Süleyman Demirel University
Faculty of Engineering
Department of Computer Science

Fundamental Structures OF Computer Science

(Course materials)

Part 2

Data Structures and Algorithm Analysis

(*Lists, Stacks, Queues, Trees, Hashing, Heaps, Sorting*)

Almaty ◦ 2002

SÜLEYMAN DEMIREL UNIVERSITY
Faculty of Engineering
Department of Computer Science

Fundamental Structures OF Computer Science

(Course materials)

Part 2

Data Structures and Algorithm Analysis

(*Lists, Stacks, Queues, Trees, Hashing, Heaps, Sorting*)

Almaty • 2002

ББК 32.973-01я73

П27

Mikhail G.Peretyat'kin.
П27 "Fundamental Structures of Computer Science. (Course materials)
Part 2. Data Structures and Algorithm Analysis",
Almaty – 2002, 266 pages

ISBN 9965-9150-5-9

ББК 32.973-01я73

П $\frac{4310020000}{00(05)-02}$

ISBN 9965-9150-5-9

Serial Parts of the Course:

Part 1 :
Introduction to Automata and Formal Languages [1], [5]
(Finite Memory Programs, Finite Automata, Recursion, and Formal Languages and Grammars)

Part 2 :
Data Structures and Algorithm Analysis [2], [3], [4] [5]
(analysis, correctness, lists, stacks, queues, trees, hashing, heaps, sorting)

References

1. EITAN GURARI, ***An introduction to the Theory of Computation***, Computer Science Press, 1989. (Lecture notes, Ohio University, Spring 1999, 321p.).
2. JEFFRY H. KINGSTON, ***Algorithms and Data Structures (Design, Correctness, Analysis)***. Addison-Wesley, Second edition, 1998, 380p.
3. MARK ALLEN WEISS, ***Data Structures and Algorithm Analysis in C***, Addison-Wesley, Second edition, 511p.
4. ROBERT L. KRUSE, ***Data Structures and Program Design***, Prentice Hall of India, New Delhi–110 001, 1996, Third Edition, 689p.
5. KENNETH H. ROSEN, ***Discrete Mathematics and its Applications***, McGraw-Hill, Mathematics series, 1995, 709p.

The course materials are prepared and assembled by

Mikhail G. Peretyat'kin

PREFACE

This book describes data structures, methods of organizing large amounts of data, and algorithm analysis, the estimation of the running time of algorithms. As computers become faster and faster, the need for programs that can handle large amounts of input becomes more acute. Paradoxically, this requires more careful attention to efficiency, since inefficiencies in programs become most obvious when input sizes are large. By analyzing an algorithm before it is actually coded, students can decide if a particular solution will be feasible. Therefore, no algorithm or data structure is presented without an explanation of its running time. As computers have become more powerful, the problems they must solve have become larger and more complex, requiring development of more intricate programs. The goal of this text is to teach students good programming and algorithm analysis skills simultaneously so that they can develop such programs with the maximum amount of efficiency.

Chapters 1 and 2 give some general approach to investigate programs and describe main classes of algorithms.

Chapter 3 covers lists, stacks, and queues. The emphasis here is on coding these data structures using abstract data types, fast implementation of these data structures, and an exposition of some of their uses. There are almost no programs (just routines), but the exercises contain plenty of ideas for programming assignments.

Chapter 4 covers trees, with an emphasis on search trees, including external search trees (B-trees). The UNIX file system and expression trees are used as examples. AVL trees and splay trees are introduced but not analyzed. Seventy-five percent of the code is written, leaving similar cases to be completed by the student.

Chapter 5 is a relatively short chapter concerning hash tables. Some analysis is performed, and extendible hashing is covered at the end of the chapter.

Chapter 6 is about priority queues. Binary heaps are covered, and there is additional material on some of the theoretically interesting implementations of priority queues.

Chapter 7 covers sorting. It is very specific with respect to coding details and analysis. All the important general-purpose sorting algorithms are covered and compared. Four algorithms are analyzed in detail: insertion sort, Shellsort, heapsort, and quicksort. The analysis of the average-case running time of heap sort is new to this edition. External sorting is covered at the end of the chapter.

Abstract data types have helped greatly in organizing the subject matter, both by classifying and specifying data structures, and by removing them from the algorithms. They permeate the book, and there are whole chapters devoted to their implementation. By counting the number of times that a characteristic operation is performed, the analyses give quite precise results, without excessive detail.

Exercises, provided at the end of each chapter, match the order in which material is presented. The last exercises may address the chapter as a whole rather than a specific section. Difficult exercises are marked with an asterisk, and more challenging exercises have two asterisks.

References are placed at the end of each chapter. Generally the references either are historical, representing the original source of the material, or they represent extensions and improvements to the results given in the text. Some references represent solutions to exercises.

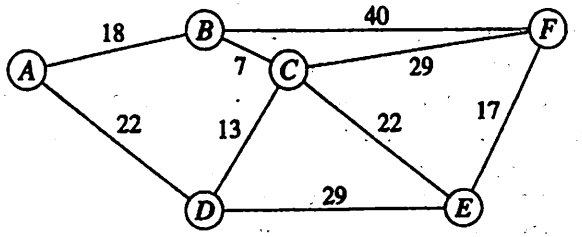
This book is intended as a text for a one-semester second or third year course on algorithms and data structures. Chapters 1-7 of the book provide enough material for one-semester data structures courses. It aims to present the central topics of the subject under a coherent organization, with emphasis more on depth of treatment than on broad survey.

Chapter 1

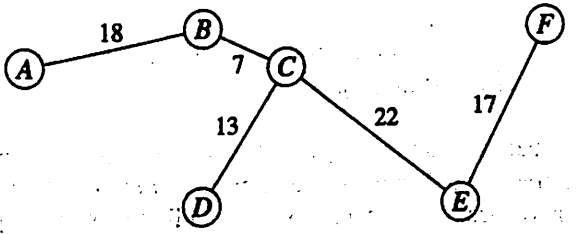
Algorithm Correctness

Most algorithms are straightforward and obviously correct. For example, the algorithm for summing the elements of a set of numbers, by adding each in turn into a *sum* variable, is of this kind. If all algorithms were like this, there would be no need to study algorithm correctness.

Here is an algorithm, simple to state, whose correctness is not nearly so obvious. We are given a map showing cities and the distances between them:



The problem is to connect all the cities together with fiber-optic cable, using links whose total distance is as small as possible. For the map just given, the answer is



One algorithm for this problem (Kruskal's algorithm from Section 12.2) takes each link in turn, from the shortest to the longest, and adds it to the growing result whenever doing so would not introduce a cycle. This is plausible, and it works

for the instance above, but is it obviously correct in every instance? Not at all. To understand Kruskal's algorithm and others like it, intuition is not enough: something more formal is needed.

The study of algorithm correctness, as this book goes about it, is known as *axiomatic semantics*, and it is principally owing to Floyd (1967) and Hoare (1969). It is possible, using the methods of axiomatic semantics, to prove that an algorithm is correct as rigorously as one can prove a theorem in logic. This will not be attempted here, because it is an entire subject in itself; see, for example, Dijkstra (1976) and Gries (1981). Instead, a less rigorous approach will be used which is compatible with the fully rigorous one, but which is more appropriate to our aim of understanding why algorithms are correct.

1.1 Problems and specifications

A *problem* is a general question to be answered, usually possessing one or more *parameters*. A problem is specified by describing the form the parameters are to take, and the question that is being asked about them. For example, the *minimum-finding problem* is 'S is a set of numbers. What is a minimum element of S?' It has one parameter, the set of numbers S.

An *instance* of a problem is an assignment of values to the parameters. For example, '$S = \{5, 2, 6, 9\}$' is an instance of the minimum-finding problem.

An *algorithm* is a step-by-step procedure for taking any instance of a problem and producing a correct answer for that instance. If several answers are equally correct, as often happens, the algorithm may produce any one. An algorithm is *correct* if it is guaranteed to produce a correct answer to every instance of the problem.

Specifying a problem can be difficult, because great precision is needed. For example, the empty set has no minimum element, so the specification above for the minimum-finding problem is flawed. A good way to state a specification precisely is to give two boolean expressions, or *conditions*: the first, the *precondition*, states what may be assumed true initially; the second, the *postcondition*, states what is to be true about the result. The minimum-finding problem could be specified like this:

Pre: S is a finite, non-empty set of integers
Post: m is a minimum element of S

By writing (there exists $x \in S$ such that $m = x$) **and** (for all $x \in S$, $m \leq x$), it is possible to express more formally what it means for m to be a minimum element of S. Whether this degree of formalism is worthwhile or not will depend on the use to which the specification is put.

By assuming that all instances are non-empty, we are saying that we don't care what an algorithm for this problem does if it is given the empty set. It is the user's responsibility to supply only instances in accord with the precondition.

1.2 Recursive algorithms

Newcomers to recursion are often confused by the apparent circularity of recursive definitions: to solve a problem, first solve the problem. As a first, simple example, consider this well-known recursive algorithm for calculating $n!$, the product of the first n natural numbers:

```
factorial(n: INTEGER): INTEGER is
  do
    if n = 0 then
      Result := 1
    else
      Result := n * factorial(n–1)
    end
  end
```

A naive approach to understanding this algorithm, based on tracing its behavior, is possible but becomes very confusing as the recursive calls build up.

A much clearer view is obtainable when the ideas of problems, instances, and formal specification using preconditions and postconditions are used. The problem of calculating $n!$ has specification

```
-- Pre: n is an integer such that n ≥ 0
x := factorial(n)
-- Post: x = n!
```

The convention used here is to include the conditions as comments in a program fragment at the points where they should be true.

Instead of trying to understand the algorithm by tracing it, the approach recommended here is to prove that the program satisfies its specification, and to use the proof as a guide to understanding the algorithm.

It must be shown that for all $n \geq 0$, *factorial*(n) returns $n!$. This statement is clearly suited to a proof by induction on n. First it must be shown that *factorial*(0) returns 0!, and second, that if *factorial*(j) returns $j!$ for all j such that $0 \leq j \leq n - 1$, then *factorial*(n) returns $n!$.

The key point is that during the proof we will be *assuming* that *factorial*(n–1) returns $(n - 1)!$, as proof by induction permits us to do. That is, to understand this algorithm there is no need to trace the call to *factorial*(n–1), and in fact to do so only leads to confusion. Instead, the recursive call is assumed correct by induction, and the formal specification defines its effect without any need to trace it.

Here is the formal proof:

Theorem 1.1: For all integers $n \geq 0$, *factorial*(n) returns $n!$.
Proof: by induction on n.
Basis step: $n = 0$. Then the test $n = 0$ succeeds, and the algorithm returns 1. This is correct, since $0! = 1$.

Inductive step: The inductive hypothesis is that *factorial(j)* returns $j!$ for all j in the range $0 \le j \le n-1$. It must be shown that *factorial(n)* returns $n!$. Since $n > 0$, the algorithm returns $n *$ *factorial*$(n-1)$. By the inductive hypothesis, *factorial*$(n-1)$ returns $(n-1)!$, and so *factorial(n)* returns $n \times (n-1)!$, which equals $n!$. □

Notice that the proof is only possible because the recursive call is given a smaller instance than the original, so that the inductive hypothesis may be applied to it. Also, the theorem says nothing about the behavior of *factorial(n)* for $n < 0$, and in fact the algorithm never halts for these n.

For a second example, consider the *binary search* algorithm, whose goal is to determine whether x is present in the sorted array *entries.item(a..b)*:

```
-- Pre: a ≤ b + 1 and entries.item(a..b) is a sorted array
found := binary_search(a, b, x);
-- Post: found = x ∈ entries.item(a..b) and entries is unchanged
```

The code is

```
binary_search(a, b: INTEGER; x: KEY_TYPE): BOOLEAN is
    local
        mid: INTEGER;
    do
        if a > b then
            Result := false
        else
            mid := (a + b) // 2;
            if x = entries.item(mid) then
                Result := true
            elseif x < entries.item(mid) then
                Result := binary_search(a, mid-1, x)
            else
                Result := binary_search(mid+1, b, x)
            end
        end
    end
```

where // denotes integer division. Binary search first compares x with the middle entry of the array, *entries.item(mid)*. If $x <$ *entries.item(mid)*, x must lie in the left half of the array if it is present at all; if $x >$ *entries.item(mid)*, it must lie in the right half. The proof is by induction on the size of the array *entries.item(a..b)*:

Theorem 1.2: For all $n \ge 0$, where $n = b - a + 1$ equals the number of elements in the array *entries.item(a..b)*, *binary_search(a, b, x)* correctly returns the value of the condition $x \in$ *entries.item(a..b)*.
Proof: by induction on n.
Basis step: $n = 0$. The array is empty, so $a = b + 1$, the test $a > b$ succeeds, and the algorithm returns **false**. This is correct: x cannot be present in an empty array.

Inductive step: $n > 0$. The inductive hypothesis is that, for all j lying in the range $0 \le j \le n-1$, where $j = b' - a' + 1$, *binary_search*(a', b', x) correctly returns the condition $x \in$ *entries.item*$(a'..b')$. From the calculation $mid := (a + b)$ // 2 it follows that $a \le mid \le b$. If $x =$ *entries.item(mid)*, clearly $x \in$ *entries.item(a..b)* and the algorithm correctly returns the value **true**. If $x <$ *entries.item(mid)*, then since *entries* is sorted it follows that $x \in$ *entries.item(a..b)* if and only if $x \in$ *entries.item*$(a..mid-1)$. By the inductive hypothesis, this second condition is returned by *binary_search*$(a, mid-1, x)$. The inductive hypothesis does apply, since $0 \le (mid - 1) - a + 1 \le n - 1$. The case $x >$ *entries.item(mid)* is similar, and so the algorithm is correct for all instances of size n. □

1.3 Iterative algorithms

Iterative algorithms (those containing a loop) are much easier to trace than recursive algorithms, but they are not always easier to understand. For example, the mysterious algorithm given at the beginning of this chapter, for building communication networks, has a simple iterative form.

It turns out that proofs of correctness of iterative algorithms can supply the ideas needed to understand such algorithms. In particular, a condition called the *loop invariant*, which lies at the heart of every such proof, is the key to understanding even the most mysterious iterative algorithms.

A *loop invariant I* of an algorithm containing an **until** loop is a condition which is true at the beginning of each iteration of the loop, at the moment just before the **until** condition is tested.

Two questions are raised by this. First, how do we go about finding loop invariants? And second, how do we use them to prove that algorithms are correct? It seems best to tackle both questions simultaneously, by way of examples. Our first, very simple, example finds the sum of the elements of the array *entries.item(a..b)*:

```
-- Pre: a ≤ b + 1
from i := a; sum := 0 until i = b + 1 loop
    sum := sum + entries.item(i);
    i := i + 1
end
-- Post: sum = Σ_{j=a}^{b} entries.item(j)
```

As usual, the precondition and postcondition have been included as comments at the points where they should be true. By definition, *entries.item*$(a..a-1)$ denotes an empty array whose sum is 0, and this algorithm calculates this empty sum correctly.

Now, what is the loop invariant of this algorithm? We are looking for a condition which is true at the beginning of each iteration of the loop. Some trivial examples are **true** and $i \ge a$; but, to be useful, a loop invariant must express everything that the algorithm has achieved up to the point where it occurs.

It is often helpful to imagine the state of affairs when about half the iterations are complete; what is true then? At that point, the *sum* variable contains the sum of all the items examined so far. More precisely,

$$sum = \sum_{j=a}^{i-1} entries.item(j)$$

This condition is the loop invariant of the summing algorithm; the reader may easily verify intuitively that this condition holds at the beginning of each iteration.

Even without proof, the loop invariant is very useful to know. For example, it makes an excellent comment (at least, it does for loops less trivial than this summing example). To know the loop invariant is to understand the algorithm.

For the record, and as a model for the more difficult proofs that appear later in this book, here is a proof that the condition really is a loop invariant:

Theorem 1.3 (Loop invariant of summing algorithm): At the beginning of the *k*th iteration of the summing algorithm above, $sum = \sum_{j=a}^{i-1} entries.item(j)$.

Proof: by induction on *k*.

Basis step: $k = 1$. At the beginning of the first iteration, the initialization statements ensure that $sum = 0$ and $i = a$. Since $0 = \sum_{j=a}^{a-1} entries.item(j)$, the condition holds.

