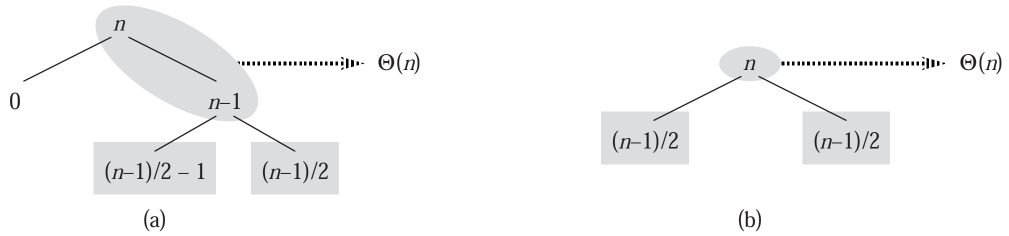


Figure 7.5 **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 0 and $n-1$. The partitioning of the subarray of size $n-1$ costs $n-1$ and produces a "good" split: subarrays of size $(n-1)/2-1$ and $(n-1)/2$. **(b)** A single level of a recursion tree that is very well balanced. In both parts, the partitioning cost for the subproblems shown with elliptical shading is $\Theta(n)$. Yet the subproblems remaining to be solved in (a), shown with square shading, are no larger than the corresponding subproblems remaining to be solved in (b).

that about 80 percent of the time PARTITION produces a split that is more balanced than 9 to 1, and about 20 percent of the time it produces a split that is less balanced than 9 to 1.

In the average case, PARTITION produces a mix of "good" and "bad" splits. In a recursion tree for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree. Suppose for the sake of intuition, however, that the good and bad splits alternate levels in the tree, and that the good splits are best-case splits and the bad splits are worst-case splits. Figure 7.5(a) shows the splits at two consecutive levels in the recursion tree. At the root of the tree, the cost is n for partitioning, and the subarrays produced have sizes $n-1$ and 0: the worst case. At the next level, the subarray of size $n-1$ is best-case partitioned into subarrays of size $(n-1)/2-1$ and $(n-1)/2$. Let's assume that the boundary-condition cost is 1 for the subarray of size 0.

The combination of the bad split followed by the good split produces three subarrays of sizes 0, $(n-1)/2-1$, and $(n-1)/2$ at a combined partitioning cost of $\Theta(n)+\Theta(n-1)=\Theta(n)$. Certainly, this situation is no worse than that in Figure 7.5(b), namely a single level of partitioning that produces two subarrays of size $(n-1)/2$, at a cost of $\Theta(n)$. Yet this latter situation is balanced! Intuitively, the $\Theta(n-1)$ cost of the bad split can be absorbed into the $\Theta(n)$ cost of the good split, and the resulting split is good. Thus, the running time of quicksort, when levels alternate between good and bad splits, is like the running time for good splits alone: still $O(n \lg n)$, but with a slightly larger constant hidden by the O-notation. We shall give a rigorous analysis of the average case in Section 7.4.2.

8 Sorting in Linear Time

We have now introduced several algorithms that can sort n numbers in $O(n \lg n)$ time. Merge sort and heapsort achieve this upper bound in the worst case; quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \lg n)$ time.

These algorithms share an interesting property: *the sorted order they determine is based only on comparisons between the input elements*. We call such sorting algorithms ***comparison sorts***. All the sorting algorithms introduced thus far are comparison sorts.

In Section 8.1, we shall prove that any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Sections 8.2, 8.3, and 8.4 examine three sorting algorithms—counting sort, radix sort, and bucket sort—that run in linear time. Needless to say, these algorithms use operations other than comparisons to determine the sorted order. Consequently, the $\Omega(n \lg n)$ lower bound does not apply to them.

8.1 Lower bounds for sorting

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, or $a_i > a_j$ to determine their relative order. We may not inspect the values of the elements or gain order information about them in any other way.

In this section, we assume without loss of generality that all of the input elements are distinct. Given this assumption, comparisons of the form $a_i = a_j$ are useless, so we can assume that no comparisons of this form are made. We also note that the comparisons $a_i \le a_j$, $a_i \ge a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent in that

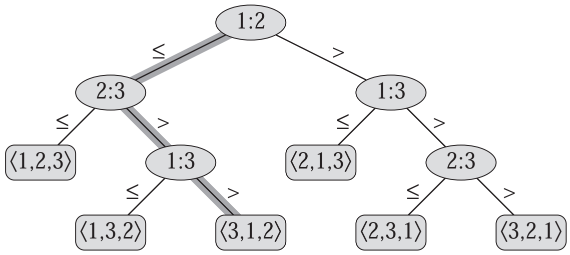


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i{:}j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, so the decision tree must have at least 6 leaves.

they yield identical information about the relative order of a_i and a_j. We therefore assume that all comparisons have the form $a_i \le a_j$.

The decision-tree model

Comparison sorts can be viewed abstractly in terms of ***decision trees***. A decision tree is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. Figure 8.1 shows the decision tree corresponding to the insertion sort algorithm from Section 2.1 operating on an input sequence of three elements.

In a decision tree, each internal node is annotated by $i{:}j$ for some i and j in the range $1 \le i, j \le n$, where n is the number of elements in the input sequence. Each leaf is annotated by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$. (See Section C.1 for background on permutations.) The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each internal node, a comparison $a_i \le a_j$ is made. The left subtree then dictates subsequent comparisons for $a_i \le a_j$, and the right subtree dictates subsequent comparisons for $a_i > a_j$. When we come to a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. Because any correct sorting algorithm must be able to produce each permutation of its input, a necessary condition for a comparison sort to be correct is that each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree, and that each of these leaves must be reachable from the root by a path corresponding to an actual execution of the comparison sort. (We shall refer to such leaves as "reachable.") Thus, we shall consider only decision trees in which each permutation appears as a reachable leaf.

A lower bound for the worst case

The length of the longest path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a lower bound.

Theorem 8.1
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \le l$. Since a binary tree of height h has no more than 2^h leaves, we have

$$n! \le l \le 2^h ,$$

which, by taking logarithms, implies

$$\begin{aligned} h &\ge \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.18))} . \end{aligned}$$ ■

Corollary 8.2
Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof The $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from Theorem 8.1. ■

Exercises

8.1-1
What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

8.1-2
Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.

8.4 Bucket sort

Bucket sort runs in linear time when the input is drawn from a uniform distribution. Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the interval $[0, 1)$. (See Section C.2 for a definition of uniform distribution.)

The idea of bucket sort is to divide the interval $[0, 1)$ into n equal-sized subintervals, or ***buckets***, and then distribute the n input numbers into the buckets. Since the inputs are uniformly distributed over $[0, 1)$, we don't expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

Our code for bucket sort assumes that the input is an n-element array A and that each element $A[i]$ in the array satisfies $0 \le A[i] < 1$. The code requires an auxiliary array $B[0 \,..\, n-1]$ of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists. (Section 10.2 describes how to implement basic operations on linked lists.)

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3       do insert A[i] into list B[⌊nA[i]⌋]
4  for i ← 0 to n − 1
5       do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Figure 8.4 shows the operation of bucket sort on an input array of 10 numbers.

To see that this algorithm works, consider two elements $A[i]$ and $A[j]$. Assume without loss of generality that $A[i] \le A[j]$. Since $\lfloor nA[i] \rfloor \le \lfloor nA[j] \rfloor$, element $A[i]$ is placed either into the same bucket as $A[j]$ or into a bucket with a lower index. If $A[i]$ and $A[j]$ are placed into the same bucket, then the **for** loop of lines 4–5 puts them into the proper order. If $A[i]$ and $A[j]$ are placed into different buckets, then line 6 puts them into the proper order. Therefore, bucket sort works correctly.

To analyze the running time, observe that all lines except line 5 take $O(n)$ time in the worst case. It remains to balance the total time taken by the n calls to insertion sort in line 5.