Inductive step: The inductive hypothesis is that $sum = \sum_{j=a}^{i-1} entries.item(j)$ at the beginning of the *k*th iteration. Since the aim here is to prove that this condition holds after one more iteration, it may also be assumed at this point that the loop is not about to terminate, or in other words, that $i \neq b + 1$. Let sum' and i' be the values of *sum* and *i* at the beginning of the $(k + 1)$st iteration. It is required to show that $sum' = \sum_{j=a}^{i'-1} entries.item(j)$. Since $sum' = sum + entries.item(i)$, and $i' = i + 1$,

$$\begin{aligned} sum' &= sum + entries.item(i) \\ &= \sum_{j=a}^{i-1} entries.item(j) + entries.item(i) \\ &= \sum_{j=a}^{i} entries.item(j) \\ &= \sum_{j=a}^{i'-1} entries.item(j) \end{aligned}$$

and so the condition holds at the beginning of the $(k + 1)$st iteration. □

Establishing the loop invariant is invariably the hard part of the proof, but there are two easier steps remaining. First, it must be shown that the postcondition holds at the end. Consider the last iteration of the loop in the summing algorithm. At the

end of it, the loop invariant holds, as has been shown. Then the test $i = b + 1$ is made, succeeds, and execution passes to the point after the loop. Clearly, at that moment the condition

$$sum = \sum_{j=a}^{i-1} entries.item(j) \text{ and } i = b + 1$$

holds. But this condition implies

$$sum = \sum_{j=a}^{b} entries.item(j)$$

which is the desired postcondition; so the postcondition holds when the algorithm terminates. Notice that this conclusion could not have been reached so simply if $i > b$ (the condition that is usually written in practice) had been used as the condition at the top of the loop. In general, just after the completion of the execution of the loop '**from** ... **until** *B* ...', with loop invariant *I*, the condition *I* **and** *B* holds, and it is necessary to prove that this implies *Post*.

The final step is to show that there is no risk of an infinite loop. This is usually obvious, so may be done briefly. The method of proof is to identify some integer quantity that is strictly increasing (or decreasing) from one iteration to the next, and to show that when this becomes sufficiently large (or small) the loop must terminate. For the summing algorithm, *i* is strictly increasing, and when it reaches $b + 1$, the loop must terminate. This argument depends on *i* being no greater than $b + 1$ initially; in other words, the condition $a \leq b + 1$ must be true initially in order for termination to be guaranteed.

To summarize, then, the steps required to prove that the iterative algorithm

```
-- Pre
from ... until B loop
  ...
end
-- Post
```

is correct are as follows:

1. Guess a condition *I*.
2. Prove by induction that *I* is a loop invariant.
3. Prove that *I* **and** $B \Rightarrow Post$.
4. Prove that the loop is guaranteed to terminate.

With practice, a clear intuitive understanding of the correctness of an algorithm will lead immediately to the loop invariant. Remember that the loop invariant must mention all the variables whose values change within the loop, but that it expresses

an unchanging relationship among those variables. It must also contain complete information about what the algorithm has achieved up to the point in the program text where it occurs.

For example, the loop invariant

$$sum = \sum_{j=a}^{i-1} entries.item(j)$$

makes good intuitive sense. It simply expresses the fact that, at the beginning of each iteration, *sum* contains the sum of all the values examined so far.

Some guidance on the general form of the loop invariant may be obtained from *Post*, since I must satisfy I **and** $B \Rightarrow Post$, where B and *Post* are known. In fact, it is good policy to take *Post* and generalize it in some way to obtain I. For example, in the summing algorithm above, I is just *Post* with b replaced by $i - 1$. This simple relationship ensures that the condition I **and** $B \Rightarrow Post$ is readily proved.

At the other extreme, check that the initialization statements establish I. If they do, and I **and** $B \Rightarrow Post$, it is probably worthwhile to proceed with the main part of the induction.

Correctness of selection sorting

Selection sorting is a simple sorting method which works by repeatedly finding the smallest item among those that remain unsorted, and adding it to the end of a growing sorted sequence. It makes an interesting example because there is a natural choice of loop invariant which turns out to be too weak, as will be seen.

Assuming that the items to be sorted are stored in an array *entries.item*(*a*..*b*), the sorting problem may be specified like this:

-- *Pre*: $a \le b + 1$
-- *Post*: *entries.item*(*a*..*b*) contains some permutation of its initial values
and $entries.item(a) \le entries.item(a + 1) \le \ldots \le entries.item(b)$

The precondition permits the array to be empty, when $a = b + 1$. The first part of the postcondition prevents an algorithm from changing (as distinct from moving) the items in the array; without it, a 'sorting' algorithm could replace every item by zero and declare the array to be sorted.

In the following algorithm, it is assumed that *min_index*(*entries*, *i*, *j*) returns the index of a smallest item in *entries.item*(*i*..*j*), and that *swap*(*entries*, *i*, *j*) exchanges items i and j. Here then is the selection sorting algorithm:

```
from i := a until i = b+1 loop
  j := min_index(entries, i, b);
  if j /= i then swap(entries, i, j) end;
  i := i + 1
end
```


On the first iteration of the loop, *min_index*(*entries*, *a*, *b*) finds an overall smallest item, and then *swap*(*entries*, *a*, *j*) swaps it into *entries.item*(*a*). On the second iteration of the loop, *min_index*(*entries*, *a*+1, *b*) finds a smallest remaining item, and *swap*(*entries*, *a*+1, *j*) swaps it into *entries.item*(*a*+1). This process continues until the array is sorted.

It should be clear that the first part of the postcondition, '*entries.item*(*a*..*b*) contains some permutation of its initial values,' may be incorporated into the loop invariant without change. It is trivially true at the beginning of the algorithm, and since the only changes to *entries* are those made by *swap*(*entries*, *i*, *j*), which permutes items but does not change them, it remains true throughout the entire execution of the algorithm.

It is clear informally that the purpose of one iteration is to get the correct item into *entries.item*(*i*); so at the beginning of this iteration,

$$entries.item(a) \le \ldots \le entries.item(i - 1)$$

and since this condition plus the termination condition $i = b + 1$ implies the remainder of the postcondition, it is a natural candidate for the remainder of the loop invariant. However, although this condition is true, it is inadequate: something is missing.

A simple way to see the problem is to consider the special case $i = a + 1$, that is, to consider the state of affairs at the beginning of the second iteration. The condition is

$$entries.item(a) \le \ldots \le entries.item(a)$$

which is vacuously true. But at this moment the loop invariant should be expressing everything that the algorithm has achieved, which in this case is to have swapped a smallest item into *entries.item*(*a*). We need to strengthen the loop invariant, which altogether comes to

entries.item(*a*..*b*) contains some permutation of its initial values
and $entries.item(a) \le \ldots \le entries.item(i - 1) \le entries.item(i..b)$

The last inequality means that the items of *entries.item*(*i*..*b*) are no smaller than the items of *entries.item*(*a*..*i* − 1).

The actual proof of correctness is now quite straightforward. The loop invariant clearly holds initially, since *entries.item*(*a*..*i* − 1) is empty, and when $i = b + 1$ it trivially implies the postcondition. The combined action of *min_index*(*entries*, *i*, *b*) and *swap*(*entries*, *i*, *j*) clearly produces

$$entries.item(a) \le \ldots \le entries.item(i) \le entries.item(i + 1..b)$$

and the final $i := i + 1$ returns us to the loop invariant.

Correctness of binary search

This section concludes with a study of the correctness of the following non-recursive binary search algorithm:

```
binary_search(a, b: INTEGER; x: KEY_TYPE): BOOLEAN is
    local
        i, j, mid: INTEGER;
        found: BOOLEAN;
    do
        found := false;
        from i := a; j := b until i = j + 1 or found loop
            mid := (i + j) // 2;
            if x = entries.item(mid) then
                found := true
            elseif x < entries.item(mid) then
                j := mid - 1
            else
                i := mid + 1
            end
        end;
        Result := found
    end
```

From the discussion of the recursive binary search algorithm in Section 1.2, it is fairly evident that the loop invariant should state that $x \in entries.item(a..b)$ if and only if $x \in entries.item(i..j)$. This takes care of the variables i and j.

The harder question is how to bring *found* and *mid* into the loop invariant, especially since *mid* is undefined at the beginning of the first iteration. Perhaps the best way to handle these two is to imagine another version of the algorithm in which we actually return the index of x if it is found. For this version it would be necessary to add $found \Rightarrow (a \le mid \le b$ **and** $x = entries.item(mid))$ to the postcondition, and this immediately suggests that it be included in the loop invariant:

($x \in entries.item(a..b)$ if and only if $x \in entries.item(i..j)$) **and**
($found \Rightarrow (a \le mid \le b$ **and** $x = entries.item(mid)$))

The initialization *found* := **false**; $i := a$; $j := b$; clearly establishes this invariant.

At termination, the loop invariant holds and so does $i = j + 1$ **or** *found*. If *found* is true, the loop invariant shows that $a \le mid \le b$ **and** $x = entries.item(mid)$, so it must be that $x \in entries.item(a..b)$; on the other hand, if *found* is false, then $i = j + 1$ and so $x \notin entries.item(i..j)$ and therefore $x \notin entries.item(a..b)$. Thus the postcondition holds.

The rest of the proof is left as an exercise; it is quite similar to the argument used to prove that the recursive binary search algorithm was correct.

1.4 Exercises

1.1 Consider the following recursive algorithm:

```
g(n: INTEGER): INTEGER;
    do
        if n <= 1 then
            Result := n
        else
            Result := 5*g(n-1) - 6*g(n-2)
        end
    end
```

Prove by induction on n that $g(n)$ returns $3^n - 2^n$ for all $n \ge 0$.

1.2 Prove that the specification

```
-- Pre:  a ≤ b + 1
-- Post: entries.item(a) ≤ entries.item(a + 1) ≤ ... ≤ entries.item(b)
```

is satisfied by the routine

```
selection_sort(a, b: INTEGER) is
    local
        i: INTEGER;
    do
        if a = b + 1 then
            -- do nothing
        else
            i := min_index(entries, a, b);
            if i /= a then swap(entries, i, a) end;
            selection_sort(a + 1, b)
        end
    end;
```

You may assume that *min_index*(*entries*, i, j) will return the index of a minimum element of the non-empty subarray $entries.item(i..j)$, and that *swap*(*entries*, i, a) swaps the two indicated elements.

1.3 It is not within the scope of this book to explain how program correctness may be reduced to a formal logical system. Here though is a glimpse. Consider an assignment statement:

```
-- Pre:  ?
x := E
-- Post: f(x)
```

where x is any variable, E is any expression, and $f(x)$ is any condition. Show that $f(E)$ must be true to guarantee the truth of the given postcondition.

1.4 Use the method of the previous question to show formally that the precondition **true** is sufficient to make the given postcondition true:

```
i := a;
sum := 0
-- Post: sum = Σ_{j=a}^{i-1} entries.item(j)
```

Then use the method to show formally that the postcondition shown below will be true where it occurs if that same condition is true at the beginning:

```
sum := sum + entries.item(i);
i := i + 1
-- Post: sum = Σ_{j=a}^{i-1} entries.item(j)
```

These two results taken together prove formally that the given condition is a loop invariant of the summing algorithm from Section 1.3.

1.5 Prove that the following linear search algorithm is correct with respect to the given precondition and postcondition:

```
-- Pre: a ≤ b and x ∈ entries.item(a..b)
from i := a until x = entries.item(i) loop
   i := i + 1
end
-- Post: a ≤ i ≤ b and x ∉ entries.item(a..i - 1)
      and x = entries.item(i)
```

1.6 This algorithm for evaluating the polynomial $a_0 + a_1x + \cdots + a_{k-1}x^{k-1}$ at the point $x = x_0$ is named after William G. Horner:

```
horner(a: ARRAY[INTEGER]; x0: INTEGER): INTEGER is
   local
      i: INTEGER;
   do
      Result := 0;
      from i := k - 1 until i < 0 loop
         Result := a.item(i) + Result * x0;
         i := i - 1
      end
   end
```

Find the loop invariant of this algorithm and prove it correct.

1.7 In addition to finding whether $x \in$ *entries.item*$(a..b)$, the following iterative version of binary search finds the index of the place where x lies, or, if x is not present, the index of the place just to the left of x's place in the ordering:


```
binary_search(a, b: INTEGER; x: KEY_TYPE): INTEGER is
   local
      i, j, mid: INTEGER;
      found: BOOLEAN;
   do
      found := false;
      from i := a; j := b until i = j + 1 or found loop
         mid := (i + j) // 2;
         if x = entries.item(mid) then
            found := true
         elseif x < entries.item(mid) then
            j := mid - 1
         else
            i := mid + 1
         end
      end;
      if found then Result := mid else Result := j end
   end
```

where // is integer division. Prove that this satisfies the specification

```
-- Pre:
   a ≤ b + 1 and entries.item(a) ≤ ... ≤ entries.item(b)
-- Post:
   (found ⇒ a ≤ Result ≤ b and entries.item(Result) = x) and
   (not found ⇒ a - 1 ≤ Result ≤ b and
      (for all k such that a ≤ k ≤ Result, entries.item(k) < x) and
      (for all k such that Result + 1 ≤ k ≤ b, x < entries.item(k)))
```

Your first task is to determine what must be true just after the loop terminates, in order for the final **if** statement to establish the postcondition.

1.8 *The bill-splitting problem* (J. McCormack). A group of p people living in a shared household receive a bill for c cents. They want to split the bill p ways as fairly as possible, given that fractions of one cent are not allowed. The following algorithm is proposed:

```
from m := c; n := p until n < 0 loop
   r := m // n;
   output.put_int(r);
   output.next_line;
   m := m - r;
   n := n - 1
end
```

where // is integer division. Does it always work?

1.9 *Tail recursion elimination.* Recursion is a powerful tool for expressing algorithms, and it is used extensively throughout this book. However, when a

recursive algorithm is heavily used in a production system, it may be worthwhile to tune it by eliminating some or all of the recursive calls. In general, recursion elimination requires the replacement of the runtime stack with an explicit stack appearing in the algorithm; but, in the case where there is only one recursive call, and it is the last statement in the body of the routine, the recursion can be replaced with a loop. This case is known as *tail recursion*. In general, the tail-recursive routine

```
tail_rec(x: instance_type): result_type is
    local
        y: instance_type;
    do
        if b(x) then
            Result := c(x)
        else
            y := d(x);
            Result := tail_rec(y)
        end
    end
```

has identical effect to the non-recursive routine

```
non_rec(x: instance_type): result_type is
    do
        from until b(x) loop
            x := d(x)
        end;
        Result := c(x)
    end
```

where $b(x)$, $c(x)$, and $d(x)$ are any functions of type *BOOLEAN*, *result_type*, and *instance_type* respectively. Prove this assertion by showing that the two routines execute identical statements in identical order.

1.10 Eliminate tail recursion from the *selection_sort* routine of Exercise 1.2.

1.11 Although the recursive binary search given in Section 1.2 is not in the form of a tail-recursive routine, since it has two recursive calls within its body it can be made tail-recursive by careful rewriting. Do this and compare the non-recursive version obtained by eliminating tail recursion with the version given in Section 1.3.

Chapter 2

Analysis of Algorithms

The speed of computation has increased so much over the past 40 years that it might seem that efficiency in algorithms is no longer important. But, paradoxically, efficiency matters more today than ever before. The obvious reason why this is so is that our ambition has grown with our computing power. Virtually all applications of computing – the simulation of continuous systems, high-resolution graphics, and the interpretation of physical data, for example – are demanding more speed.

The more subtle, and more important reason is as follows. The time that many algorithms take to execute is a non-linear function of the size of their input, and this can greatly reduce their ability to benefit from increases in speed. For example, consider an algorithm that sorts n numbers into increasing order in n^2 steps. Suppose that over the course of a few years, computing speed increases by a factor of 100. In the time that it used to take to execute the n^2 steps, it is now possible to execute $100n^2 = (10n)^2$ steps. Thus, only 10 times as many numbers can be sorted as before. One of the potential two orders of magnitude improvement has been lost to an inefficient algorithm.