To analyze the cost of the calls to insertion sort, let n_i be the random variable denoting the number of elements placed in bucket $B[i]$. Since insertion sort runs in quadratic time (see Section 2.2), the running time of bucket sort is

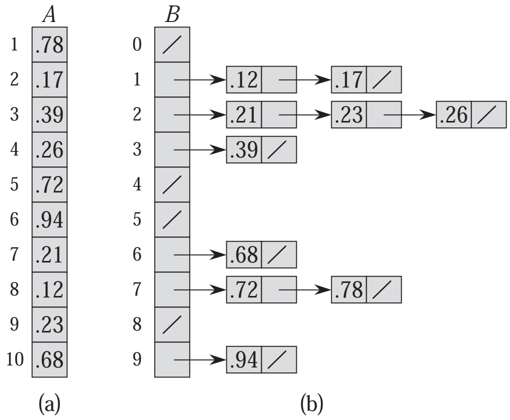


Figure 8.4 The operation of BUCKET-SORT. **(a)** The input array $A[1\,..\,10]$. **(b)** The array $B[0\,..\,9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \ .$$

Taking expectations of both sides and using linearity of expectation, we have

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] \qquad \text{(by linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) \qquad \text{(by equation (C.21))} \ . \qquad (8.1)
\end{aligned}$$

We claim that

$$\mathrm{E}[n_i^2] = 2 - 1/n \qquad (8.2)$$

for $i = 0, 1, \ldots, n-1$. It is no surprise that each bucket i has the same value of $\mathrm{E}[n_i^2]$, since each value in the input array A is equally likely to fall in any bucket. To prove equation (8.2), we define indicator random variables

$$X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$$

for $i = 0, 1, \ldots, n-1$ and $j = 1, 2, \ldots, n$. Thus,

$$n_i = \sum_{j=1}^{n} X_{ij} \; .$$

To compute $\mathrm{E}[n_i^2]$, we expand the square and regroup terms:

$$\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\;\sum_{\substack{1\le k\le n\\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n}\mathrm{E}\left[X_{ij}^2\right] + \sum_{1\le j\le n}\;\sum_{\substack{1\le k\le n\\ k\ne j}} \mathrm{E}[X_{ij}X_{ik}] \; ,
\end{aligned} \tag{8.3}$$

where the last line follows by linearity of expectation. We evaluate the two summations separately. Indicator random variable X_{ij} is 1 with probability $1/n$ and 0 otherwise, and therefore

$$\begin{aligned}
\mathrm{E}\left[X_{ij}^2\right] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\
&= \frac{1}{n} \; .
\end{aligned}$$

When $k \ne j$, the variables X_{ij} and X_{ik} are independent, and hence

$$\begin{aligned}
\mathrm{E}[X_{ij}X_{ik}] &= \mathrm{E}[X_{ij}]\,\mathrm{E}[X_{ik}] \\
&= \frac{1}{n} \cdot \frac{1}{n} \\
&= \frac{1}{n^2} \; .
\end{aligned}$$

Substituting these two expected values in equation (8.3), we obtain

$$\begin{aligned}
\mathrm{E}[n_i^2] &= \sum_{j=1}^{n}\frac{1}{n} + \sum_{1\le j\le n}\;\sum_{\substack{1\le k\le n\\ k\ne j}} \frac{1}{n^2} \\
&= n \cdot \frac{1}{n} + n(n-1)\cdot\frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 2 - \frac{1}{n} \; ,
\end{aligned}$$

which proves equation (8.2).

Using this expected value in equation (8.1), we conclude that the expected time for bucket sort is $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$. Thus, the entire bucket sort algorithm runs in linear expected time.

Even if the input is not drawn from a uniform distribution, bucket sort may still run in linear time. As long as the input has the property that the sum of the squares of the bucket sizes is linear in the total number of elements, equation (8.1) tells us that bucket sort will run in linear time.

Exercises

8.4-1

Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

8.4-2

What is the worst-case running time for the bucket-sort algorithm? What simple change to the algorithm preserves its linear expected running time and makes its worst-case running time $O(n \lg n)$?

8.4-3

Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $\mathrm{E}[X^2]$? What is $\mathrm{E}^2[X]$?

8.4-4 ⋆

We are given n points in the unit circle, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \leq 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed; that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Design a $\Theta(n)$ expected-time algorithm to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (*Hint:* Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit circle.)

8.4-5 ⋆

A ***probability distribution function*** $P(x)$ for a random variable X is defined by $P(x) = \Pr\{X \leq x\}$. Suppose that a list of n random variables $X_1, X_2, \ldots, X_n$ is drawn from a continuous probability distribution function P that is computable in $O(1)$ time. Show how to sort these numbers in linear expected time.

Problems

8-1 Average-case lower bounds on comparison sorting

In this problem, we prove an $\Omega(n \lg n)$ lower bound on the expected running time of any deterministic or randomized comparison sort on n distinct input elements. We begin by examining a deterministic comparison sort A with decision tree T_A. We assume that every permutation of A's inputs is equally likely.

a. Suppose that each leaf of T_A is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $1/n!$ and that the rest are labeled 0.

b. Let $D(T)$ denote the external path length of a decision tree T; that is, $D(T)$ is the sum of the depths of all the leaves of T. Let T be a decision tree with $k > 1$ leaves, and let LT and RT be the left and right subtrees of T. Show that $D(T) = D(LT) + D(RT) + k$.

c. Let $d(k)$ be the minimum value of $D(T)$ over all decision trees T with $k > 1$ leaves. Show that $d(k) = \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$. (*Hint:* Consider a decision tree T with k leaves that achieves the minimum. Let i_0 be the number of leaves in LT and $k - i_0$ the number of leaves in RT.)

d. Prove that for a given value of $k > 1$ and i in the range $1 \le i \le k-1$, the function $i \lg i + (k-i)\lg(k-i)$ is minimized at $i = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.

e. Prove that $D(T_A) = \Omega(n! \lg(n!))$, and conclude that the expected time to sort n elements is $\Omega(n \lg n)$.

Now, consider a *randomized* comparison sort B. We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and "randomization" nodes. A randomization node models a random choice of the form RANDOM$(1, r)$ made by algorithm B; the node has r children, each of which is equally likely to be chosen during an execution of the algorithm.

f. Show that for any randomized comparison sort B, there exists a deterministic comparison sort A that makes no more comparisons on the average than B does.

8-2 Sorting in place in linear time

Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

a. Give an algorithm that satisfies criteria 1 and 2 above.

b. Give an algorithm that satisfies criteria 1 and 3 above.

c. Give an algorithm that satisfies criteria 2 and 3 above.

d. Can any of your sorting algorithms from parts (a)–(c) be used to sort n records with b-bit keys using radix sort in $O(bn)$ time? Explain how or why not.

e. Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that the records can be sorted in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable? (*Hint:* How would you do it for $k = 3$?)

8-3 Sorting variable-length items

a. You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over *all* the integers in the array is n. Show how to sort the array in $O(n)$ time.

b. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time.

(Note that the desired order here is the standard alphabetical order; for example, $\texttt{a} < \texttt{ab} < \texttt{b}$.)

8-4 Water jugs

Suppose that you are given n red and n blue water jugs, all of different shapes and sizes. All red jugs hold different amounts of water, as do the blue ones. Moreover, for every red jug, there is a blue jug that holds the same amount of water, and vice versa.

It is your task to find a grouping of the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation will tell you whether the red or the blue jug can hold more water, or if they are of the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

a. Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.

b. Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons an algorithm solving this problem must make.

c. Give a randomized algorithm whose expected number of comparisons is $O(n \lg n)$, and prove that this bound is correct. What is the worst-case number of comparisons for your algorithm?

8-5 Average sorting

Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an n-element array A ***k-sorted*** if, for all $i = 1, 2, \ldots, n-k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

a. What does it mean for an array to be 1-sorted?

b. Give a permutation of the numbers $1, 2, \ldots, 10$ that is 2-sorted, but not sorted.

c. Prove that an n-element array is k-sorted if and only if $A[i] \leq A[i+k]$ for all $i = 1, 2, \ldots, n-k$.

d. Give an algorithm that k-sorts an n-element array in $O(n \lg(n/k))$ time.

We can also show a lower bound on the time to produce a k-sorted array, when k is a constant.

e. Show that a k-sorted array of length n can be sorted in $O(n \lg k)$ time. (*Hint:* Use the solution to Exercise 6.5-8.)

f. Show that when k is a constant, it requires $\Omega(n \lg n)$ time to k-sort an n-element array. (*Hint:* Use the solution to the previous part along with the lower bound on comparison sorts.)

8-6 Lower bound on merging sorted lists

The problem of merging two sorted lists arises frequently. It is used as a subroutine of MERGE-SORT, and the procedure to merge two sorted lists is given as MERGE in Section 2.3.1. In this problem, we will show that there is a lower bound of $2n-1$

on the worst-case number of comparisons required to merge two sorted lists, each containing n items.

First we will show a lower bound of $2n - o(n)$ comparisons by using a decision tree.

a. Show that, given $2n$ numbers, there are $\binom{2n}{n}$ possible ways to divide them into two sorted lists, each with n numbers.

b. Using a decision tree, show that any algorithm that correctly merges two sorted lists uses at least $2n - o(n)$ comparisons.

Now we will show a slightly tighter $2n - 1$ bound.

c. Show that if two elements are consecutive in the sorted order and from opposite lists, then they must be compared.

d. Use your answer to the previous part to show a lower bound of $2n - 1$ comparisons for merging two sorted lists.

Chapter notes

The decision-tree model for studying comparison sorts was introduced by Ford and Johnson [94]. Knuth's comprehensive treatise on sorting [185] covers many variations on the sorting problem, including the information-theoretic lower bound on the complexity of sorting given here. Lower bounds for sorting using generalizations of the decision-tree model were studied comprehensively by Ben-Or [36].

Knuth credits H. H. Seward with inventing counting sort in 1954, and also with the idea of combining counting sort with radix sort. Radix sorting starting with the least significant digit appears to be a folk algorithm widely used by operators of mechanical card-sorting machines. According to Knuth, the first published reference to the method is a 1929 document by L. J. Comrie describing punched-card equipment. Bucket sorting has been in use since 1956, when the basic idea was proposed by E. J. Isaac and R. C. Singleton.

Munro and Raman [229] give a stable sorting algorithm that performs $O(n^{1+\epsilon})$ comparisons in the worst case, where $0 < \epsilon \leq 1$ is any fixed constant. Although any of the $O(n \lg n)$-time algorithms make fewer comparisons, the algorithm by Munro and Raman moves data only $O(n)$ times and operates in place.

The case of sorting n b-bit integers in $o(n \lg n)$ time has been considered by many researchers. Several positive results have been obtained, each under slightly different assumptions about the model of computation and the restrictions placed on the algorithm. All the results assume that the computer memory is divided into addressable b-bit words. Fredman and Willard [99] introduced the fusion tree data structure and used it to sort n integers in $O(n \lg n / \lg \lg n)$ time. This bound was later improved to $O(n\sqrt{\lg n})$ time by Andersson [16]. These algorithms require the use of multiplication and several precomputed constants. Andersson, Hagerup, Nilsson, and Raman [17] have shown how to sort n integers in $O(n \lg \lg n)$ time without using multiplication, but their method requires storage that can be unbounded in terms of n. Using multiplicative hashing, one can reduce the storage needed to $O(n)$, but the $O(n \lg \lg n)$ worst-case bound on the running time becomes an expected-time bound. Generalizing the exponential search trees of Andersson [16], Thorup [297] gave an $O(n(\lg \lg n)^2)$-time sorting algorithm that does not use multiplication or randomization, and uses linear space. Combining these techniques with some new ideas, Han [137] improved the bound for sorting to $O(n \lg \lg n \lg \lg \lg n)$ time. Although these algorithms are important theoretical breakthroughs, they are all fairly complicated and at the present time seem unlikely to compete with existing sorting algorithms in practice.

10.1-5

Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a ***deque*** (double-ended queue) allows insertion and deletion at both ends. Write four $O(1)$-time procedures to insert elements into and delete elements from both ends of a deque constructed from an array.

10.1-6

Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

10.1-7

Show how to implement a stack using two queues. Analyze the running time of the stack operations.

10.2 Linked lists

A ***linked list*** is a data structure in which the objects are arranged in a linear order. Unlike an array, though, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. Linked lists provide a simple, flexible representation for dynamic sets, supporting (though not necessarily efficiently) all the operations listed on page 198.

As shown in Figure 10.3, each element of a ***doubly linked list*** L is an object with a *key* field and two other pointer fields: *next* and *prev*. The object may also contain other satellite data. Given an element x in the list, $next[x]$ points to its successor in the linked list, and $prev[x]$ points to its predecessor. If $prev[x] = \text{NIL}$, the element x has no predecessor and is therefore the first element, or ***head***, of the list. If $next[x] = \text{NIL}$, the element x has no successor and is therefore the last element, or ***tail***, of the list. An attribute $head[L]$ points to the first element of the list. If $head[L] = \text{NIL}$, the list is empty.

A list may have one of several forms. It may be either singly linked or doubly linked, it may be sorted or not, and it may be circular or not. If a list is ***singly linked***, we omit the *prev* pointer in each element. If a list is ***sorted***, the linear order of the list corresponds to the linear order of keys stored in elements of the list; the minimum element is the head of the list, and the maximum element is the tail. If the list is ***unsorted***, the elements can appear in any order. In a ***circular list***, the *prev* pointer of the head of the list points to the tail, and the *next* pointer of the tail of the list points to the head. The list may thus be viewed as a ring of elements. In the remainder of this section, we assume that the lists with which we are working are unsorted and doubly linked.

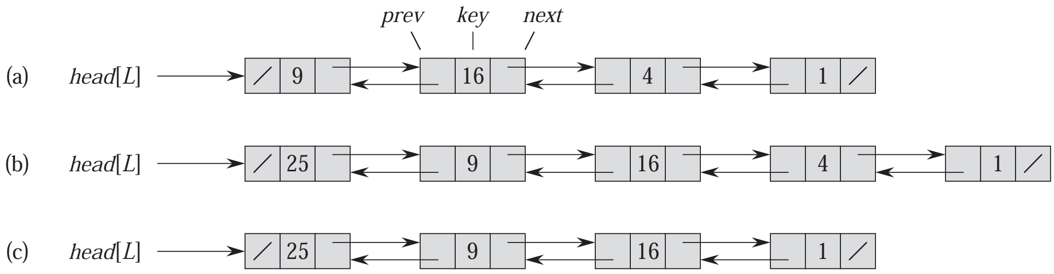


Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with fields for the key and pointers (shown by arrows) to the next and previous objects. The *next* field of the tail and the *prev* field of the head are NIL, indicated by a diagonal slash. The attribute *head*$[L]$ points to the head. **(b)** Following the execution of LIST-INSERT(L, x), where *key*$[x] = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

Searching a linked list

The procedure LIST-SEARCH(L, k) finds the first element with key k in list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then NIL is returned. For the linked list in Figure 10.3(a), the call LIST-SEARCH$(L, 4)$ returns a pointer to the third element, and the call LIST-SEARCH$(L, 7)$ returns NIL.

LIST-SEARCH(L, k)

$x \leftarrow$ *head*$[L]$
while $x \neq$ NIL and *key*$[x] \neq k$
  do $x \leftarrow$ *next*$[x]$
return x

To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

Inserting into a linked list

Given an element x whose *key* field has already been set, the LIST-INSERT procedure "splices" x onto the front of the linked list, as shown in Figure 10.3(b).

```
LIST-INSERT(L, x)
next[x] ← head[L]
if head[L] ≠ NIL
   then prev[head[L]] ← x
head[L] ← x
prev[x] ← NIL
```

The running time for LIST-INSERT on a list of n elements is $O(1)$.