Another example is the multiplication of integers. It takes longer to perform one 64-bit multiplication than it does to perform two 32-bit multiplications, as far as anyone knows.

The faster computers run, the more are efficient algorithms needed to take advantage of their power. The branch of computer science that studies efficiency is known as *analysis of algorithms*.

2.1 Characteristic operations and time complexity

Consider the following algorithm, which finds the index of a minimum element of the non-empty array *entries.item*(*a..b*):

```
min_index(a, b: INTEGER): INTEGER is
    local
        i: INTEGER
    do
        Result := a;
        from i := a + 1 until i > b loop
            if entries.item(i) < entries.item(Result) then
                Result := i
            end;
            i := i + 1
        end
    end
```

How long does *min_index*$(1, n)$ take to execute? The answer to this question depends on the particular implementation (that is, computer and compiler) used to execute the algorithm, and on the size of the array, n. Since our interest is in the algorithm itself, and not in any particular implementation of it, these two factors must be separated. In general, this will be done as follows. Choose some *characteristic operation* that the algorithm performs repeatedly. Define the *time complexity* $T(n)$ of an algorithm to be the number of characteristic operations it performs when given an input of size n.

For example, if the operation *Result* := a is chosen as the characteristic operation for *min_index*$(1, n)$, it turns out that $T(n) = 1$, since this operation is performed exactly once. Or, if the comparison *entries.item*(i) < *entries.item*(*Result*) is taken as the characteristic operation, $T(n)$ is the number of times the body of the loop is executed. It is not hard to see that this is $T(n) = n - 1$, since *entries.item*(*Result*) is compared once with each of the $n - 1$ numbers *entries.item*$(2..n)$. Finally, if *Result* := i is chosen as the characteristic operation, $T(n)$ could be anything from 0 to $n - 1$, depending on the values in the array: if the first entry is the smallest, $T(n) = 0$; if the first entry is the largest and then every entry after the first is smaller than the preceding one, $T(n) = n - 1$. This dependence on values will be considered further in the next example.

One way to choose among these answers is to refer to some particular implementation. Suppose it takes p microseconds to execute the body of the loop once, and q microseconds to execute the initialization and return parts. Then the execution time is $p(n - 1) + q$, which is $pT(n) + q$ if the second complexity function is chosen. A characteristic operation and its corresponding complexity function $T(n)$ are called *realistic* if the execution time with respect to some implementation is bounded by a linear function of $T(n)$. By choosing a realistic characteristic operation, the inherent complexity $T(n)$ is neatly separated from the implementation-dependent details p and q. The choice of a realistic characteristic operation is almost always so obvious that it rarely needs justification; in principle the analysis could be done for every possible operation and the largest answer taken.

Now consider the following algorithm for determining whether x is an element of *entries.item*$(a..b)$:


```
linear_search(a, b: INTEGER; x: KEY_TYPE): BOOLEAN is
    local
        i: INTEGER;
        found: BOOLEAN
    do
        found := false;
        from i := a until i > b or found loop
            found := (x = entries.item(i));
            i := i + 1
        end;
        Result := found
    end
```

The time complexity of *linear_search*$(1, n, x)$ depends on the value of x and on the contents of the array. This leads to two questions:

> Over all instances of size n, what is the maximum time the algorithm takes to execute? This is its *worst-case time complexity*, denoted $W(n)$.
>
> Over all instances of size n, what is the average time the algorithm takes to execute? This is its *average time complexity*, denoted $A(n)$.

More formally, suppose algorithm P accepts k different instances of size n. Let $T_i(n)$ be the time complexity of P when given the ith instance, for $1 \le i \le k$, and let p_i be the probability that this instance occurs. Then

$$W(n) = \max_{1 \le i \le k} T_i(n)$$

$$A(n) = \sum_{i=1}^{k} p_i T_i(n)$$

Incidentally, it follows that $A(n) \le W(n)$, with equality, assuming all the probabilities are non-zero, if and only if $T_1(n) = T_2(n) = \ldots = T_k(n)$ (Exercises 2.1 and 2.2).

Average complexity analysis is complicated by the need to find suitable values for the probabilities p_i. In one sense, any values would do, but if the result is to be useful the values must reflect the conditions under which the algorithm will be used – a hazy and subjective requirement. For some problems, no consensus on suitable probabilities has been reached (for example, graph problems).

Here now is the calculation of $W(n)$ and $A(n)$ for *linear_search*$(1, n, x)$, choosing x = *entries.item*(i) as the characteristic operation. Two reasonable assumptions are (a) the probability that x will be found somewhere in the array is a constant, p; and (b) if x is present, it is equally likely to be found at any position in the array.

Although *linear_search*$(1, n, x)$ has an infinite number of instances, they fall into just $k = n + 1$ classes. If x = *entries.item*(i), the algorithm will determine this and stop after comparing x with *entries.item*(1), *entries.item*(2), ..., *entries.item*(i); that is, after performing i characteristic operations. Since the probability p of x being

present is spread equally among the n cases $x = entries.item(1)$, $x = entries.item(2)$, ..., $x = entries.item(n)$, each must have probability p/n. If x is not present, then x is compared with all n elements of the array before stopping. This is all summarized in the following table:

i	*Instance*	p_i	$T_i(n)$
1	$x = entries.item(1)$	p/n	1
2	$x = entries.item(2)$	p/n	2
	...		
i	$x = entries.item(i)$	p/n	i
	...		
n	$x = entries.item(n)$	p/n	n
$n+1$	$x \notin entries.item(1..n)$	$1-p$	n

From this table, it is clear that *linear_search*$(1, n, x)$ has worst-case time complexity $W(n) = n$ comparisons. This occurs when $x = entries.item(n)$, and also when x is not present. The average complexity is

$$A(n) = \sum_{i=1}^{n+1} p_i T_i(n)$$

$$= \sum_{i=1}^{n} p_i T_i(n) + p_{n+1} T_{n+1}(n)$$

$$= \sum_{i=1}^{n} \frac{p}{n} i + (1-p)n$$

$$= \frac{p(n+1)}{2} + (1-p)n$$

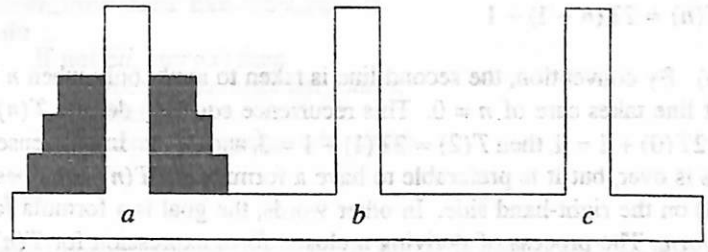
For example, if $p = 1$ the algorithm scans halfway along the array on average.

2.2 Recursive algorithms

In Section 1.2 it was shown how the correctness of a recursive algorithm is proved by induction on n, the size of its input. That approach allowed the recursive calls to be assumed correct; no investigation of them was needed.

A similar strategy often applies to the analysis of a recursive algorithm. By definition, for all n, $T(n)$ is the time complexity of the algorithm when given an input of size n. By definition, then, a recursive call of size $n/2$, say, has time complexity $T(n/2)$; no further investigation of it is needed. Just as the algorithm is defined in terms of itself, this approach will lead to an expression for $T(n)$ in terms of itself: a *recurrence equation* for $T(n)$, which must then be solved.

For example, consider the *Towers of Hanoi* problem, defined as follows. There are three pegs, labelled a, b, and c. On peg a there is a stack of n disks, each with a hole in the middle to accommodate the peg:



As the diagram shows, the disks increase in size going down. The problem is to transfer the stack of disks to peg c, one disk at a time, in such a way as to ensure that no disk is ever placed on top of a smaller disk. Here is a well-known algorithm for the Towers of Hanoi problem:

```
hanoi(n: INTEGER; from_peg, to_peg, spare_peg: CHARACTER) is
  do
    if n > 0 then
      hanoi(n–1, from_peg, spare_peg, to_peg);
      output.put_string("Move the top disk from peg ");
      output.put_character(from_peg);
      output.put_string(" to peg ");
      output.put_character(to_peg);
      output.next_line;
      hanoi(n–1, spare_peg, to_peg, from_peg)
    end
  end
```

Incidentally, it is the policy of this book to make the base of recursive algorithms as low as possible. In the case of *hanoi*, it makes sense to move zero disks from one peg to another. This policy invariably leads to simpler algorithms.

Let $T(n)$ be the time complexity of *hanoi*(n, x, y, z), when the characteristic operation is the printing of one line. Clearly, $T(0) = 0$, because the test $n > 0$ fails and nothing is printed. For larger n, the following statements are executed and costs incurred:

hanoi$(n–1, from\_peg, spare\_peg, to\_peg)$;	$T(n-1)$
output.put_string(...);	1
hanoi$(n–1, spare\_peg, to\_peg, from\_peg)$;	$T(n-1)$

As discussed above, the time complexity of *hanoi*$(n–1, x, y, z)$ is $T(n-1)$ by definition, and no further investigation of it is needed. (Many beginners to recursion stumble at this point; they persist in attempting to enter and investigate the recursive calls, when the whole point of recursion is that all needed information about the

recursive calls may be *assumed*, by induction.) Summing these contributions gives the *recurrence equation*

$$T(0) = 0$$
$$T(n) = 2T(n-1) + 1$$

for $T(n)$. By convention, the second line is taken to apply only when $n \geq 1$, since the first line takes care of $n = 0$. This recurrence equation defines $T(n)$ for all n: $T(1) = 2T(0) + 1 = 1$, then $T(2) = 2T(1) + 1 = 3$, and so on. In one sense, then, the analysis is over, but it is preferable to have a formula for $T(n)$ that does not have $T(n-1)$ on the right-hand side. In other words, the goal is a formula for $T(n)$ in *closed form*. The process of deriving a closed form expression for $T(n)$ is called *solving* the recurrence equation.

Of the variety of techniques for solving recurrence equations, only the simplest technique, *repeated substitution*, is used in this book. Since the formula $T(n) = 2T(n-1) + 1$ holds for all $n \geq 1$, it is valid to substitute $n-1$ for n in it to obtain $T(n-1) = 2T(n-2) + 1$ for all $n \geq 2$. Similarly, $T(n-2) = 2T(n-3) + 1$. Therefore

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2[2T(n-2) + 1] + 1 \\ &= 2[2[2T(n-3) + 1] + 1] + 1 \\ &= 2^3 T(n-3) + 2^2 + 2^1 + 2^0 \end{aligned}$$

(provided $n \geq 3$), expanding the brackets in a way that elucidates the emerging pattern. If this substitution is repeated i times, clearly the result is

$$T(n) = 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2^0$$

($n \geq i$). Induction on i could be used to prove this, but that is rarely necessary. By choosing i as large as possible, the base of the recurrence equation may be used to eliminate T from the right-hand side: if $i = n$, then $T(n-i) = T(0) = 0$. Hence

$$\begin{aligned} T(n) &= 2^n 0 + 2^{n-1} + 2^{n-2} + \cdots + 2^0 \\ &= \sum_{i=0}^{n-1} 2^i \\ &= 2^n - 1 \end{aligned}$$

applying the standard formula for the sum of a geometric progression. This completes the analysis of *hanoi*.

There are cases where the information needed for an analysis is in the data, not the code. Consider this algorithm from Section 6.2 for the inorder traversal of a binary tree:

```
inorder_traversal(x: like entry_type) is
    do
        if not nil_entry(x) then
            inorder_traversal(x.left_child);
            visit(x);
            inorder_traversal(x.right_child)
        end
    end;
```

where *visit*(*x*) stands for some operation to be performed at each node in the tree, such as printing its contents.

Choose *visit*(*x*) as the characteristic operation. It is performed once for each node in the tree being traversed, so $T(n) = n$, where n is the number of nodes. For want of a better term, this will be called the *global structure* approach to analysis. The global structure over which the algorithm travels is identified, and the number of characteristic operations performed is related to the size of this structure.

Analysis of binary search

This section ends with an example which shows how to deal with the practical difficulties and complications that often hinder analyses. The binary search algorithm, which was proved correct in Section 1.2, determines whether x is present in the sorted array *entries.item*(*a..b*):

```
binary_search(a, b: INTEGER; x: KEY_TYPE): BOOLEAN is
    local
        mid: INTEGER;
    do
        if a > b then
            Result := false
        else
            mid := (a + b) // 2;
            if x = entries.item(mid) then
                Result := true
            elseif x < entries.item(mid) then
                Result := binary_search(a, mid-1, x)
            else
                Result := binary_search(mid+1, b, x)
            end
        end
    end
```

The array size is halved after each comparison between x and *entries.item*(*mid*), roughly, and an array of length n can be halved only about $\log_2 n$ times before

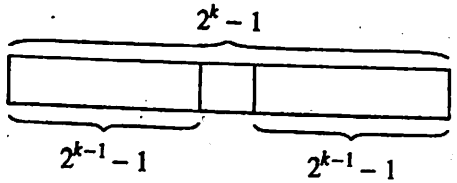
reaching a trivial length, so the worst-case complexity of *binary_search*$(1, n, x)$ is about $\log_2 n$.

A more precise analysis can be made using recurrence equations. Let $T(n)$ be the time complexity of *binary_search*$(1, n, x)$, where the characteristic operation is one comparison between x and *entries.item*(*mid*) (with a three-way outcome).

If $n > 0$, the algorithm begins by setting *mid* to $\lfloor (n+1)/2 \rfloor$[1] and examination of the program text reveals that

$$\begin{aligned} T(0) &= 0 \\ T(n) &= 1 && \text{if } x = \textit{entries.item}(\textit{mid}) \\ &= 1 + T(\lfloor (n+1)/2 \rfloor - 1) && \text{if } x < \textit{entries.item}(\textit{mid}) \\ &= 1 + T(n - \lfloor (n+1)/2 \rfloor) && \text{if } x > \textit{entries.item}(\textit{mid}) \end{aligned}$$

Although it is sometimes possible to solve messy recurrence equations like this one, in general it is better to make some simplifying assumptions. The first step in simplifying this recurrence is the elimination of the floor function, which can be done by restricting n to values of the form $n = 2^k - 1$, where k is a non-negative integer. This choice ensures that the array always breaks symmetrically into two equal pieces plus middle element:



Algebraically this is $\lfloor (n+1)/2 \rfloor = \lfloor (2^k - 1 + 1)/2 \rfloor = 2^{k-1}$ for $k \geq 1$, giving

$$\begin{aligned} T(0) &= 0 \\ T(2^k - 1) &= 1 && \text{if } x = \textit{entries.item}(\textit{mid}) \\ &= 1 + T(2^{k-1} - 1) && \text{if } x < \textit{entries.item}(\textit{mid}) \\ &= 1 + T(2^{k-1} - 1) && \text{if } x > \textit{entries.item}(\textit{mid}) \end{aligned}$$

A further simplification can be made by considering only the worst case, which by inspection occurs (for example) when the test $x = \textit{entries.item}(\textit{mid})$ always fails:

$$\begin{aligned} W(0) &= 0 \\ W(2^k - 1) &= 1 + W(2^{k-1} - 1) \end{aligned}$$

[1]The notation $\lfloor x \rfloor$, 'floor of x,' denotes the greatest integer less than or equal to x; the result of an integer division of one number by another is always truncated in this way. Similarly, $\lceil x \rceil$, 'ceiling of x,' is the smallest integer greater than or equal to x.

This may now be solved by repeated substitution:

$$\begin{aligned} W(2^k - 1) &= 1 + W(2^{k-1} - 1) \\ &= 1 + [1 + W(2^{k-2} - 1)] \\ &= 1 + [1 + [1 + W(2^{k-3} - 1)]] \\ &= \ldots \\ &= i + W(2^{k-i} - 1) \end{aligned}$$

$(i \leq k)$, and letting $i = k$ gives

$$\begin{aligned} W(2^k - 1) &= k + W(0) \\ &= k \end{aligned}$$

But now $2^k - 1 = n$, and $k = \log_2(n+1)$, so finally

$$W(n) = \log_2(n+1)$$

(for $n = 2^k - 1$), which concludes this analysis of binary search.

Although it might seem that the restriction to values of n of the form $2^k - 1$ weakens the result, in practice this does not matter very much: $W(n)$ is a monotone increasing function of n, and hence the formula given is a good approximation even when n is not of the form $2^k - 1$ (Exercise 2.15). In Exercise 6.10 it is shown that $W(n) = \lceil \log_2(n+1) \rceil$ for arbitrary n.

2.3 Iterative algorithms

Just as recursive algorithms lead naturally to recurrence equations, so iterative algorithms lead naturally to formulas involving summations.

The simplest iterative algorithms to analyze are those containing only loops that iterate over a fixed range of integers. The technique is based on the observation that in the code fragment

```
from i := a until i > b loop
    S;
    i := i + 1
end
```

the statement S is executed $b - a + 1$ times, provided that $a \leq b + 1$. In particular, if $a = b + 1$, S is executed zero times. (This also applies when counting downwards from b to a.)

For example, the *min_index* algorithm analyzed earlier in this chapter has this form. The loop executed by *min_index*(1, *n*) is

```
from i := 2 until i > n loop
   if entries.item(i) < entries.item(Result) then
      Result := i
   end;
   i := i + 1
end
```

By the observation about loops, the **if** statement and hence its comparison is made $n - 2 + 1 = n - 1$ times.

When analyzing an algorithm containing nested loops that iterate over a fixed range of integers in this way, it is generally best to begin with the innermost loop. For example, consider the following algorithm for adding two matrices *a.item*(1..*n*, 1..*m*) and *b.item*(1..*n*, 1..*m*) together:

```
from i := 1 until i > n loop
   from j := 1 until j > m loop
      c.put(a.item(i, j) + b.item(i, j), i, j);
      j := j + 1
   end;
   i := i + 1
end
```

A good choice for the characteristic operation here is *a.item*(*i*, *j*) + *b.item*(*i*, *j*), since it is characteristic of the algorithm and lies inside the inner loop. Having made this choice, the complexity of

```
from j := 1 until j > m loop
   c.put(a.item(i, j) + b.item(i, j), i, j);
   j := j + 1
end;
```

is clearly *m* additions, by the observation about loops. This reduces the problem to analyzing the outer loop, which now has the form

```
from i := 1 until i > n loop
   perform m additions;
   i := i + 1
end
```

During each of the *n* iterations of the outer loop, *m* additions are performed, giving a total of $T(n, m) = nm$ additions overall. The time complexity of the algorithm is a function of two parameters in this example, *n* and *m*.

In some algorithms, the cost of the inner loop depends on the value of the index variable of the outer loop. Consider this algorithm for sorting the elements of the array *entries.item*(*a*..*b*) into non-decreasing order:


```
bubble_sort(a, b: INTEGER) is
   local
      i, j: INTEGER;
   do
      from i := b until i < a loop
         from j := a + 1 until j > i loop
            if entries.item(j–1) > entries.item(j) then
               swap(entries, j–1, j)
            end;
            j := j + 1
         end;
         i := i – 1
      end
   end
```

It works by 'bubbling' the largest entry up to *entries.item*(*b*), the second largest to *entries.item*(*b*–1), and so on. The outer loop's invariant is '*entries.item*(*a*..*b*) contains a permutation of its original contents, and *entries.item*(*i*+1..*b*) contains the $b - i$ largest entries, in sorted order.'

Take the comparison *entries.item*(*j*–1) > *entries.item*(*j*) as the characteristic operation, and consider the analysis of *bubble_sort*(1, *n*). As before, the first step is to take the inner loop in isolation:

```
from j := 2 until j > i loop
   if entries.item(j–1) > entries.item(j) then
      swap(entries, j–1, j)
   end;
   j := j + 1
end
```

where *a*+1 has been replaced by 2. Applying the observation about loops over fixed ranges shows that the operation *entries.item*(*j*–1) > *entries.item*(*j*) is performed $i - 1$ times, and so the problem reduces to analyzing the outer loop

```
from i := n until i < 1 loop
   perform i–1 comparisons;
   i := i – 1
end
```

where *b* has been replaced by *n* and *a* by 1. The first iteration, when $i = n$, has a cost of $n - 1$ comparisons; the second iteration, when $i = n - 1$, has a cost of $n - 2$ comparisons. Continuing in this way, the total cost is $(n - 1) + (n - 2) + \cdots + 0$ comparisons, so

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

Compared with the algorithms of Chapter 9, *bubble_sort* is highly inefficient.

Incidentally, it is the policy of this book to produce algorithms that work correctly on the smallest possible input. It makes sense to sort an empty array (that is, an array of length zero); accordingly, this version of *bubble_sort* is well defined when $a = b + 1$, and the analysis is correct too.

As with recursive algorithms, sometimes the information needed for the analysis is in the data, not the code. In such cases the global structure method introduced in the previous section is needed: identify the global structure, and relate $T(n)$ to its size.

Perhaps the simplest example is the traversal of a linked list, using this code taken from Section 5.1:

```
from x := l.first until l.nil_entry(x) loop
    visit(x.value);
    x := l.next(x)
end
```

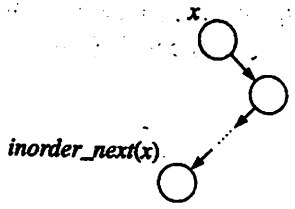
Using any reasonable linked list implementation, the cost of both *l.first* and *l.next*(x) will be $O(1)$, and so *visit*(*x.value*) is a realistic characteristic operation. It is clearly performed once for each entry of the list, so the time complexity of traversing a list containing n items is just n.

Tree traversal

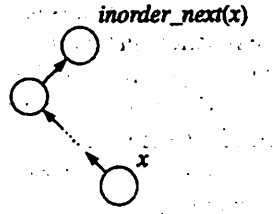
For a more complex example of the global structure method just introduced, consider the non-recursive implementation of the inorder traversal of a binary tree, as presented in Figure 6.2 (the recursive version was analyzed in Section 2.2). The idea is to proceed from the first node of the traversal to its successor, to the next successor, and so on:

```
from x := t.inorder_first until t.nil_entry(x) loop
    visit(x);
    x := t.inorder_next(x)
end
```

Parent references are needed in addition to the usual left child and right child links. As explained in detail in Section 6.2, if x has a right child, its successor in the inorder traversal is

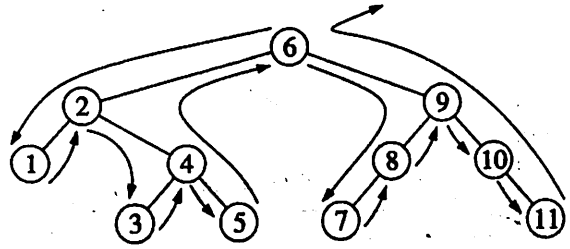


If x has no right child, its successor is



If, following the recursive analysis, *visit*(x) is chosen as the characteristic operation, it follows immediately that $T(n) = n$, the number of nodes in the tree. But it is not clear that this choice is realistic, because *visit*(x) does not lie inside the inner loops of the code just given.

Another characteristic operation, which *is* realistic, is the *edge-traverse*. An edge-traverse is the movement of the algorithm's attention across one link, which could be by any one of the operations $y := t.left\_child(x)$, $y := t.right\_child(x)$, or $y := t.parent(x)$. Examination of examples such as



shows that every link is traversed exactly twice: once on the way down, and once on the way back. The final traverse out of the root may be ignored. If $n \geq 1$, there are $n - 1$ edges in an n-node binary tree (Exercise 6.1), so $T(n) = 2(n - 1)$. Note that the algorithm treats $n = 0$ as a special case, and accordingly the analysis must do so too: $T(0) = 0$.

2.4 Evaluating efficiency, and the *O*-notation

A number of algorithms have been analyzed here, but so far no opinion has been expressed about their efficiency. Consider the Towers of Hanoi algorithm, for example, whose worst-case time complexity was shown in Section 2.2 to be $T(n) = 2^n - 1$ lines of output. Is this an efficient algorithm?

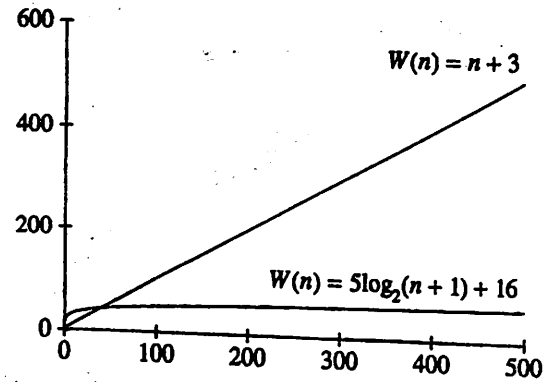
Clearly, the Towers of Hanoi algorithm is not efficient. When given the small instance $n = 10$, the algorithm produces 1023 lines of output; when given the instance $n = 20$, it produces 1 048 575 lines.

Even if its time complexity were $T(n) = 2^n - 100$, or $T(n) = 2^n/100$, the Towers of Hanoi algorithm would still be inefficient. A general assessment of an algorithm's efficiency does not depend very strongly on constant factors in the time

complexity function, unless they are unusually large or small. What is important is the general form of the function, in this case 2^n.

Similarly, when two algorithms for the same problem are compared, the general form of their complexity functions is usually sufficient to determine which is best. For example, consider a comparison of linear search, of worst-case time complexity $W(n) = n$ comparisons, with binary search, of worst-case time complexity $W(n) = \log_2(n + 1)$ comparisons.

It could be argued that this comparison is unfair, since more work is done in binary search per comparison than is done in linear search. This objection can be answered by first adding implementation-dependent constants so as to express the complexities in microseconds. The result might be $W(n) = n + 3$ microseconds for linear search, and $W(n) = 5\log_2(n + 1) + 16$ microseconds for binary search, say. Here is a graph of these two functions:



It shows that for n less than about 40, linear search is superior to binary search; for larger n, binary search becomes dramatically more efficient: when $n = 500$, for example, binary search is about eight times faster than linear search given these constant factors.

The crossover point is rather sensitive to the values of the constant factors, but the superiority of binary search for large n is not, since it is a well-known fact that for any positive constants a, b, c, and d, the quotient

$$\frac{a\log_2(n + 1) + b}{cn + d}$$

approaches 0 as n increases. That is, even without knowing the values of the constant factors, it can be concluded that binary search will be superior for sufficiently large n; and unless some exceptionally large constant factors are present, binary search would be preferred for this reason.

The *O*-notation

A convenient way to express the general form of a function is provided by the *O-notation*. For example, one may say that the time complexity of binary search is $O(\log n)$, pronounced 'big oh of log n,' meaning that it has the general form of $\log n$. The term *asymptotic time complexity* is also used for the general form of a time complexity function.

The technical definition of the O-notation is less important than the idea, explained above, that the asymptotic time complexity determines our evaluation of the efficiency of an algorithm. For the record, however, here is the definition:

Definition 2.1: The notation $O(f(n))$, appearing in a formula, stands for a quantity x_n which may not be explicitly known but which is known to satisfy $|x_n| \le M|f(n)|$ for all $n \ge n_0$, where M and n_0 are fixed constants.

Let us take the simplest example, $O(1)$. According to the definition, this stands for a quantity x_n which satisfies $|x_n| \le M$ for all $n \ge n_0$. In other words, $O(1)$ is a quantity which is no larger than some fixed constant, whose precise value is not stated. It succinctly describes the time complexity of any algorithm containing no loops and no procedure calls; such an algorithm takes at most some constant amount of time to execute.

Here is another example:

$$\lceil \log_2(n + 1) \rceil = O(\log_2 n)$$

This follows from the definition above by letting $x_n = \lceil \log_2(n + 1) \rceil$, $f(n) = \log_2 n$, $M = 2$, and $n_0 = 2$, for then it is indeed true that $|x_n| \le M|f(n)|$ for all $n \ge n_0$. It is very convenient to be able to describe binary search as 'an $O(\log_2 n)$ algorithm,' presenting just the essential fact of its asymptotic time complexity.

It turns out that the base of a logarithm is irrelevant inside O. In other words, if $x_n = O(\log_a n)$, then $x_n = O(\log_b n)$ for any bases a and b greater than 1. The proof of this is not difficult; it uses the formula $\log_b n = \log_a n / \log_a b$, which shows that the two logarithms only differ by the constant factor $\log_a b$. For this reason, the base of a logarithm is usually omitted inside O.

Various useful, if obvious, theorems may be proved about the O-notation. For example, if $x_n = O(f(n))$ and $y_n = O(g(n))$, then $x_n y_n = O(f(n)g(n))$. The proofs of these theorems have been left as an exercise (Exercise 2.17); they all follow easily from the formal definition.

Since the definition of the O-notation is based on the condition $|x_n| \le M|f(n)|$, it is allowable for $f(n)$ to be a much faster-growing function than x_n. For example,

$$\lceil \log_2(n + 1) \rceil = O(n^2)$$

is a true statement, although not a very useful one.

This looseness in the definition of the O-notation does have one virtue, however. Suppose that analysis reveals that for some algorithm,

$$T(n) = \sum_{i=1}^{n} \lfloor n/i \rfloor$$

The analyst might then proceed to estimate $T(n)$ as follows. Since $\lfloor n/i \rfloor \leq n$,

$$T(n) \leq \sum_{i=1}^{n} n = n^2$$

and this may be correctly reported as $T(n) = O(n^2)$. In fact, it is also true that $T(n) = O(n \log n)$, but more advanced algebra is needed to prove this, and it is not always possible to find the lowest general form in this way. In such cases, an upper bound like the $O(n^2)$ one just given is the next best thing, and the O-notation is ideal for reporting it, without making any claim that the result is the best possible.

The Ω and Θ notations

The Ω-notation is defined very similarly to the O-notation:

Definition 2.2: The notation $\Omega(f(n))$, pronounced 'big omega of $f(n)$,' appearing in a formula, stands for a quantity x_n which may not be explicitly known but which is known to satisfy $|x_n| \geq M|f(n)|$ for all $n \geq n_0$, where M and n_0 are fixed constants.

The only change here is the replacement of '$|x_n| \leq M|f(n)|$' by '$|x_n| \geq M|f(n)|$,' and so it follows that $g(n) = \Omega(f(n))$ if and only if $f(n) = O(g(n))$.

The Ω-notation is used in work on lower bounds, to state that every algorithm for some problem must require at least a certain amount of time to execute. For example, we might state that every algorithm for searching an unsorted sequence to determine whether some item x is present must take $\Omega(n)$ time.

It is perhaps worth pointing out that the statement $W(n) = \Omega(f(n))$ does not mean that every instance of some problem takes at least on the order of $f(n)$ time to solve; rather, it means that for all n there exists an instance which takes this long. For example, the complexity of linear search is $\Omega(n)$, since when x is not found the entire list must be searched; but there are instances of arbitrarily large size which require only $O(1)$ time: those where x is present at the front of the list.

Finally, $g(n) = \Theta(f(n))$, 'big theta of $f(n)$,' means that $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$. It is perhaps surprising that the Θ-notation is not used more widely in such statements as 'binary search is $\Theta(\log n)$ in the worst case,' but the O-notation is much older and more widely used than the other two notations, and is used customarily despite its relative lack of precision.

Problems for which $O(n \log n)$ algorithms exist are said to be *feasible*, meaning that large instances can be solved. Problems for which all algorithms are $\Omega(a^n)$

where a is any constant strictly greater than 1, are *infeasible*, because even on the fastest computing equipment available today or in the foreseeable future, it will be possible to solve only small instances. The feasibility of problems with intermediate complexities, such as $O(n^2)$ and $O(n^3)$, will depend on the size of the instance to be solved. This is illustrated in the following table, which shows the largest instance solvable in a given time, for various complexity functions.