Deleting from a linked list

The procedure LIST-DELETE removes an element x from a linked list L. It must be given a pointer to x, and it then "splices" x out of the list by updating pointers. If we wish to delete an element with a given key, we must first call LIST-SEARCH to retrieve a pointer to the element.

```
LIST-DELETE(L, x)
if prev[x] ≠ NIL
   then next[prev[x]] ← next[x]
   else head[L] ← next[x]
if next[x] ≠ NIL
   then prev[next[x]] ← prev[x]
```

Figure 10.3(c) shows how an element is deleted from a linked list. LIST-DELETE runs in $O(1)$ time, but if we wish to delete an element with a given key, $\Theta(n)$ time is required in the worst case because we must first call LIST-SEARCH.

Sentinels

The code for LIST-DELETE would be simpler if we could ignore the boundary conditions at the head and tail of the list.

```
LIST-DELETE′(L, x)
next[prev[x]] ← next[x]
prev[next[x]] ← prev[x]
```

A ***sentinel*** is a dummy object that allows us to simplify boundary conditions. For example, suppose that we provide with list L an object *nil*$[L]$ that represents NIL but has all the fields of the other list elements. Wherever we have a reference to NIL in list code, we replace it by a reference to the sentinel *nil*$[L]$. As shown in Figure 10.4, this turns a regular doubly linked list into a ***circular, doubly linked list with a sentinel***, in which the sentinel *nil*$[L]$ is placed between the head and

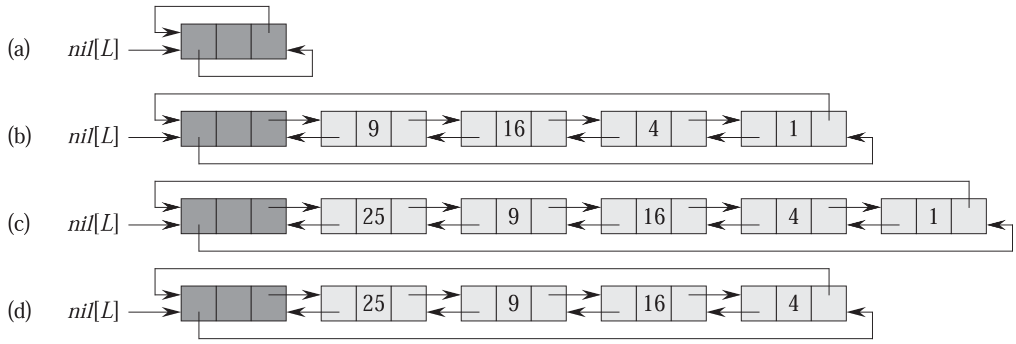


Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel *nil*[L] appears between the head and tail. The attribute *head*[L] is no longer needed, since we can access the head of the list by *next*[*nil*[L]]. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT′(L, x), where *key*[x] = 25. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4.

tail; the field *next*[*nil*[L]] points to the head of the list, and *prev*[*nil*[L]] points to the tail. Similarly, both the *next* field of the tail and the *prev* field of the head point to *nil*[L]. Since *next*[*nil*[L]] points to the head, we can eliminate the attribute *head*[L] altogether, replacing references to it by references to *next*[*nil*[L]]. An empty list consists of just the sentinel, since both *next*[*nil*[L]] and *prev*[*nil*[L]] can be set to *nil*[L].

The code for LIST-SEARCH remains the same as before, but with the references to NIL and *head*[L] changed as specified above.

LIST-SEARCH′(L, k)

$x \leftarrow next[nil[L]]$
while $x \neq nil[L]$ and $key[x] \neq k$
  do $x \leftarrow next[x]$
return x

We use the two-line procedure LIST-DELETE′ to delete an element from the list. We use the following procedure to insert an element into the list.

LIST-INSERT′(L, x)

$next[x] \leftarrow next[nil[L]]$
$prev[next[nil[L]]] \leftarrow x$
$next[nil[L]] \leftarrow x$
$prev[x] \leftarrow nil[L]$

Figure 10.4 shows the effects of LIST-INSERT′ and LIST-DELETE′ on a sample list.

Sentinels rarely reduce the asymptotic time bounds of data structure operations, but they can reduce constant factors. The gain from using sentinels within loops is usually a matter of clarity of code rather than speed; the linked list code, for example, is simplified by the use of sentinels, but we save only $O(1)$ time in the LIST-INSERT′ and LIST-DELETE′ procedures. In other situations, however, the use of sentinels helps to tighten the code in a loop, thus reducing the coefficient of, say, n or n^2 in the running time.

Sentinels should not be used indiscriminately. If there are many small lists, the extra storage used by their sentinels can represent significant wasted memory. In this book, we use sentinels only when they truly simplify the code.

Exercises

10.2-1

Can the dynamic-set operation INSERT be implemented on a singly linked list in $O(1)$ time? How about DELETE?

10.2-2

Implement a stack using a singly linked list L. The operations PUSH and POP should still take $O(1)$ time.

10.2-3

Implement a queue by a singly linked list L. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time.

10.2-4

As written, each loop iteration in the LIST-SEARCH′ procedure requires two tests: one for $x \neq nil[L]$ and one for $key[x] \neq k$. Show how to eliminate the test for $x \neq nil[L]$ in each iteration.

10.2-5

Implement the dictionary operations INSERT, DELETE, and SEARCH using singly linked, circular lists. What are the running times of your procedures?

10.2-6

The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 \cup S_2$ consisting of all the elements of S_1 and S_2. The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

10.2-7
Give a $\Theta(n)$-time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

10.2-8 $\star$
Explain how to implement doubly linked lists using only one pointer value $np[x]$ per item instead of the usual two (*next* and *prev*). Assume that all pointer values can be interpreted as k-bit integers, and define $np[x]$ to be $np[x] = next[x]$ XOR $prev[x]$, the k-bit "exclusive-or" of $next[x]$ and $prev[x]$. (The value NIL is represented by 0.) Be sure to describe what information is needed to access the head of the list. Show how to implement the SEARCH, INSERT, and DELETE operations on such a list. Also show how to reverse such a list in $O(1)$ time.

10.3 Implementing pointers and objects

How do we implement pointers and objects in languages, such as Fortran, that do not provide them? In this section, we shall see two ways of implementing linked data structures without an explicit pointer data type. We shall synthesize objects and pointers from arrays and array indices.

A multiple-array representation of objects

We can represent a collection of objects that have the same fields by using an array for each field. As an example, Figure 10.5 shows how we can implement the linked list of Figure 10.3(a) with three arrays. The array *key* holds the values of the keys currently in the dynamic set, and the pointers are stored in the arrays *next* and *prev*. For a given array index x, $key[x]$, $next[x]$, and $prev[x]$ represent an object in the linked list. Under this interpretation, a pointer x is simply a common index into the *key*, *next*, and *prev* arrays.

In Figure 10.3(a), the object with key 4 follows the object with key 16 in the linked list. In Figure 10.5, key 4 appears in $key[2]$, and key 16 appears in $key[5]$, so we have $next[5] = 2$ and $prev[2] = 5$. Although the constant NIL appears in the *next* field of the tail and the *prev* field of the head, we usually use an integer (such as 0 or -1) that cannot possibly represent an actual index into the arrays. A variable L holds the index of the head of the list.

In our pseudocode, we have been using square brackets to denote both the indexing of an array and the selection of a field (attribute) of an object. Either way, the meanings of $key[x]$, $next[x]$, and $prev[x]$ are consistent with implementation practice.

12 Binary Search Trees

Search trees are data structures that support many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, a search tree can be used both as a dictionary and as a priority queue.

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(\lg n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst-case time. We shall see in Section 12.4 that the expected height of a randomly built binary search tree is $O(\lg n)$, so that basic dynamic-set operations on such a tree take $\Theta(\lg n)$ time on average.