Complexity (microseconds)	*Largest instance solvable in one second*	*Largest instance solvable in one day*	*Largest instance solvable in one year*
$W(n) = n$	$n = 1\,000\,000$	$n = 86\,400\,000\,000$	$n = 31\,536\,000\,000\,000$
$W(n) = n \log_2 n$	$n = 62\,746$	$n = 2\,755\,147\,514$	$n = 798\,160\,978\,500$
$W(n) = n^2$	$n = 1\,000$	$n = 293\,938$	$n = 5\,615\,692$
$W(n) = n^3$	$n = 100$	$n = 4\,421$	$n = 31\,593$
$W(n) = 2^n$	$n = 19$	$n = 36$	$n = 44$

2.5 Exercises

2.1 Using the formal definitions of $W(n)$ and $A(n)$ given in Section 2.1, prove that $A(n) \leq W(n)$.

2.2 Assuming that all the p_i are non-zero, show that $A(n) = W(n)$ if and only if $T_1(n) = \ldots = T_k(n)$.

2.3 Consider the following algorithm for the linear search of a sorted array *entries.item*(*a*..*b*). The algorithm employs a *sentinel*: it adds x to the end of *entries* before commencing, so as to simplify the search.

```
sorted_search(a, b: INTEGER; x: KEY_TYPE): BOOLEAN is
  local
    i: INTEGER
  do
    entries.put(x, b+1);
    from i := a until entries.item(i) >= x loop
      i := i + 1
    end;
    Result := (i <= b) and (x = entries.item(i))
  end
```

(a) Choose a characteristic operation. Construct a table of cases for the analysis, similar to the table given in Section 2.1. What is the worst-case complexity of *sorted_search*(1, *n*, *x*)?

(b) Making reasonable assumptions about the probabilities of the cases, determine the average complexity of *sorted_search*(1, *n*, *x*).

2.4 Consider the following algorithm for finding the index of a minimum element of *entries.item(a..b)*:

```
min_index(a, b: INTEGER): INTEGER is
    do
        if a = b then
            Result := a
        else
            Result := min_index(a + 1, b);
            if entries.item(a) < entries.item(Result) then
                Result := a
            end
        end
    end
```

(a) Show that, if $T(n)$ is the number of comparisons of the form *entries.item(a)* < *entries.item(Result)* made by *min_index*$(1, n)$, then

$$T(1) = 0$$
$$T(n) = 1 + T(n-1)$$

(b) Solve this recurrence by repeated substitution. How does the performance of this version of *min_index* compare with the one given in Section 2.1?

(c) What is the average number of times that the operation *Result* := *a* will be performed, counting recursive calls? (Hint: it will be performed only when *entries.item(a)* is a minimum element of *entries.item(a..b)*. What is a reasonable probability to assign to this event?)

2.5 The following version of binary search is often preferred because it has a two-way rather than a three-way branch, and so is more amenable to efficient compilation. It assumes that the array to be searched is non-empty.

```
binary_search(a, b: INTEGER; x: KEY_TYPE): BOOLEAN is
    local
        mid: INTEGER;
    do
        if a = b then
            Result := (x = entries.item(a))
        else
            mid := (a + b) // 2;
            if x <= entries.item(mid) then
                Result := binary_search(a, mid, x)
            else
                Result := binary_search(mid+1, b, x)
            end
        end
    end
```


Analyze this algorithm, taking as your measure of complexity the number of comparisons between x and elements of *entries*.

2.6 The *Fibonacci numbers* are defined by the recurrence equation

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

The first few numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... Unfortunately, the recurrence equation cannot be solved by repeated substitution; a more advanced technique, the use of generating functions, is required. Prove by induction on n that

$$\phi^{n-2} \le F(n) \le \phi^{n-1}$$

for all $n \ge 2$, where $\phi = (1+\sqrt{5})/2 \approx 1.6180339$. You should begin by showing that $\phi^2 = \phi + 1$, and use this identity to prove the result.

2.7 Consider the following algorithm for calculating $F(n)$, as defined in the previous question:

```
fib(n: INTEGER): INTEGER is
    do
        if n <= 1 then
            Result := n
        else
            Result := fib(n-1) + fib(n-2)
        end
    end
```

Choosing *Result* := *n* as characteristic operation, show that $T(n) = F(n+1)$, and use the previous question to conclude that $T(n) = \Theta(\phi^n)$. (A more obviously realistic characteristic operation is *n* <= 1, and analyzing it is an interesting but more difficult exercise.)

2.8 *Euclid's algorithm.* The following algorithm, for finding the greatest common divisor of two positive integers, is similar to one given by Euclid:

```
gcd(n, m: INTEGER): INTEGER is
    do
        if m = 0 then
            Result := n
        else
            Result := gcd(m, n \\ m)
        end
    end
```

where $n \setminus\setminus m$ is the remainder after division of n by m. The correctness of Euclid's algorithm follows from a theorem in number theory, and it will not be explored here.

(a) Show that the number of $\setminus\setminus$ operations performed by this algorithm is given by the recurrence equation

$$T(n,0) = 0$$
$$T(n,m) = 1 + T(m, n \setminus\setminus m)$$

(b) Solve this recurrence equation for the special case of the Fibonacci numbers defined in the previous question; that is, letting $n = F(k+1)$ and $m = F(k)$.

(c) Show that, for all integers n and m such that $n \geq m > 0$,

$$n + m \geq \frac{3}{2}(m + n \setminus\setminus m)$$

(Hint: let $n = am + b$ where $a \geq 1$ and $0 \leq b < m$, and then consider the fraction $(n+m)/(m+n \setminus\setminus m)$.

(d) Use (c) to prove by induction on m that $T(n,m) \leq \log_{3/2}(n+m)$ provided $n \geq m > 0$.

2.9 Solve the following recurrence equations; c is a constant, and where necessary you may assume that $n = 2^k$.

(a) $T(0) = 1$
$T(n) = cT(n-1)$

(b) $T(0) = 1$
$T(n) = nT(n-1)$

(c) $T(1) = 1$
$T(n) = 1 + T(\lfloor n/2 \rfloor)$

(d) $T(1) = 1$
$T(n) = 1 + 2T(\lfloor n/2 \rfloor)$

(e) $T(1) = 0$
$T(n) = c\lceil \log_2 n \rceil + T(\lceil n/2 \rceil)$

(f) $T(1) = 0$
$T(n) = \lceil n \log_2 n \rceil + T(\lceil n/2 \rceil)$

2.10 Show that the solution of the two-dimensional recurrence equation

$$T(n,0) = 1$$
$$T(0,m) = 1$$
$$T(n,m) = T(n-1,m) + T(n,m-1)$$

is the binomial coefficient

$$T(n,m) = \binom{n+m}{n}$$

2.11 Consider the following algorithm for multiplying two n by n matrices a and b, placing the result in *Result*:

```
matrix_multiply(a, b: MATRIX): MATRIX is
    local
        i, j, k, total: INTEGER;
    do
        from i := 1 until i > n loop
            from j := 1 until j > n loop
                total := 0;
                from k := 1 until k > n loop
                    total := total + a.item(i, k)*b.item(k, j);
                    k := k + 1
                end;
                Result.put(total, i, j);
                j := j + 1
            end;
            i := i + 1
        end
    end
```

If the characteristic operation is one multiplication of matrix elements, what is the time complexity of this algorithm as a function of n?

2.12 Consider the following algorithm for sorting the array *entries.item(a..b)* into non-decreasing order:

```
straight_selection_sort(a, b: INTEGER) is
    local
        i, m: INTEGER;
    do
        from i := a until i >= b loop
            m := min_index(i, b);
            if i /= m then swap(entries, i, m) end;
            i := i + 1
        end
    end
```

where *swap(entries, i, m)* swaps *entries.item(i)* with *entries.item(m)*. Taking

as your measure of complexity one comparison between elements of *entries*, as occurs within *min_index* (Exercise 2.4), what is the complexity of this algorithm?

2.13 Consider the following algorithm for finding both the index of a minimum element and the index of a maximum element of the array *e.item*(*a*..*b*). Routine *is_odd* returns true when its parameter is an odd number.

```
min_max(a, b: INTEGER) is
    local
        i: INTEGER
    do
        if is_odd(b–a+1) then
            min := a;
            max := a;
            i := a+1
        elseif e.item(a) < e.item(a+1) then
            min := a;
            max := a+1;
            i := a+2
        else
            min := a+1;
            max := a;
            i := a+2
        end;
        from until i > b loop
            if e.item(i) < e.item(i+1) then
                if e.item(i) < e.item(min) then min := i end;
                if e.item(i+1) > e.item(max) then max := i+1 end
            else
                if e.item(i+1) < e.item(min) then min := i+1 end;
                if e.item(i) > e.item(max) then max := i end
            end;
            i := i + 2
        end
    end
```

(a) Explain how this algorithm works, clearly indicating the differences between the odd and even cases. Then give a formal loop invariant.

(b) Analyze *min_max*(1, *n*) for arbitrary $n \geq 1$, using one comparison between elements of *e* as the characteristic operation, and show that its time complexity is

$$T(n) = \left\lceil \frac{3n}{2} \right\rceil - 2$$

Recall that $\lceil x \rceil$ ('ceiling of x') means the smallest integer greater than or equal to x. (Hint: consider the cases n even and n odd separately.)

2.14 The following algorithm solves a standard problem encountered by text editors. It searches for the first occurrence of the string (that is, array of characters) *pattern.item*(1..*m*) within the string *text.item*(1..*n*), returning the index where *pattern* begins if found, or else zero. The value *limit* = *n*–*m*+1 is the rightmost place in *text* where *pattern* could possibly begin.

```
string_search(text, pattern: ARRAY[CHARACTER]; n, m: INTEGER):
        BOOLEAN is
    local
        found: BOOLEAN;
        start, i, j, limit: INTEGER;
    do
        found := false;
        limit := n – m + 1;
        from start := 1 until found or start > limit loop
            from
                i := start; j := 1
            until
                j = m+1 or else text.item(i) /= pattern.item(j)
            loop
                i := i + 1;
                j := j + 1
            end;
            found := (j = m + 1);
            start := start + 1
        end;
        if found then
            Result := start – 1
        else
            Result := 0
        end
    end
```

How many times is the comparison *text.item*(*i*) /= *pattern.item*(*j*), which is a realistic characteristic operation, performed in the worst case?

2.15 Prove, by induction on n, that the function $W(n)$ defined by

$$W(0) = 0$$
$$W(n) = 1 + W(\lfloor (n+1)/2 \rfloor - 1)$$

is monotone non-decreasing, or in other words that $W(n) \leq W(n+1)$ for all n. You may assume that $\lfloor x \rfloor$ is a monotone non-decreasing function of x.

2.16 Suppose you have one hour of computer time each evening to run a certain program. You find that the hour is exactly long enough for your program to process an input of size $n = 1\,000\,000$. Then your employer buys a computer which runs one hundred times faster than the old one. How large an input

will your program handle in one hour now, if its complexity $T(n)$ is, for some constants k_i:

(a) $k_1 n$

(b) $k_2 n \log_{10} n$

(c) $k_3 n^2$

(d) $k_4 n^3$

(e) $k_5 10^n$

2.17 Assuming that $x_n = O(f(n))$ and $y_n = O(g(n))$, use the formal definition of the O-notation to prove that

(a) $cx_n = O(f(n))$ for any constant c

(b) $x_n y_n = O(f(n)g(n))$

(c) $x_n + y_n = O(f(n) + g(n)) = O(\max(f(n), g(n))$

In the last part, $f(n)$ and $g(n)$ must be non-negative.

2.18 Use the results of the previous question to prove that

$$a_0 + a_1 n + a_2 n^2 + \cdots + a_k n^k = O(n^k)$$

2.19 Does $2^n = \Theta(3^n)$?

CHAPTER 3

Lists, Stacks, and Queues

This chapter discusses three of the most simple and basic data structures. Virtually every significant program will use at least one of these structures explicitly, and a stack is always implicitly used in a program, whether or not you declare one. Among the highlights of this chapter, we will

- Introduce the concept of Abstract Data Types (ADTs).
- Show how to efficiently perform operations on lists.
- Introduce the stack ADT and its use in implementing recursion.
- Introduce the queue ADT and its use in operating systems and algorithm design.

Because these data structures are so important, one might expect that they are hard to implement. In fact, they are extremely easy to code up; the main difficulty is maintaining enough discipline to write good general-purpose code for routines that are generally only a few lines long.

3.1. Abstract Data Types (ADTs)

One of the basic rules concerning programming is that no routine should ever exceed a page. This is accomplished by breaking the program down into *modules*. Each module is a logical unit and does a specific job. Its size is kept small by calling other modules. Modularity has several advantages. First, it is much easier to debug small routines than large routines. Second, it is easier for several people to work on a modular program simultaneously. Third, a well-written modular program places certain dependencies in only one routine, making changes easier. For instance, if output needs to be written in a certain format, it is certainly important to have one routine to do this. If printing statements are scattered throughout the program, it will take considerably longer to make modifications. The idea that global variables and side effects are bad is directly attributable to the idea that modularity is good.

An *abstract data type* (ADT) is a set of operations. Abstract data types are mathematical abstractions; nowhere in an ADT's definition is there any mention of

how the set of operations is implemented. This can be viewed as an extension of modular design.

Objects such as lists, sets, and graphs, along with their operations, can be viewed as abstract data types, just as integers, reals, and booleans are data types. Integers, reals, and booleans have operations associated with them, and so do abstract data types. For the set ADT, we might have such operations as *union, intersection, size,* and *complement.* Alternatively, we might only want the two operations *union* and *find,* which would define a different ADT on the set.

The basic idea is that the implementation of these operations is written once in the program, and any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate function. If for some reason implementation details need to be changed, it should be easy to do so by merely changing the routines that perform the ADT operations. This change, in a perfect world, would be completely transparent to the rest of the program.

There is no rule telling us which operations must be supported for each ADT; this is a design decision. Error handling and tie breaking (where appropriate) are also generally up to the program designer. The three data structures that we will study in this chapter are primary examples of ADTs. We will see how each can be implemented in several ways, but if they are done correctly, the programs that use them will not need to know which implementation was used.

3.2. The List ADT

We will deal with a general list of the form $A_1, A_2, A_3, \ldots, A_N$. We say that the size of this list is N. We will call the special list of size 0 an *empty list.*

For any list except the empty list, we say that A_{i+1} follows (or succeeds) A_i ($i < N$) and that A_{i-1} precedes A_i ($i > 1$). The first element of the list is A_1, and the last element is A_N. We will not define the predecessor of A_1 or the successor of A_N. The *position* of element A_i in a list is i. Throughout this discussion, we will assume, to simplify matters, that the elements in the list are integers, but in general, arbitrarily complex elements are allowed.

Associated with these "definitions" is a set of operations that we would like to perform on the list ADT. Some popular operations are *PrintList* and *MakeEmpty,* which do the obvious things; *Find,* which returns the position of the first occurrence of a key; *Insert* and *Delete,* which generally insert and delete some key from some position in the list; and *FindKth,* which returns the element in some position (specified as an argument). If the list is 34, 12, 52, 16, 12, then *Find*(52) might return 3; *Insert*(X, 3) might make the list into 34, 12, 52, X, 16, 12 (if we insert after the position given); and *Delete*(52) might turn that list into 34, 12, X, 16, 12.

Of course, the interpretation of what is appropriate for a function is entirely up to the programmer, as is the handling of special cases (for example, what does *Find*(1) return above?). We could also add operations such as *Next* and *Previous,* which would take a position as argument and return the position of the successor and predecessor, respectively.

3.2.1. Simple Array Implementation of Lists

All of these instructions can be implemented just by using an array. Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high overestimate, which wastes considerable space. This could be a serious limitation, especially if there are many lists of unknown size.

An array implementation allows *PrintList* and *Find* to be carried out in linear time, which is as good as can be expected, and the *FindKth* operation takes constant time. However, insertion and deletion are expensive. For example, inserting at position 0 (which amounts to making a new first element) requires first pushing the entire array down one spot to make room, whereas deleting the first element requires shifting all the elements in the list up one, so the worst case of these operations is $O(N)$. On average, half of the list needs to be moved for either operation, so linear time is still required. Merely building a list by N successive inserts would require quadratic time.

Because the running time for insertions and deletions is so slow and the list size must be known in advance, simple arrays are generally not used to implement lists.

3.2.2. Linked Lists

In order to avoid the linear cost of insertion and deletion, we need to ensure that the list is not stored contiguously, since otherwise entire parts of the list will need to be moved. Figure 3.1 shows the general idea of a *linked list.*

The linked list consists of a series of structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a structure containing its successor. We call this the *Next* pointer. The last cell's *Next* pointer points to *NULL;* this value is defined by C and cannot be confused with another pointer. ANSI C specifies that *NULL* is zero.