In practice, we can't always guarantee that binary search trees are built randomly, but there are variations of binary search trees whose worst-case performance on basic operations can be guaranteed to be good. Chapter 13 presents one such variation, red-black trees, which have height $O(\lg n)$. Chapter 18 introduces B-trees, which are particularly good for maintaining databases on random-access, secondary (disk) storage.

After presenting the basic properties of binary search trees, the following sections show how to walk a binary search tree to print its values in sorted order, how to search for a value in a binary search tree, how to find the minimum or maximum element, how to find the predecessor or successor of an element, and how to insert into or delete from a binary search tree. The basic mathematical properties of trees appear in Appendix B.

12.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, as shown in Figure 12.1. Such a tree can be represented by a linked data structure in which each node is an object. In addition to a *key* field and satellite data, each node

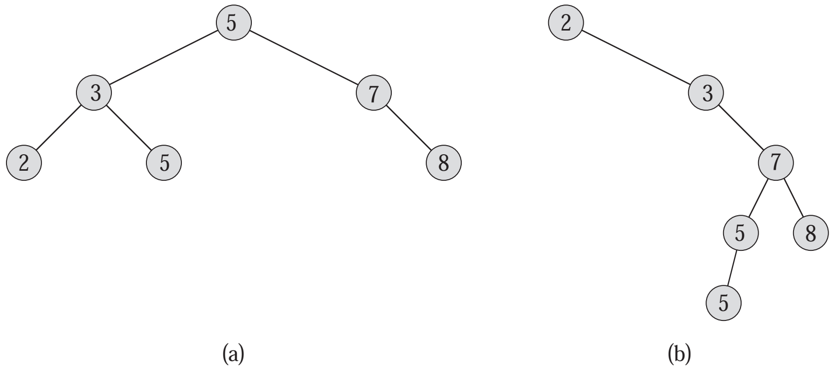


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $key[x]$, and the keys in the right subtree of x are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

contains fields *left*, *right*, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is NIL.

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $key[y] \leq key[x]$. If y is a node in the right subtree of x, then $key[x] \leq key[y]$.

Thus, in Figure 12.1(a), the key of the root is 5, the keys 2, 3, and 5 in its left subtree are no larger than 5, and the keys 7 and 8 in its right subtree are no smaller than 5. The same property holds for every node in the tree. For example, the key 3 in Figure 12.1(a) is no smaller than the key 2 in its left subtree and no larger than the key 5 in its right subtree.

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an ***inorder tree walk***. This algorithm is so named because the key of the root of a subtree is printed between the values in its left subtree and those in its right subtree. (Similarly, a ***preorder tree walk*** prints the root before the values in either subtree, and a ***postorder tree walk*** prints the root after the values in its subtrees.) To use the following procedure to print all the elements in a binary search tree T, we call INORDER-TREE-WALK($root[T]$).

```
INORDER-TREE-WALK(x)
if x ≠ NIL
   then INORDER-TREE-WALK(left[x])
        print key[x]
        INORDER-TREE-WALK(right[x])
```

As an example, the inorder tree walk prints the keys in each of the two binary search trees from Figure 12.1 in the order $2, 3, 5, 5, 7, 8$. The correctness of the algorithm follows by induction directly from the binary-search-tree property.

It takes $\Theta(n)$ time to walk an n-node binary search tree, since after the initial call, the procedure is called recursively exactly twice for each node in the tree—once for its left child and once for its right child. The following theorem gives a more formal proof that it takes linear time to perform an inorder tree walk.

Theorem 12.1
If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

Proof Let $T(n)$ denote the time taken by INORDER-TREE-WALK when it is called on the root of an n-node subtree. INORDER-TREE-WALK takes a small, constant amount of time on an empty subtree (for the test $x \neq$ NIL), and so $T(0) = c$ for some positive constant c.

For $n > 0$, suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes. The time to perform INORDER-TREE-WALK(x) is $T(n) = T(k) + T(n-k-1) + d$ for some positive constant d that reflects the time to execute INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

We use the substitution method to show that $T(n) = \Theta(n)$ by proving that $T(n) = (c+d)n + c$. For $n = 0$, we have $(c+d) \cdot 0 + c = c = T(0)$. For $n > 0$, we have

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c\,, \end{aligned}$$

which completes the proof. ■

Exercises

12.1-1
For the set of keys $\{1, 4, 5, 10, 16, 17, 21\}$, draw binary search trees of height 2, 3, 4, 5, and 6.

12.1-2
What is the difference between the binary-search-tree property and the min-heap property (see page 129)? Can the min-heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time? Explain how or why not.

12.1-3
Give a nonrecursive algorithm that performs an inorder tree walk. (*Hint:* There is an easy solution that uses a stack as an auxiliary data structure and a more complicated but elegant solution that uses no stack but assumes that two pointers can be tested for equality.)

12.1-4
Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

12.1-5
Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

12.2 Querying a binary search tree

A common operation performed on a binary search tree is searching for a key stored in the tree. Besides the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR. In this section, we shall examine these operations and show that each can be supported in time $O(h)$ on a binary search tree of height h.

Searching

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

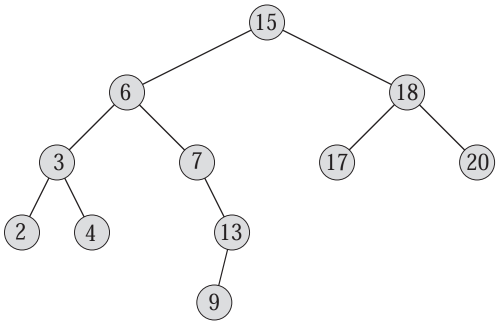


Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \to 6 \to 7 \to 13$ from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
   then return x
if k < key[x]
   then return TREE-SEARCH(left[x], k)
   else return TREE-SEARCH(right[x], k)
```

The procedure begins its search at the root and traces a path downward in the tree, as shown in Figure 12.2. For each node x it encounters, it compares the key k with $key[x]$. If the two keys are equal, the search terminates. If k is smaller than $key[x]$, the search continues in the left subtree of x, since the binary-search-tree property implies that k could not be stored in the right subtree. Symmetrically, if k is larger than $key[x]$, the search continues in the right subtree. The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of TREE-SEARCH is $O(h)$, where h is the height of the tree.

The same procedure can be written iteratively by "unrolling" the recursion into a **while** loop. On most computers, this version is more efficient.

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ key[x]
    do if k < key[x]
          then x ← left[x]
          else x ← right[x]
return x
```

Minimum and maximum

An element in a binary search tree whose key is a minimum can always be found by following *left* child pointers from the root until a NIL is encountered, as shown in Figure 12.2. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x.

```
TREE-MINIMUM(x)
while left[x] ≠ NIL
      do x ← left[x]
return x
```

The binary-search-tree property guarantees that TREE-MINIMUM is correct. If a node x has no left subtree, then since every key in the right subtree of x is at least as large as $key[x]$, the minimum key in the subtree rooted at x is $key[x]$. If node x has a left subtree, then since no key in the right subtree is smaller than $key[x]$ and every key in the left subtree is not larger than $key[x]$, the minimum key in the subtree rooted at x can be found in the subtree rooted at $left[x]$.

The pseudocode for TREE-MAXIMUM is symmetric.

```
TREE-MAXIMUM(x)
while right[x] ≠ NIL
      do x ← right[x]
return x
```

Both of these procedures run in $O(h)$ time on a tree of height h since, as in TREE-SEARCH, the sequence of nodes encountered forms a path downward from the root.

Successor and predecessor

Given a node in a binary search tree, it is sometimes important to be able to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node x is the node with the smallest key greater than $key[x]$. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree.