Recall that a pointer variable is just a variable that contains the address where some other data are stored. Thus, if P is declared to be a pointer to a structure, then the value stored in P is interpreted as the location, in main memory, where a structure can be found. A field of that structure can be accessed by *P->FieldName,* where *FieldName* is the name of the field we wish to examine. Figure 3.2 shows the actual representation of the list in Figure 3.1. The list contains five structures, which happen

Figure 3.1 A linked list

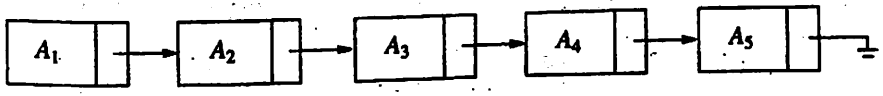
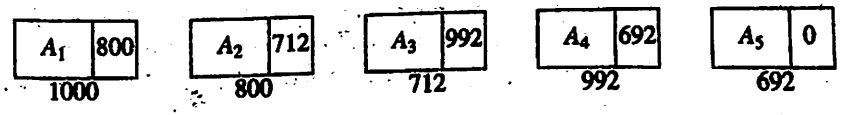


Figure 3.2 Linked list with actual pointer values



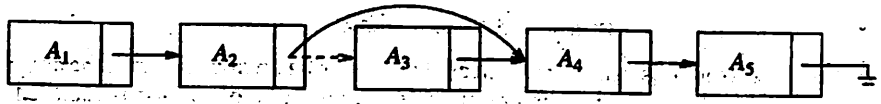


Figure 3.3 Deletion from a linked list

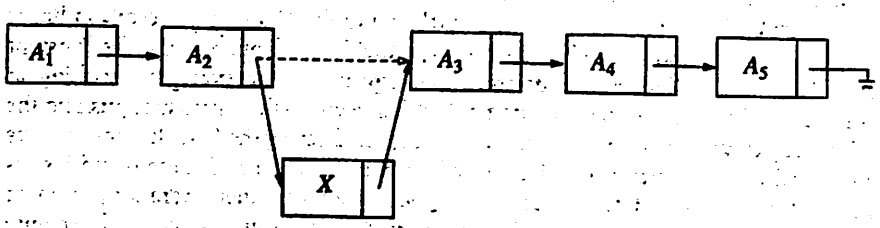


Figure 3.4 Insertion into a linked list

to reside in memory locations 1000, 800, 712, 992, and 692, respectively. The *Nex* pointer in the first structure has the value 800, which provides the indication o where the second structure is. The other structures each have a pointer that serve a similar purpose. Of course, in order to access this list, we need to know wher the first cell can be found. A pointer variable can be used for this purpose. It i important to remember that a pointer is just a number. For the rest of this chapter we will draw pointers with arrows, because they are more illustrative.

To execute *PrintList(L)* or *Find(L,Key)*, we merely pass a pointer to the firs element in the list and then traverse the list by following the *Next* pointers. Thi operation is clearly linear-time, although the constant is likely to be larger than i an array implementation were used. The *FindKth* operation is no longer quite a efficient as an array implementation; *FindKth(L,i)* takes $O(i)$ time and works b traversing down the list in the obvious manner. In practice, this bound is pessimistic because frequently the calls to *FindKth* are in sorted order (by i). As an example *FindKth(L,2)*, *FindKth(L,3)*, *FindKth(L,4)*, and *FindKth(L,6)* can all be executed ir one scan down the list.

The *Delete* command can be executed in one pointer change. Figure 3.3 show the result of deleting the third element in the original list.

The *Insert* command requires obtaining a new cell from the system by usin a *malloc* call (more on this later) and then executing two pointer maneuvers. Th general idea is shown in Figure 3.4. The dashed line represents the old pointer.

3.2.3. Programming Details

The description above is actually enough to get everything working, but there ar several places where you are likely to go wrong. First of all, there is no really obviou way to insert at the front of the list from the definitions given. Second, deleting from the front of the list is a special case, because it changes the start of the list; careles coding will lose the list. A third problem concerns deletion in general. Although the

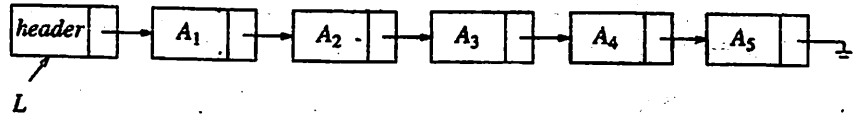


Figure 3.5 Linked list with a header

pointer moves above are simple, the deletion algorithm requires us to keep track of the cell *before* the one that we want to delete.

It turns out that one simple change solves all three problems. We will keep a sentinel node, which is sometimes referred to as a *header* or *dummy* node. This is a common practice, which we will see several times in the future. Our convention will be that the header is in position 0. Figure 3.5 shows a linked list with a header representing the list $A_1, A_2, \ldots, A_5$.

To avoid the problems associated with deletions, we need to write a routine *FindPrevious*, which will return the position of the predecessor of the cell we wish to delete. If we use a header, then if we wish to delete the first element in the list, *FindPrevious* will return the position of the header. The use of a header node is somewhat controversial. Some people argue that avoiding special cases is not sufficient justification for adding fictitious cells; they view the use of header nodes as little more than old-style hacking. Even so, we will use them here, precisely because they allow us to show the basic pointer manipulations without obscuring the code with special cases. Otherwise, whether or not a header should be used is a matter of personal preference.

As examples, we will write about half of the list ADT routines. First, we need our declarations, which are given in Figure 3.6. Following C conventions, the types *List* and *Position*, as well as the function prototypes, are listed in what is to be a *.h* file. The actual *Node* declaration is in a *.c* file.

The first function that we will write tests for an empty list. When we write code for any data structure that involves pointers, it is always best to draw a picture first. Figure 3.7 shows an empty list; from the figure it is easy to write the function in Figure 3.8.

The next function, which is shown in Figure 3.9, tests whether the current element, which by assumption exists, is the last of the list.

The next routine we will write is *Find*. *Find*, shown in Figure 3.10, returns the position in the list of some element. Line 2 takes advantage of the fact that the *and* (&&) operation is *short-circuited:* if the first half of the *and* is false, the result is automatically false and the second half is not executed.

Some programmers find it tempting to code the *Find* routine recursively, possibly because it avoids the sloppy termination condition. We shall see later that this is a very bad idea and should be avoided at all costs.

Our fourth routine will delete some element X in list L. We need to decide what to do if X occurs more than once or not at all. Our routine deletes the first occurrence of X and does nothing if X is not in the list. To do this, we find P, which is the cell prior to the one containing X, via a call to *FindPrevious*. The code to

```
#ifndef _List_H

struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;

List MakeEmpty( List L );
int IsEmpty( List L );
int IsLast( Position P, List L );
Position Find( ElementType X, List L );
void Delete( ElementType X, List L );
Position FindPrevious( ElementType X, List L );
void Insert( ElementType X, List L, Position P );
void DeleteList( List L );
Position Header( List L );
Position First( List L );
Position Advance( Position P );
ElementType Retrieve( Position P );

#endif    /* _List_H */


/* Place in the implementation file */
struct Node
{
    ElementType Element;
    Position    Next;
};
```

Figure 3.6 Type declarations for linked lists

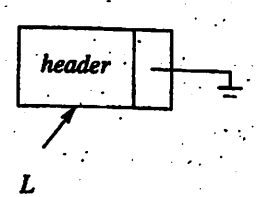


Figure 3.7 Empty list with header


```
/* Return true if L is empty */

int
IsEmpty( List L )
{
    return L->Next == NULL;
}
```

Figure 3.8 Function to test whether a linked list is empty

```
/* Return true if P is the last position in list L */
/* Parameter L is unused in this implementation */

int
IsLast( Position P, List L )
{
    return P->Next == NULL;
}
```

Figure 3.9 Function to test whether current position is the last in a linked list

```
/* Return Position of X in L; NULL if not found */

Position
Find( ElementType X, List L )
{
    Position P;

/* 1*/      P = L->Next;
/* 2*/      while( P != NULL && P->Element != X )
/* 3*/          P = P->Next;

/* 4*/      return P;
}
```

Figure 3.10 *Find* routine

implement this is shown in Figure 3.11. The *FindPrevious* routine is similar to *Find* and is shown in Figure 3.12.

The last routine we will write is an insertion routine. We will pass an element to be inserted along with the list L and a position P. Our particular insertion routine

```
/* Delete first occurrence of X from a list */
/* Assume use of a header node *

void
Delete( ElementType X, List L )
{
    Position P, TmpCell;

    P = FindPrevious( X, L );

    if( !IsLast( P, L ) )  /* Assumption of header use */
    {                      /* X is found; delete it */
        TmpCell = P->Next;
        P->Next = TmpCell->Next;  /* Bypass deleted cell */
        free( TmpCell );
    }
}
```

Figure 3.11 Deletion routine for linked lists

```
/* If X is not found, then Next field of returned */
/* Position is NULL */
/* Assumes a header */

Position
FindPrevious( ElementType X, List L )
{
    Position P;

/* 1*/  P = L;
/* 2*/  while( P->Next != NULL && P->Next->Element != X )
/* 3*/      P = P->Next;

/* 4*/  return P;
}
```

Figure 3.12 *FindPrevious*—the *Find* routine for use with *Delete*

will insert an element *after* the position implied by *P*. This decision is arbitrary and is meant to show that there are no set rules for what insertion does. It is quite possible to insert the new element into position *P* (which means before the element currently in position *P*), but doing this requires knowledge of the element before position *P*. This could be obtained by a call to *FindPrevious*. It is thus important to comment what you are doing. This has been done in Figure 3.13.


```
/* Insert (after legal position P) */
/* Header implementation assumed */
/* Parameter L is unused in this implementation */

void
Insert( ElementType X, List L, Position P )
{
    Position TmpCell;

/* 1*/  TmpCell = malloc( sizeof( struct Node ) );
/* 2*/  if( TmpCell == NULL )
/* 3*/      FatalError( "Out of space!!!" );

/* 4*/  TmpCell->Element = X;
/* 5*/  TmpCell->Next = P->Next;
/* 6*/  P->Next = TmpCell;
}
```

Figure 3.13 Insertion routine for linked lists

Notice that we have passed the list to the *Insert* and *IsLast* routines, even though it was never used. We did this because another implementation might need this information, and so not passing the list would defeat the idea of using ADTs.*

With the exception of the *Find* and *FindPrevious* routines (and *Delete*, which calls *FindPrevious*), all of the operations we have coded take $O(1)$ time. This is because in all cases only a fixed number of instructions are performed, no matter how large the list is. For the *Find* and *FindPrevious* routines, the running time is $O(N)$ in the worst case, because the entire list might need to be traversed if the element either is not found or is last in the list. On average, the running time is $O(N)$, because on average, half the list must be traversed.

Additional routines, listed in Figure 3.6, are fairly straightforward. We could also write a routine to implement *Previous*. We leave these as exercises.

3.2.4. Common Errors

The most common error you will encounter is that your program will crash with a nasty error message from the system, such as "memory access violation" or "segmentation violation." This message usually means that a pointer variable contains a bogus address. One common reason is failure to initialize the variable. For instance, if line 1 in Figure 3.14 is omitted, then *P* is undefined and is not likely to be pointing at a valid part of memory. Another typical error concerns line 6 in Figure 3.13. If *P* is *NULL*, then the indirection is illegal. This function knows that *P* is not *NULL*, so the routine is OK. Of course, you should comment this so that

*This is legal, but some compilers will issue a warning.

```
        /* Incorrect DeleteList algorithm */

        void
        DeleteList( List L )
        {
            Position P;

/* 1*/      P = L->Next;  /* Header assumed */
/* 2*/      L->Next = NULL;
/* 3*/      while( P != NULL )
            {
/* 4*/          free( P );
/* 5*/          P = P->Next;
            }
        }
```

Figure 3.14 Incorrect way to delete a list

the routine that calls *Insert* will ensure this. *Whenever you do an indirection, you must make sure that the pointer is not* NULL. Some C compilers will implicitly do this check for you, but this is not part of the C standard. When you port a program from one compiler to another, you may find that it no longer works. This is one of the common reasons why.

The second common mistake concerns when and when not to use *malloc* to get a new cell. You must remember that declaring a pointer to a structure does not create the structure but only gives enough space to hold the address where some structure might be. The only way to create a record that is not already declared is to use the *malloc* library routine. *malloc*(*HowManyBytes*) has the system create, magically, a new structure and return a pointer to it. If, on the other hand, if you want to use a pointer variable to run down a list, there is no need to create a new structure; in that case the *malloc* command is inappropriate. A type cast is needed on very old compilers to make both sides of the assignment operator compatible. The C library provides other variations of *malloc* such as *calloc*. Both of these routines require the inclusion of *stdlib.h*.

When things are no longer needed, you can issue a *free* command to inform the system that it may reclaim the space. A consequence of the *free*(*P*) command is that the address that *P* is pointing to is unchanged, but the data that reside at that address are now undefined.

If you never delete from a linked list, the number of calls to *malloc* should equal the size of the list, plus 1 if a header is used. Any fewer, and you cannot possibly have a working program. Any more, and you are wasting space and probably time. Occasionally, if your program uses a lot of space, the system may be unable to satisfy your request for a new cell. In this case a *NULL* pointer is returned.

After a deletion in a linked list, it is usually a good idea to free the cell, especially if there are lots of insertions and deletions intermingled and memory might become a problem. You need to keep a temporary variable set to the cell to be disposed of,


```
        /* Correct DeleteList algorithm */

        void
        DeleteList( List L )
        {
            Position P, Tmp;

/* 1*/      P = L->Next;  /* Header assumed */
/* 2*/      L->Next = NULL;
/* 3*/      while( P != NULL )
            {
/* 4*/          Tmp = P->Next;
/* 5*/          free( P );
/* 6*/          P = Tmp;
            }
        }
```

Figure 3.15 Correct way to delete a list

because after the pointer moves are finished, you will not have a reference to it. As an example, the code in Figure 3.14 is not the correct way to delete an entire list (although it may work on some systems).

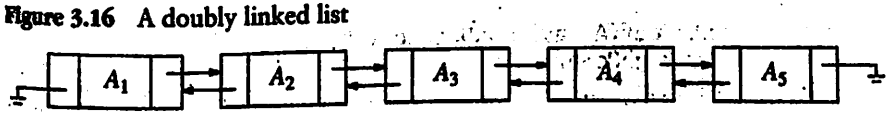
Figure 3.15 shows the correct way to do this. Disposal is not necessarily a fast thing, so you might want to check to see if the disposal routine is causing any slow performance, and comment it out if this is the case. This author has written a program (see the exercises) that was made 25 times faster by commenting out the disposal (of 10,000 nodes). It turned out that the cells were freed in a rather peculiar order and apparently caused an otherwise linear program to spend $O(N \log N)$ time to dispose of N cells.

One last warning: *malloc*(*sizeof*(*PtrToNode*)) is legal, but it doesn't allocate enough space for a structure. It allocates space only for a pointer.

3.2.5. Doubly Linked Lists

Sometimes it is convenient to traverse lists backwards. The standard implementation does not help here, but the solution is simple. Merely add an extra field to the data structure, containing a pointer to the previous cell. The cost of this is an extra link, which adds to the space requirement and also doubles the cost of insertions and deletions because there are more pointers to fix. On the other hand, it simplifies deletion, because you no longer have to refer to a key by using a pointer to the previous cell; this information is now at hand. Figure 3.16 shows a doubly linked list.

Figure 3.16 A doubly linked list



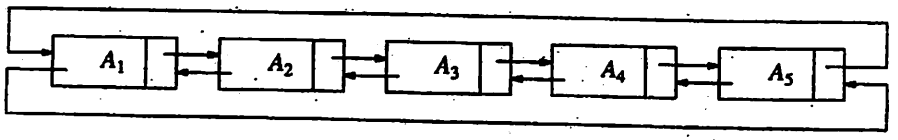


Figure 3.17 A double circularly linked list

3.2.6. *Circularly Linked Lists*

A popular convention is to have the last cell keep a pointer back to the first. This can be done with or without a header (if the header is present, the last cell points to it) and can also be done with doubly linked lists (the first cell's previous pointer points to the last cell). This clearly affects some of the tests, but the structure is popular in some applications. Figure 3.17 shows a double circularly linked list with no header.

3.2.7. *Examples*

We provide three examples that use linked lists. The first is a simple way to represent single-variable polynomials. The second is a method to sort in linear time, for some special cases. Finally, we show a complicated example of how linked lists might be used to keep track of course registration at a university.

The Polynomial ADT

We can define an abstract data type for single-variable polynomials (with nonnegative exponents) by using a list. Let $F(X) = \sum_{i=0}^{N} A_i X^i$. If most of the coefficients A_i are nonzero, we can use a simple array to store the coefficients. We could then write routines to perform addition, subtraction, multiplication, differentiation, and other operations on these polynomials. In this case, we might use the type declarations given in Figure 3.18. We could then write routines to perform various operations. Two possibilities are addition and multiplication; these are shown in Figures 3.19 to 3.21. Ignoring the time to initialize the output polynomials to zero, the running time of the multiplication routine is proportional to the product of the degree of the two input polynomials. This is adequate for dense polynomials, where most of the terms are present, but if $P_1(X) = 10X^{1000} + 5X^{14} + 1$ and $P_2(X) = 3X^{1990} - 2X^{1492} + 11X + 5$, then the running time is likely to be unacceptable. One can see that

Figure 3.18 Type declarations for array implementation of the polynomial ADT

```
typedef struct
{
    int CoeffArray[ MaxDegree + 1 ];
    int HighPower;
} * Polynomial;
```



```
void
ZeroPolynomial( Polynomial Poly )
{
    int i;

    for( i = 0; i <= MaxDegree; i++ )
        Poly->CoeffArray[ i ] = 0;
    Poly->HighPower = 0;
}
```

Figure 3.19 Procedure to initialize a polynomial to zero

```
void
AddPolynomial( const Polynomial Poly1,
               const Polynomial Poly2, Polynomial PolySum )
{
    int i;

    ZeroPolynomial( PolySum );
    PolySum->HighPower = Max( Poly1->HighPower,
                              Poly2->HighPower );

    for( i = PolySum->HighPower; i >= 0; i-- )
        PolySum->CoeffArray[ i ] = Poly1->CoeffArray[ i ]
                                 + Poly2->CoeffArray[ i ]
}
```

Figure 3.20 Procedure to add two polynomials

```
void
MultPolynomial( const Polynomial Poly1,
                const Polynomial Poly2, Polynomial PolyProd )
{
    int i, j;

    ZeroPolynomial( PolyProd );
    PolyProd->HighPower = Poly1->HighPower + Poly2->HighPower;

    if( PolyProd->HighPower > MaxDegree )
        Error( "Exceeded array size" );
    else
        for( i = 0; i <= Poly1->HighPower; i++ )
            for( j = 0; j <= Poly2->HighPower; j++ )
                PolyProd->CoeffArray[ i + j ] +=
                    Poly1->CoeffArray[ i ] *
                    Poly2->CoeffArray[ j ];
}
```

Figure 3.21 Procedure to multiply two polynomials

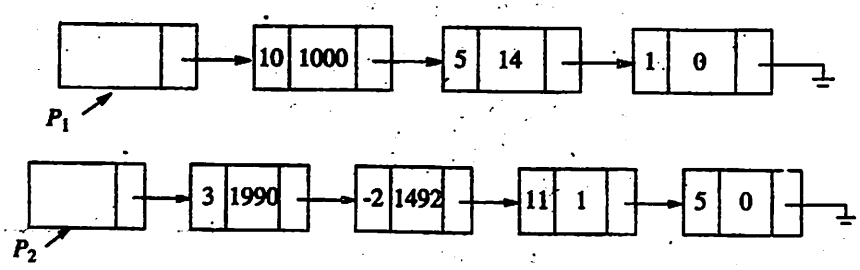


Figure 3.22 Linked list representations of two polynomials

```
typedef struct Node *PtrToNode;

struct Node
{
    int Coefficient;
    int Exponent;
    PtrToNode Next;
};

typedef PtrToNode Polynomial;  /* Nodes sorted by exponent */
```

Figure 3.23 Type declaration for linked list implementation of the Polynomial ADT

most of the time is spent multiplying zeros and stepping through what amounts to nonexistent parts of the input polynomials. This is always undesirable.

An alternative is to use a singly linked list. Each term in the polynomial is contained in one cell, and the cells are sorted in decreasing order of exponents. For instance, the linked lists in Figure 3.22 represent $P_1(X)$ and $P_2(X)$. We could then use the declarations in Figure 3.23.

The operations would then be straightforward to implement. The only potential difficulty is that when two polynomials are multiplied, the resultant polynomial will have to have like terms combined. There are several ways to do this, but we leave this as an exercise.

Radix Sort

A second example where linked lists are used is called *radix sort*. Radix sort is sometimes known as *card sort*, because it was used, until the advent of modern computers, to sort old-style punch cards.

If we have N integers in the range 1 to M (or 0 to $M - 1$), we can use this information to obtain a fast sort known as *bucket sort*. We keep an array called *Count*, of size M, which is initialized to zero. Thus, *Count* has M cells (or buckets), which are initially empty. When A_i is read, increment (by 1) *Count*$[A_i]$. After all the

input is read, scan the *Count* array, printing out a representation of the sorted list. This algorithm takes $O(M + N)$; the proof is left as an exercise. If $M = \Theta(N)$, then bucket sort is $O(N)$.

Radix sort is a generalization of this. The easiest way to see what happens is by example. Suppose we have 10 numbers, in the range 0 to 999, that we would like to sort. In general, this is N numbers in the range 0 to $N^P - 1$ for some constant P. Obviously, we cannot use bucket sort; there would be too many buckets. The trick is to use several passes of bucket sort. The natural algorithm would be to bucket-sort by the most significant "digit" (digit is taken to base N), then the next most significant, and so on. That algorithm does not work, but if we perform bucket sorts by the least significant "digit" first, then the algorithm works. Of course, more than one number could fall into the same bucket and, unlike the original bucket sort, these numbers could be different, so we keep them in a list. Notice that all the numbers could have some digit in common, so if a simple array were used for the lists, each array would have to be of size N, for a total space requirement of $\Theta(N^2)$.

The following example shows the action of radix sort on 10 numbers. The input is 64, 8, 216, 512, 27, 729, 0, 1, 343, 125 (the first 10 cubes, arranged randomly). The first step bucket-sorts by the least significant digit. In this case the math is in base 10 (to make things simple), but do not assume this in general. The buckets are as shown in Figure 3.24, so the list, sorted by least significant digit, is 0, 1, 512, 343, 64, 125, 216, 27, 8, 729. These are now sorted by the next least significant digit (the tens digit here) (see Fig. 3.25). Pass 2 gives output 0, 1, 8, 512, 216, 125, 27, 729, 343, 64. This list is now sorted with respect to the two least significant digits. The final pass, shown in Figure 3.26, bucket-sorts by the most significant digit. The final list is 0, 1, 8, 27, 64, 125, 216, 343, 512, 729.

To see that the algorithm works, notice that the only possible failure would occur if two numbers came out of the same bucket in the wrong order. But the previous passes ensure that when several numbers enter a bucket, they enter in sorted order. The running time is $O(P(N + B))$ where P is the number of passes, N is the number of elements to sort, and B is the number of buckets. In our case, $B = N$.

Figure 3.24 Buckets after first step of radix sort

0	1	512	343	64	125	216	27	8	729
0	1	2	3	4	5	6	7	8	9

Figure 3.25 Buckets after the second pass of radix sort

8 1 0		729 27 125							
	216 512			343		64			
0	1	2	3	4	5	6	7	8	9

64									
27									
8									
1									
0	125	216	343		512		729		
0	1	2	3	4	5	6	7	8	9

Figure 3.26 Buckets after the last pass of radix sort

As an example, we could sort all integers that are representable on a computer (32 bits) by radix sort, if we did three passes over a bucket size of 2^{11}. This algorithm would always be $O(N)$ on this computer, but probably still not as efficient as some of the algorithms we shall see in Chapter 7, because of the high constant involved. (Remember that a factor of $\log N$ is not all that high, and this algorithm would have the overhead of maintaining linked lists.)

Multilists

Our last example shows a more complicated use of linked lists. A university with 40,000 students and 2,500 courses needs to be able to generate two types of reports. The first report lists the registration for each class, and the second report lists, by student, the classes that each student is registered for.

The obvious implementation might be to use a two-dimensional array. Such an array would have 100 million entries. The average student registers for about three courses, so only 120,000 of these entries, or roughly 0.1 percent, would actually have meaningful data.

What is needed is a list for each class containing the students in the class. We also need a list for each student containing the classes the student is registered for. Figure 3.27 shows our implementation.

As the figure shows, we have combined two lists into one. All lists use a header and are circular. To list all of the students in class C3, we start at C3 and traverse its list (by going right). The first cell belongs to student S1. Although there is no explicit information to this effect, this can be determined by following the student's linked list until the header is reached. Once this is done, we return to C3's list (we stored the position we were at in the course list before we traversed the student's list) and find another cell, which can be determined to belong to S3. We can continue and find that S4 and S5 are also in this class. In a similar manner, we can determine, for any student, all of the classes in which the student is registered.

Using a circular list saves space but does so at the expense of time. In the worst case, if the first student was registered for every course, then every entry would need to be examined to determine all the course names for that student. Because in this application there are relatively few courses per student and few students per course, this is not likely to happen. If it were suspected that this could cause a problem, then each of the (nonheader) cells could have pointers directly back to the student and class header. This would double the space requirement but would simplify and speed the implementation.

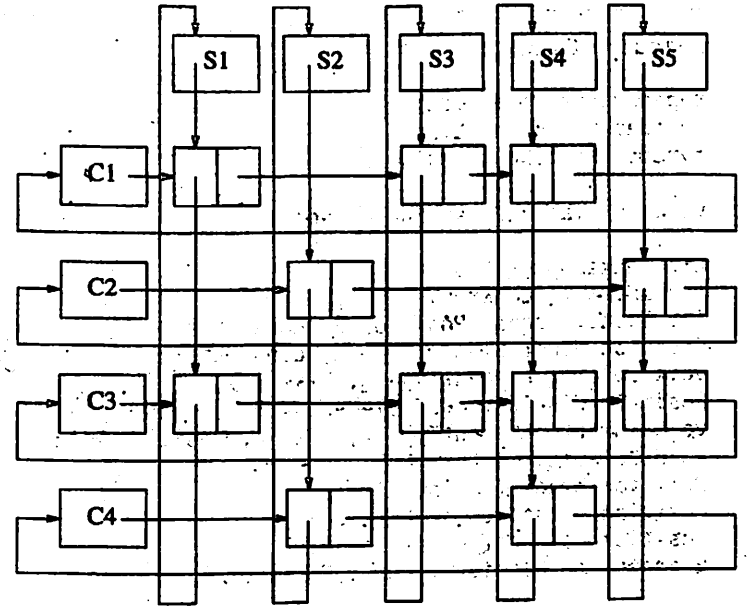


Figure 3.27 Multilist implementation for registration problem

3.2.8. *Cursor Implementation of Linked Lists*

Many languages, such as BASIC and FORTRAN, do not support pointers. If linked lists are required and pointers are not available, then an alternative implementation must be used. The method we will describe is known as a *cursor* implementation.

The two important features present in a pointer implementation of linked lists are as follows:

1. The data are stored in a collection of structures. Each structure contains data and a pointer to the next structure.
2. A new structure can be obtained from the system's global memory by a call to *malloc* and released by a call to *free*.

Our cursor implementation must be able to simulate this. The logical way to satisfy condition 1 is to have a global array of structures. For any cell in the array, its array index can be used in place of an address. Figure 3.28 gives the declarations for a cursor implementation of linked lists.

We must now simulate condition 2 by allowing the equivalent of *malloc* and *free* for cells in the *CursorSpace* array. To do this, we will keep a list (the *freelist*) of cells that are not in any list. The list will use cell 0 as a header. The initial configuration is shown in Figure 3.29.

A value of 0 for *Next* is the equivalent of a *NULL* pointer. The initialization of *CursorSpace* is a straightforward loop, which we leave as an exercise. To perform a *malloc*, the first element (after the header) is removed from the freelist. To perform

```
#ifndef _Cursor_H

typedef int PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;

void InitializeCursorSpace( void );

List MakeEmpty( List L );
int IsEmpty( const List L );
int IsLast( const Position P, const List L );
Position Find( ElementType X, const List L );
void Delete( ElementType X, List L );
Position FindPrevious( ElementType X, const List L );
void Insert( ElementType X, List L, Position P );
void DeleteList( List L );
Position Header( const List L );
Position First( const List L );
Position Advance( const Position P );
ElementType Retrieve( const Position P );

#endif     /* _Cursor_H */


/* Place in the implementation file */
struct Node
{
    ElementType Element;
    Position    Next;
};

struct Node CursorSpace[ SpaceSize ];
```

Figure 3.28 Declarations for cursor implementation of linked lists

a *free*, we place the cell at the front of the freelist. Figure 3.30 shows the cursor implementation of *malloc* and *free*. Notice that if there is no space available, our routine does the correct thing by setting $P = 0$. This indicates that there are no more cells left, and also makes the second line of *CursorAlloc* a nonoperation (no-op).

Given this, the cursor implementation of linked lists is straightforward. For consistency, we will implement our lists with a header node. As an example, in Figure 3.31, if the value of L is 5 and the value of M is 3, then L represents the list a, b, e, and M represents the list c, d, f.

To write the functions for a cursor implementation of linked lists, we must pass and return the identical parameters as the pointer implementation. The routines are straightforward. Figure 3.32 implements a function to test whether a list is empty. Figure 3.33 implements the test of whether the current position is the

Slot	Element	Next
0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

Figure 3.29 An initialized *CursorSpace*

```
static Position
CursorAlloc( void )
{
    Position P;

    P = CursorSpace[ 0 ].Next;
    CursorSpace[ 0 ].Next = CursorSpace[ P ].Next;

    return P;
}

static void
CursorFree( Position P )
{
    CursorSpace[ P ].Next = CursorSpace[ 0 ].Next;
    CursorSpace[ 0 ].Next = P;
}
```

Figure 3.30 Routines: *CursorAlloc* and *CursorFree*

last in a linked list. The function *Find* in Figure 3.34 returns the position of X in list L. The code to implement deletion is shown in Figure 3.35. Again, the interface for the cursor implementation is identical to the pointer implementation. Finally, Figure 3.36 shows a cursor implementation of *Insert*.

The rest of the routines are similarly coded. The crucial point is that these routines follow the ADT specification. They take specific arguments and perform specific operations. The implementation is transparent to the user. The cursor implementation could be used instead of the linked list implementation, with virtually no change required in the rest of the code. If relatively few *Finds* are

Slot	Element	Next
0	–	6
1	b	9
2	f	0
3	header	7
4	–	0
5	header	10
6	–	4
7	c	8
8	d	2
9	e	0
10	a	1

Figure 3.31 Example of a cursor implementation of linked lists

```
/* Return true if L is empty */

int
IsEmpty( List L )
{
    return CursorSpace[ L ].Next == 0;
}
```

Figure 3.32 Function to test whether a linked list is empty—cursor implementation

```
/* Return true if P is the last position in list L */
/* Parameter L is unused in this implementation */

int
IsLast( Position P, List L )
{
    return CursorSpace[ P ].Next == 0;
}
```

Figure 3.33 Function to test whether *P* is last in a linked list—cursor implementation


```
/* Return Position of X in L; 0 if not found */
/* Uses a header node */

Position
Find( ElementType X, List L )
{
    Position P;

/* 1*/  P = CursorSpace[ L ].Next;
/* 2*/  while( P && CursorSpace[ P ].Element != X )
/* 3*/      P = CursorSpace[ P ].Next;

/* 4*/  return P;
}
```

Figure 3.34 *Find* routine—cursor implementation

```
/* Delete first occurrence of X from a list */
/* Assume use of a header node */

void
Delete( ElementType X, List L )
{
    Position P, TmpCell;

    P = FindPrevious( X, L );

    if( !IsLast( P, L ) )   /* Assumption of header use */
    {                       /* X is found; delete it */
        TmpCell = CursorSpace[ P ].Next;
        CursorSpace[ P ].Next = CursorSpace[ TmpCell ].Next;
        CursorFree( TmpCell );
    }
}
```

Figure 3.35 Deletion routine for linked lists—cursor implementation

erformed, the cursor implementation could be significantly faster because of the ack of memory management routines.