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2     then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5      do x ← y
6         y ← p[y]
7  return y
```

The code for TREE-SUCCESSOR is broken into two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in the right subtree, which is found in line 2 by calling TREE-MINIMUM($right[x]$). For example, the successor of the node with key 15 in Figure 12.2 is the node with key 17.

On the other hand, as Exercise 12.2-6 asks you to show, if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. In Figure 12.2, the successor of the node with key 13 is the node with key 15. To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent; this is accomplished by lines 3–7 of TREE-SUCCESSOR.

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since we either follow a path up the tree or follow a path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time $O(h)$.

Even if keys are not distinct, we define the successor and predecessor of any node x as the node returned by calls made to TREE-SUCCESSOR(x) and TREE-PREDECESSOR(x), respectively.

In summary, we have proved the following theorem.

Theorem 12.2

The dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can be made to run in $O(h)$ time on a binary search tree of height h. ■

Exercises

12.2-1

Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could *not* be the sequence of nodes examined?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2
Write recursive versions of the TREE-MINIMUM and TREE-MAXIMUM procedures.

12.2-3
Write the TREE-PREDECESSOR procedure.

12.2-4
Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

12.2-5
Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

12.2-6
Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

12.2-7
An inorder tree walk of an n-node binary search tree can be implemented by finding the minimum element in the tree with TREE-MINIMUM and then making $n-1$ calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

12.2-8
Prove that no matter what node we start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k+h)$ time.

12.2-9
Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $key[y]$ is either the smallest key in T larger than $key[x]$ or the largest key in T smaller than $key[x]$.

12.3 Insertion and deletion

The operations of insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that the binary-search-tree property continues to hold. As we shall see, modifying the tree to insert a new element is relatively straightforward, but handling deletion is somewhat more intricate.

Insertion

To insert a new value v into a binary search tree T, we use the procedure TREE-INSERT. The procedure is passed a node z for which $key[z] = v$, $left[z] = \text{NIL}$, and $right[z] = \text{NIL}$. It modifies T and some of the fields of z in such a way that z is inserted into an appropriate position in the tree.

```
TREE-INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4       do y ← x
 5          if key[z] < key[x]
 6             then x ← left[x]
 7             else x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                    ▷ Tree T was empty
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
```

Figure 12.3 shows how TREE-INSERT works. Just like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and traces a path downward. The pointer x traces the path, and the pointer y is maintained as the parent of x. After initialization, the **while** loop in lines 3–7 causes these two pointers to move down the tree, going left or right depending on the comparison of $key[z]$ with $key[x]$, until x is set to NIL. This NIL occupies the position where we wish to place the input item z. Lines 8–13 set the pointers that cause z to be inserted.

Like the other primitive operations on search trees, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h.

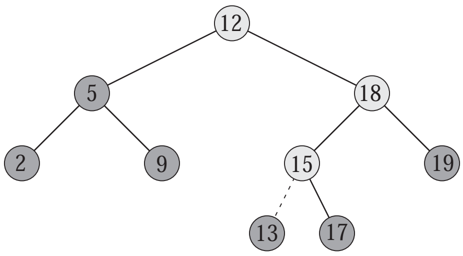


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

Deletion

The procedure for deleting a given node z from a binary search tree takes as an argument a pointer to z. The procedure considers the three cases shown in Figure 12.4. If z has no children, we modify its parent $p[z]$ to replace z with NIL as its child. If the node has only a single child, we "splice out" z by making a new link between its child and its parent. Finally, if the node has two children, we splice out z's successor y, which has no left child (see Exercise 12.2-5) and replace z's key and satellite data with y's key and satellite data.

The code for TREE-DELETE organizes these three cases a little differently.

```
TREE-DELETE(T, z)
if left[z] = NIL or right[z] = NIL
   then y ← z
   else y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
   then x ← left[y]
   else x ← right[y]
if x ≠ NIL
   then p[x] ← p[y]
if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
return y
```

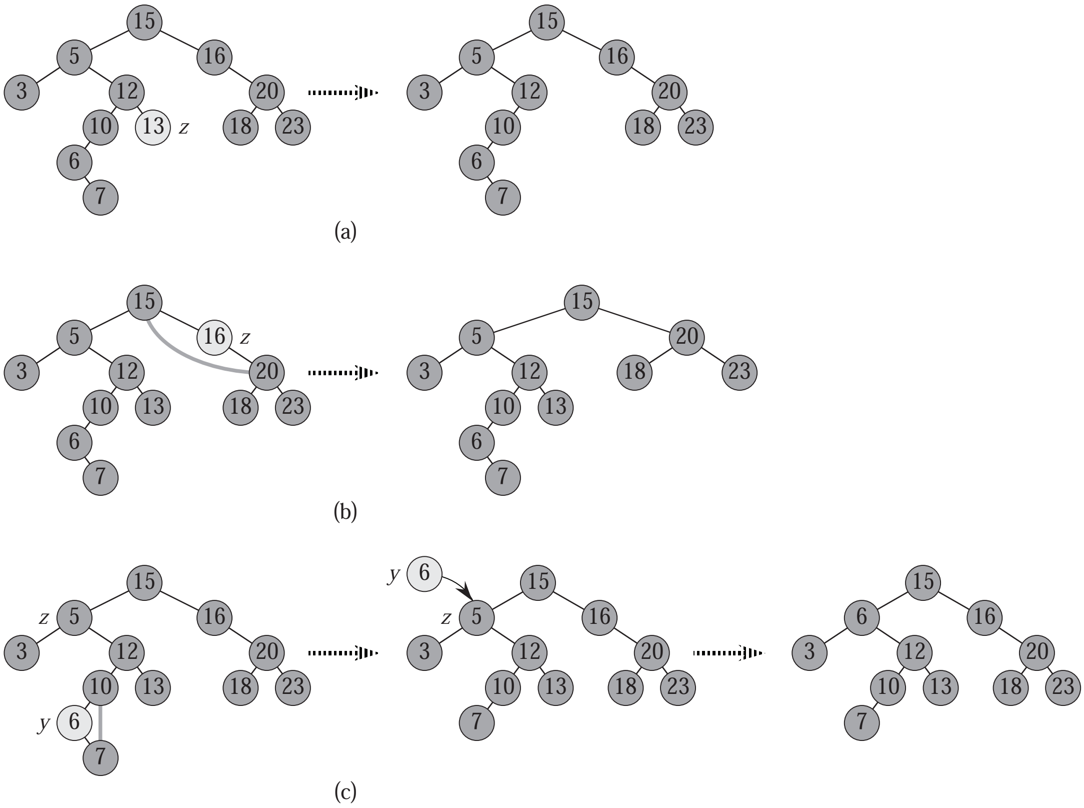


Figure 12.4 Deleting a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded. **(a)** If z has no children, we just remove it. **(b)** If z has only one child, we splice out z. **(c)** If z has two children, we splice out its successor y, which has at most one child, and then replace z's key and satellite data with y's key and satellite data.

In lines 1–3, the algorithm determines a node y to splice out. The node y is either the input node z (if z has at most 1 child) or the successor of z (if z has two children). Then, in lines 4–6, x is set to the non-NIL child of y, or to NIL if y has no children. The node y is spliced out in lines 7–13 by modifying pointers in $p[y]$ and x. Splicing out y is somewhat complicated by the need for proper handling of the boundary conditions, which occur when $x = \text{NIL}$ or when y is the root. Finally, in lines 14–16, if the successor of z was the node spliced out, y's key and satellite data are moved to z, overwriting the previous key and satellite data. The node y is returned in line 17 so that the calling procedure can recycle it via the free list. The procedure runs in $O(h)$ time on a tree of height h.

In summary, we have proved the following theorem.

Theorem 12.3
The dynamic-set operations INSERT and DELETE can be made to run in $O(h)$ time on a binary search tree of height h. ■

Exercises

12.3-1
Give a recursive version of the TREE-INSERT procedure.

12.3-2
Suppose that a binary search tree is constructed by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of nodes examined when the value was first inserted into the tree.

12.3-3
We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

12.3-4
Suppose that another data structure contains a pointer to a node y in a binary search tree, and suppose that y's predecessor z is deleted from the tree by the procedure TREE-DELETE. What problem can arise? How can TREE-DELETE be rewritten to solve this problem?

12.3-5
Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

12.3-6
When node z in TREE-DELETE has two children, we could splice out its predecessor rather than its successor. Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be changed to implement such a fair strategy?

⋆ 12.4 Randomly built binary search trees

We have shown that all the basic operations on a binary search tree run in $O(h)$ time, where h is the height of the tree. The height of a binary search tree varies, however, as items are inserted and deleted. If, for example, the items are inserted in strictly increasing order, the tree will be a chain with height $n-1$. On the other hand, Exercise B.5-4 shows that $h \geq \lfloor \lg n \rfloor$. As with quicksort, we can show that the behavior of the average case is much closer to the best case than the worst case.

Unfortunately, little is known about the average height of a binary search tree when both insertion and deletion are used to create it. When the tree is created by insertion alone, the analysis becomes more tractable. Let us therefore define a ***randomly built binary search tree*** on n keys as one that arises from inserting the keys in random order into an initially empty tree, where each of the $n!$ permutations of the input keys is equally likely. (Exercise 12.4-3 asks you to show that this notion is different from assuming that every binary search tree on n keys is equally likely.) In this section, we shall show that the expected height of a randomly built binary search tree on n keys is $O(\lg n)$. We assume that all keys are distinct.

We start by defining three random variables that help measure the height of a randomly built binary search tree. We denote the height of a randomly built binary search on n keys by X_n, and we define the ***exponential height*** $Y_n = 2^{X_n}$. When we build a binary search tree on n keys, we choose one key as that of the root, and we let R_n denote the random variable that holds this key's rank within the set of n keys. The value of R_n is equally likely to be any element of the set $\{1, 2, \ldots, n\}$. If $R_n = i$, then the left subtree of the root is a randomly built binary search tree on $i-1$ keys, and the right subtree is a randomly built binary search tree on $n-i$ keys. Because the height of a binary tree is one more than the larger of the heights of the two subtrees of the root, the exponential height of a binary tree is twice the larger of the exponential heights of the two subtrees of the root. If we know that $R_n = i$, we therefore have that

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}) .$$

As base cases, we have $Y_1 = 1$, because the exponential height of a tree with 1 node is $2^0 = 1$ and, for convenience, we define $Y_0 = 0$.

Next we define indicator random variables $Z_{n,1}, Z_{n,2}, \ldots, Z_{n,n}$, where

$$Z_{n,i} = \mathrm{I}\{R_n = i\} .$$

Because R_n is equally likely to be any element of $\{1, 2, \ldots, n\}$, we have that $\Pr\{R_n = i\} = 1/n$ for $i = 1, 2, \ldots, n$, and hence, by Lemma 5.1,

$$\mathrm{E}[Z_{n,i}] = 1/n , \tag{12.1}$$

for $i = 1, 2, \ldots, n$. Because exactly one value of $Z_{n,i}$ is 1 and all others are 0, we also have

$$Y_n = \sum_{i=1}^{n} Z_{n,i}\,(2 \cdot \max(Y_{i-1}, Y_{n-i}))\ .$$

We will show that $\mathrm{E}[Y_n]$ is polynomial in n, which will ultimately imply that $\mathrm{E}[X_n] = O(\lg n)$.

The indicator random variable $Z_{n,i} = \mathrm{I}\{R_n = i\}$ is independent of the values of Y_{i-1} and Y_{n-i}. Having chosen $R_n = i$, the left subtree, whose exponential height is Y_{i-1}, is randomly built on the $i-1$ keys whose ranks are less than i. This subtree is just like any other randomly built binary search tree on $i-1$ keys. Other than the number of keys it contains, this subtree's structure is not affected at all by the choice of $R_n = i$; hence the random variables Y_{i-1} and $Z_{n,i}$ are independent. Likewise, the right subtree, whose exponential height is Y_{n-i}, is randomly built on the $n-i$ keys whose ranks are greater than i. Its structure is independent of the value of R_n, and so the random variables Y_{n-i} and $Z_{n,i}$ are independent. Hence,

$$\begin{aligned}
\mathrm{E}[Y_n] &= \mathrm{E}\left[\sum_{i=1}^{n} Z_{n,i}\,(2 \cdot \max(Y_{i-1}, Y_{n-i}))\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}\,(2 \cdot \max(Y_{i-1}, Y_{n-i}))] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n} \mathrm{E}[Z_{n,i}]\,\mathrm{E}[2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(by independence)} \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathrm{E}[2 \cdot \max(Y_{i-1}, Y_{n-i})] && \text{(by equation (12.1))} \\
&= \frac{2}{n} \sum_{i=1}^{n} \mathrm{E}[\max(Y_{i-1}, Y_{n-i})] && \text{(by equation (C.21))} \\
&\le \frac{2}{n} \sum_{i=1}^{n} (\mathrm{E}[Y_{i-1}] + \mathrm{E}[Y_{n-i}]) && \text{(by Exercise C.3-4)}\ .
\end{aligned}$$

Each term $\mathrm{E}[Y_0], \mathrm{E}[Y_1], \ldots, \mathrm{E}[Y_{n-1}]$ appears twice in the last summation, once as $\mathrm{E}[Y_{i-1}]$ and once as $\mathrm{E}[Y_{n-i}]$, and so we have the recurrence

$$\mathrm{E}[Y_n] \le \frac{4}{n} \sum_{i=0}^{n-1} \mathrm{E}[Y_i]\ . \tag{12.2}$$

Using the substitution method, we will show that for all positive integers n, the recurrence (12.2) has the solution

$$\mathrm{E}[Y_n] \le \frac{1}{4}\binom{n+3}{3}\ .$$

In doing so, we will use the identity

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4} . \tag{12.3}$$

(Exercise 12.4-1 asks you to prove this identity.)

For the base case, we verify that the bound

$$1 = Y_1 = \mathrm{E}[Y_1] \le \frac{1}{4}\binom{1+3}{3} = 1$$

holds. For the substitution, we have that

$$\begin{aligned}
\mathrm{E}[Y_n] &\le \frac{4}{n}\sum_{i=0}^{n-1} \mathrm{E}[Y_i] \\
&= \frac{4}{n}\sum_{i=0}^{n-1} \frac{1}{4}\binom{i+3}{3} && \text{(by the inductive hypothesis)} \\
&= \frac{1}{n}\sum_{i=0}^{n-1} \binom{i+3}{3} \\
&= \frac{1}{n}\binom{n+3}{4} && \text{(by equation (12.3))} \\
&= \frac{1}{n} \cdot \frac{(n+3)!}{4!\,(n-1)!} \\
&= \frac{1}{4} \cdot \frac{(n+3)!}{3!\,n!} \\
&= \frac{1}{4}\binom{n+3}{3} .
\end{aligned}$$

We have bounded $\mathrm{E}[Y_n]$, but our ultimate goal is to bound $\mathrm{E}[X_n]$. As Exercise 12.4-4 asks you to show, the function $f(x) = 2^x$ is convex (see page 1109). Therefore, we can apply Jensen's inequality (C.25), which says that

$$2^{\mathrm{E}[X_n]} \le \mathrm{E}\left[2^{X_n}\right] = \mathrm{E}[Y_n] ,$$

to derive that

$$\begin{aligned}
2^{\mathrm{E}[X_n]} &\le \frac{1}{4}\binom{n+3}{3} \\
&= \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} \\
&= \frac{n^3 + 6n^2 + 11n + 6}{24} .
\end{aligned}$$

Taking logarithms of both sides gives $\mathrm{E}[X_n] = O(\lg n)$. Thus, we have proven the following:

Theorem 12.4
The expected height of a randomly built binary search tree on n keys is $O(\lg n)$. ■

Exercises

12.4-1
Prove equation (12.3).

12.4-2
Describe a binary search tree on n nodes such that the average depth of a node in the tree is $\Theta(\lg n)$ but the height of the tree is $\omega(\lg n)$. Give an asymptotic upper bound on the height of an n-node binary search tree in which the average depth of a node is $\Theta(\lg n)$.

12.4-3
Show that the notion of a randomly chosen binary search tree on n keys, where each binary search tree of n keys is equally likely to be chosen, is different from the notion of a randomly built binary search tree given in this section. (*Hint:* List the possibilities when $n = 3$.)

12.4-4
Show that the function $f(x) = 2^x$ is convex.

12.4-5 ⋆
Consider RANDOMIZED-QUICKSORT operating on a sequence of n input numbers. Prove that for any constant $k > 0$, all but $O(1/n^k)$ of the $n!$ input permutations yield an $O(n \lg n)$ running time.

Problems

12-1 Binary search trees with equal keys
Equal keys pose a problem for the implementation of binary search trees.

a. What is the asymptotic performance of TREE-INSERT when used to insert n items with identical keys into an initially empty binary search tree?

We propose to improve TREE-INSERT by testing before line 5 whether or not $key[z] = key[x]$ and by testing before line 11 whether or not $key[z] = key[y]$. If equality holds, we implement one of the following strategies. For each strategy, find the asymptotic performance of inserting n items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, in which

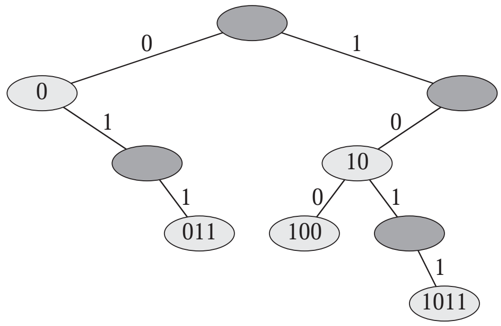


Figure 12.5 A radix tree storing the bit strings 1011, 10, 011, 100, and 0. Each node's key can be determined by traversing the path from the root to that node. There is no need, therefore, to store the keys in the nodes; the keys are shown here for illustrative purposes only. Nodes are heavily shaded if the keys corresponding to them are not in the tree; such nodes are present only to establish a path to other nodes.

we compare the keys of z and x. Substitute y for x to arrive at the strategies for line 11.)

b. Keep a boolean flag $b[x]$ at node x, and set x to either $left[x]$ or $right[x]$ based on the value of $b[x]$, which alternates between FALSE and TRUE each time x is visited during insertion of a node with the same key as x.

c. Keep a list of nodes with equal keys at x, and insert z into the list.

d. Randomly set x to either $left[x]$ or $right[x]$. (Give the worst-case performance and informally derive the average-case performance.)

12-2 Radix trees

Given two strings $a = a_0a_1 \ldots a_p$ and $b = b_0b_1 \ldots b_q$, where each a_i and each b_j is in some ordered set of characters, we say that string a is ***lexicographically less than*** string b if either

1. there exists an integer j, where $0 \le j \le \min(p, q)$, such that $a_i = b_i$ for all $i = 0, 1, \ldots, j-1$ and $a_j < b_j$, or
2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, \ldots, p$.

For example, if a and b are bit strings, then $10100 < 10110$ by rule 1 (letting $j = 3$) and $10100 < 101000$ by rule 2. This is similar to the ordering used in English-language dictionaries.

The ***radix tree*** data structure shown in Figure 12.5 stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key $a = a_0a_1 \ldots a_p$, we go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct binary strings whose lengths sum to n. Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the example in Figure 12.5, the output of the sort should be the sequence 0, 011, 10, 100, 1011.

12-3 Average node depth in a randomly built binary search tree

In this problem, we prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\lg n)$. Although this result is weaker than that of Theorem 12.4, the technique we shall use reveals a surprising similarity between the building of a binary search tree and the running of RANDOMIZED-QUICKSORT from Section 7.3.

We define the ***total path length*** $P(T)$ of a binary tree T as the sum, over all nodes x in T, of the depth of node x, which we denote by $d(x, T)$.

a. Argue that the average depth of a node in T is

$$\frac{1}{n}\sum_{x \in T} d(x, T) = \frac{1}{n} P(T) .$$

Thus, we wish to show that the expected value of $P(T)$ is $O(n \lg n)$.

b. Let T_L and T_R denote the left and right subtrees of tree T, respectively. Argue that if T has n nodes, then

$$P(T) = P(T_L) + P(T_R) + n - 1 .$$

c. Let $P(n)$ denote the average total path length of a randomly built binary search tree with n nodes. Show that

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

d. Show that $P(n)$ can be rewritten as

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n) .$$

e. Recalling the alternative analysis of the randomized version of quicksort given in Problem 7-2, conclude that $P(n) = O(n \lg n)$.

At each recursive invocation of quicksort, we choose a random pivot element to partition the set of elements being sorted. Each node of a binary search tree partitions the set of elements that fall into the subtree rooted at that node.

f. Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree. (The order in which comparisons are made may differ, but the same comparisons must be made.)

12-4 Number of different binary trees

Let b_n denote the number of different binary trees with n nodes. In this problem, you will find a formula for b_n, as well as an asymptotic estimate.

a. Show that $b_0 = 1$ and that, for $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$

b. Referring to Problem 4-5 for the definition of a generating function, let $B(x)$ be the generating function

$$B(x) = \sum_{n=0}^{\infty} b_n x^n .$$

Show that $B(x) = xB(x)^2 + 1$, and hence one way to express $B(x)$ in closed form is

$$B(x) = \frac{1}{2x}\left(1 - \sqrt{1-4x}\right) .$$

The ***Taylor expansion*** of $f(x)$ around the point $x = a$ is given by

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k ,$$

where $f^{(k)}(x)$ is the kth derivative of f evaluated at x.

c. Show that

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(the nth ***Catalan number***) by using the Taylor expansion of $\sqrt{1-4x}$ around $x = 0$. (If you wish, instead of using the Taylor expansion, you may use the generalization of the binomial expansion (C.4) to nonintegral exponents n, where for any real number n and for any integer k, we interpret $\binom{n}{k}$ to be $n(n-1)\cdots(n-k+1)/k!$ if $k \geq 0$, and 0 otherwise.)

d. Show that

$$b_n = \frac{4^n}{\sqrt{\pi}\, n^{3/2}} (1 + O(1/n)) \ .$$

Chapter notes

Knuth [185] contains a good discussion of simple binary search trees as well as many variations. Binary search trees seem to have been independently discovered by a number of people in the late 1950's. Radix trees are often called tries, which comes from the middle letters in the word retrieval. They are also discussed by Knuth [185].

Section 15.5 will show how to construct an optimal binary search tree when search frequencies are known prior to constructing the tree. That is, given the frequencies of searching for each key and the frequencies of searching for values that fall between keys in the tree, we construct a binary search tree for which a set of searches that follows these frequencies examines the minimum number of nodes.

The proof in Section 12.4 that bounds the expected height of a randomly built binary search tree is due to Aslam [23]. Martínez and Roura [211] give randomized algorithms for insertion into and deletion from binary search trees in which the result of either operation is a random binary search tree. Their definition of a random binary search tree differs slightly from that of a randomly built binary search tree in this chapter, however.

13 Red-Black Trees

Chapter 12 showed that a binary search tree of height h can implement any of the basic dynamic-set operations—such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE—in $O(h)$ time. Thus, the set operations are fast if the height of the search tree is small; but if its height is large, their performance may be no better than with a linked list. Red-black trees are one of many search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worst case.

13.1 Properties of red-black trees

A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK. By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately ***balanced***.

Each node of the tree now contains the fields *color*, *key*, *left*, *right*, and *p*. If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value NIL. We shall regard these NIL's as being pointers to external nodes (leaves) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A binary search tree is a red-black tree if it satisfies the following ***red-black properties***:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.