The freelist represents an interesting data structure in its own right. The cell hat is removed from the freelist is the one that was most recently placed there by

```
        /* Insert (after legal position P) */
        /* Header implementation assumed */
        /* Parameter L is unused in this implementation */

        void
        Insert( ElementType X, List L, Position P )
        {
            Position TmpCell;

/* 1*/      TmpCell = CursorAlloc( );
/* 2*/      if( TmpCell == 0 )
/* 3*/          FatalError( "Out of space!!!" );

/* 4*/      CursorSpace[ TmpCell ].Element = X;
/* 5*/      CursorSpace[ TmpCell ].Next = CursorSpace[ P ].Next;
/* 6*/      CursorSpace[ P ].Next = TmpCell;
        }
```

Figure 3.36 Insertion routine for linked lists—cursor implementation

virtue of *free*. Thus, the last cell placed on the freelist is the first cell taken off. The data structure that also has this property is known as a *stack*, and is the topic of the next section.

3.3. The Stack ADT

3.3.1. Stack Model

A *stack* is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the *top*. The fundamental operations on a stack are *Push*, which is equivalent to an insert, and *Pop*, which deletes the most recently inserted element. The most recently inserted element can be examined prior to performing a *Pop* by use of the *Top* routine. A *Pop* or *Top* on an empty stack is generally considered an error in the stack ADT. On the other hand, running out of space when performing a *Push* is an implementation error but not an ADT error.

Stacks are sometimes known as LIFO (last in, first out) lists. The model depicted in Figure 3.37 signifies only that *Push*es are input operations and *Pop*s and *Top*s are output. The usual operations to make empty stacks and test for emptiness are part of the repertoire, but essentially all that you can do to a stack is *Push* and *Pop*.

Figure 3.38 shows an abstract stack after several operations. The general model is that there is some element that is at the top of the stack, and it is the only element that is visible.

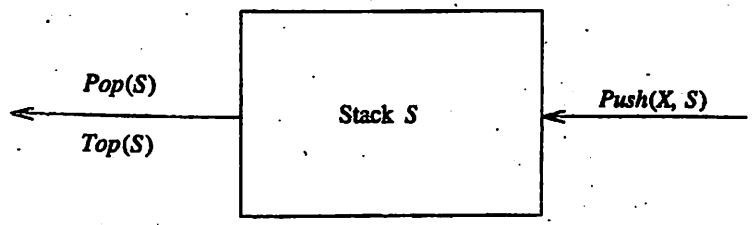


Figure 3.37 Stack model: input to a stack is by *Push*, output is by *Pop*

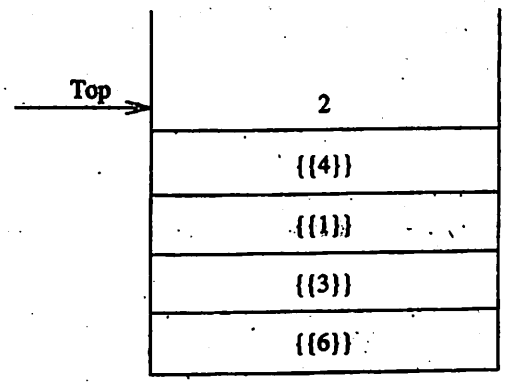


Figure 3.38 Stack model: only the top element is accessible

3.3.2. Implementation of Stacks

Since a stack is a list, any list implementation will do. We will give two popular implementations. One uses pointers and the other uses an array, but, as we saw in the previous section, if we use good programming principles, the calling routines do not need to know which method is being used.

Linked List Implementation of Stacks

The first implementation of a stack uses a singly linked list. We perform a *Push* by inserting at the front of the list. We perform a *Pop* by deleting the element at the front of the list. A *Top* operation merely examines the element at the front of the list, returning its value. Sometimes the *Pop* and *Top* operations are combined into one. We could use calls to the linked list routines of the previous section, but we will rewrite the stack routines from scratch for the sake of clarity.

First, we give the definitions in Figure 3.39. We implement the stack using a header. Figure 3.40 shows that an empty stack is tested for in the same manner as an empty list.

Creating an empty stack is also simple. We merely create a header node; *MakeEmpty* sets the *Next* pointer to *NULL* (see Fig. 3.41). The *Push* is implemented

```
#ifndef _Stack_h

struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode Stack;

int IsEmpty( Stack S );
Stack CreateStack( void );
void DisposeStack( Stack S );
void MakeEmpty( Stack S );
void Push( ElementType X, Stack S );
ElementType Top( Stack S );
void Pop( Stack S );

#endif  /* _Stack_h */


/* Place in implementation file */
/* Stack implementation is a linked list with a header */
struct Node
{
    ElementType Element;
    PtrToNode   Next;
};
```

Figure 3.39 Type declaration for linked list implementation of the stack ADT

```
int
IsEmpty( Stack S )
{
    return S->Next == NULL;
}
```

Figure 3.40 Routine to test whether a stack is empty—linked list implementation

as an insertion into the front of a linked list, where the front of the list serves as the top of the stack (see Fig. 3.42). The *Top* is performed by examining the element in the first position of the list (see Fig. 3.43). Finally, we implement *Pop* as a deletion from the front of the list (see Fig. 3.44).

It should be clear that all the operations take constant time, because nowhere in any of the routines is there even a reference to the size of the stack (except for emptiness), much less a loop that depends on this size. The drawback of this implementation is that the calls to *malloc* and *free* are expensive, especially in


```
Stack
CreateStack( void )
{
    Stack S;

    S = malloc( sizeof( struct Node ) );
    if( S == NULL )
        FatalError( "Out of space!!!" );
    MakeEmpty( S );
    return S;
}

void
MakeEmpty( Stack S )
{
    if( S == NULL )
        Error( "Must use CreateStack first" );
    else
        while( !IsEmpty( S ) )
            Pop( S );
}
```

Figure 3.41 Routine to create an empty stack—linked list implementation

```
void
Push( ElementType X, Stack S )
{
    PtrToNode TmpCell;

    TmpCell = malloc( sizeof( struct Node ) );
    if( TmpCell == NULL )
        FatalError( "Out of space!!!" );
    else
    {
        TmpCell->Element = X;
        TmpCell->Next = S->Next;
        S->Next = TmpCell;
    }
}
```

Figure 3.42 Routine to push onto a stack—linked list implementation

```
ElementType
Top( Stack S )
{
    if( !IsEmpty( S ) )
        return S->Next->Element;
    Error( "Empty stack" );
    return 0;  /* Return value used to avoid warning */
}
```

Figure 3.43 Routine to return top element in a stack—linked list implementation

```
void
Pop( Stack S )
{
    PtrToNode FirstCell;

    if( IsEmpty( S ) )
        Error( "Empty stack" );
    else
    {
        FirstCell = S->Next;
        S->Next = S->Next->Next;
        free( FirstCell );
    }
}
```

Figure 3.44 Routine to pop from a stack—linked list implementation

comparison to the pointer manipulation routines. Some of this can be avoided by using a second stack, which is initially empty. When a cell is to be dropped from the first stack, it is merely placed on the second stack. Then, when new cells are needed for the first stack, the second stack is checked first.

Array Implementation of Stacks

An alternative implementation avoids pointers and is probably the more popular solution. The only potential hazard with this strategy is that we need to declare an array size ahead of time. Generally this is not a problem, because in typical applications, even if there are quite a few stack operations, the actual number of elements in the stack at any time never gets too large. It is usually easy to declare the array to be large enough without wasting too much space. If this is not possible, then a safe course would be to use a linked list implementation.

If we use an array implementation, the implementation is trivial. Associated with each stack is *TopOfStack*, which is -1 for an empty stack (this is how an empty stack

is initialized). To push some element X onto the stack, we increment *TopOfStack* and then set *Stack*[*TopOfStack*] = X, where *Stack* is the array representing the actual stack. To pop, we set the return value to *Stack*[*TopOfStack*] and then decrement *TopOfStack*. Of course, since there are potentially several stacks, the *Stack* array and *TopOfStack* are part of one structure representing a stack. It is almost always a bad idea to use global variables and fixed names to represent this (or any) data structure, because in most real-life situations there will be more than one stack. When writing your actual code, you should attempt to follow the model as closely as possible, so that no part of your code, except for the stack routines, can attempt to access the array or top-of-stack variable implied by each stack. This is true for *all* ADT operations. Modern languages such as Ada and C++ can actually enforce this rule.

Notice that these operations are performed in not only constant time, but very fast constant time. On some machines, *Push*es and *Pop*s (of integers) can be written in one machine instruction, operating on a register with auto-increment and auto-decrement addressing. The fact that most modern machines have stack operations as part of the instruction set enforces the idea that the stack is probably the most fundamental data structure in computer science, after the array.

One problem that affects the efficiency of implementing stacks is error testing. Our linked list implementation carefully checked for errors. As described above, a *Pop* on an empty stack or a *Push* on a full stack will overflow the array bounds and cause a crash. This is obviously undesirable, but if checks for these conditions were put in the array implementation, they would likely take as much time as the actual stack manipulation. For this reason, it has become a common practice to skimp on error checking in the stack routines, except where error handling is crucial (as in operating systems). Although you can probably get away with this in most cases by declaring the stack to be large enough not to overflow and ensuring that routines that use *Pop* never attempt to *Pop* an empty stack, this can lead to code that barely works at best, especially when programs are large and are written by more than one person or at more than one time. Because stack operations take such fast constant time, it is rare that a significant part of the running time of a program is spent in these routines. This means that it is generally not justifiable to omit error checks. You should always write the error checks; if they are redundant, you can always comment them out if they really cost too much time. Having said all this, we can now write routines to implement a general stack using arrays.

A *Stack* is defined in Figure 3.45 as a pointer to a structure. The structure contains the *TopOfStack* and *Capacity* fields. Once the maximum size is known, the stack array can be dynamically allocated. Figure 3.46 creates a stack of a given maximum size. Lines 3–5 allocate the stack structure, and lines 6–8 allocate the stack array. Lines 9 and 10 initialize the *TopOfStack* and *Capacity* fields. The stack array does not need to be initialized. The stack is returned at line 11.

The routine *DisposeStack* should be written to free the stack structure. This routine first frees the stack array and then the stack structure (see Fig. 3.47). Since *CreateStack* requires an argument in the array implementation, but not in the linked list implementation, the routine that uses a stack will need to know which implementation is being used unless a dummy parameter is added for the

```
#ifndef _Stack_h

struct StackRecord;
typedef struct StackRecord *Stack;

int IsEmpty( Stack S );
int IsFull( Stack S );
Stack CreateStack( int MaxElements );
void DisposeStack( Stack S );
void MakeEmpty( Stack S );
void Push( ElementType X, Stack S );
ElementType Top( Stack S );
void Pop( Stack S );
ElementType TopAndPop( Stack S );

#endif  /* _Stack_h */


/* Place in implementatioin file */
/* Stack implementation is a dynamically allocated array */
#define EmptyTOS ( -1 )
#define MinStackSize ( 5 )

struct StackRecord
{
    int Capacity;
    int TopOfStack;
    ElementType *Array;
};
```

Figure 3.45 *Stack* declarations—array implementation

later implementation. Unfortunately, efficiency and software idealism often create conflicts.

We have assumed that all stacks deal with the same type of element. In many languages, if there are different types of stacks, then we need to rewrite a new version of the stack routines for each different type, giving each version a different name. A cleaner alternative is provided in C++, which allows one to write a set of generic stack routines that work for any type. C++ also allows stacks of several different types to retain the same procedure and function names (such as *Push* and *Pop*): The compiler decides which routines are implied by checking the type of the calling routine.

Having said all this, we will now rewrite the four stack routines. In true ADT spirit, we will make the function and procedure heading look identical to the linked list implementation. The routines themselves are very simple and follow the written description exactly (see Figs. 3.48 to 3.52).


```
Stack
CreateStack( int MaxElements )
{
    Stack S;

/* 1*/  if( MaxElements < MinStackSize )
/* 2*/      Error( "Stack size is too small" );

/* 3*/  S = malloc( sizeof( struct StackRecord ) );
/* 4*/  if( S == NULL )
/* 5*/      FatalError( "Out of space!!!" );

/* 6*/  S->Array = malloc( sizeof( ElementType ) * MaxElements );
/* 7*/  if( S->Array == NULL )
/* 8*/      FatalError( "Out of space!!!" );
/* 9*/  S->Capacity = MaxElements;
/*10*/  MakeEmpty( S );

/*11*/  return S;
}
```

Figure 3.46 Stack creation—array implementation

```
void
DisposeStack( Stack S )
{
    if( S != NULL )
    {
        free( S->Array );
        free( S );
    }
}
```

Figure 3.47 Routine for freeing stack—array implementation

```
int
IsEmpty( Stack S )
{
    return S->TopOfStack == EmptyTOS;
}
```

Figure 3.48 Routine to test whether a stack is empty—array implementation

```
void
MakeEmpty( Stack S )
{
    S->TopOfStack = EmptyTOS;
}
```

Figure 3.49 Routine to create an empty stack—array implementation

```
void
Push( ElementType X, Stack S )
{
    if( IsFull( S ) )
        Error( "Full stack" );
    else
        S->Array[ ++S->TopOfStack ] = X;
}
```

Figure 3.50 Routine to push onto a stack—array implementation

```
ElementType
Top( Stack S )
{
    if( !IsEmpty( S ) )
        return S->Array[ S->TopOfStack ];
    Error( "Empty stack" );
    return 0;  /* Return value used to avoid warning */
}
```

Figure 3.51 Routine to return top of stack—array implementation

```
void
Pop( Stack S )
{
    if( IsEmpty( S ) )
        Error( "Empty stack" );
    else
        S->TopOfStack--;
}
```

Figure 3.52 Routine to pop from a stack—array implementation


```
ElementType
TopAndPop( Stack S )
{
    if( !IsEmpty( S ) )
        return S->Array[ S->TopOfStack-- ];
    Error( "Empty stack" );
    return 0;  /* Return value used to avoid warning */
}
```

Figure 3.53 Routine to give top element and pop a stack—array implementation

Pop is occasionally written as a function that returns the popped element (and alters the stack). Although current thinking suggests that functions should not change their input variables, Figure 3.53 illustrates that this is the most convenient method in C.

3.3.3. *Applications*

It should come as no surprise that if we restrict the operations allowed on a list, those operations can be performed very quickly. The big surprise, however, is that the small number of operations left are so powerful and important. We give three of the many applications of stacks. The third application gives a deep insight into how programs are organized.

Balancing Symbols

Compilers check your programs for syntax errors, but frequently a lack of one symbol (such as a missing brace or comment starter) will cause the compiler to spill out a hundred lines of diagnostics without identifying the real error.

A useful tool in this situation is a program that checks whether everything is balanced. Thus, every right brace, bracket, and parenthesis must correspond to its left counterpart. The sequence [()] is legal, but [(]) is wrong. Obviously, it is not worthwhile writing a huge program for this, but it turns out that it is easy to check these things. For simplicity, we will just check for balancing of parentheses, brackets, and braces and ignore any other character that appears.

The simple algorithm uses a stack and is as follows:

> Make an empty stack. Read characters until end of file. If the character is an opening symbol, push it onto the stack. If it is a closing symbol, then if the stack is empty report an error. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error. At end of file, if the stack is not empty report an error.

You should be able to convince yourself that this algorithm works. It is clearly linear and actually makes only one pass through the input. It is thus on-line and quite fast. Extra work can be done to attempt to decide what to do when an error is reported—such as identifying the likely cause.

Postfix Expressions

Suppose we have a pocket calculator and would like to compute the cost of a shopping trip. To do so, we add a list of numbers and multiply the result by 1.06; this computes the purchase price of some items with local sales tax added. If the items are 4.99, 5.99, and 6.99, then a natural way to enter this would be the sequence

4.99 + 5.99 + 6.99 * 1.06 =

Depending on the calculator, this produces either the intended answer, 19.05, or the scientific answer, 18.39. Most simple four-function calculators will give the first answer, but many advanced calculators know that multiplication has higher precedence than addition.

On the other hand, some items are taxable and some are not, so if only the first and last items were actually taxable, then the sequence

4.99 * 1.06 + 5.99 + 6.99 * 1.06 =

would give the correct answer (18.69) on a scientific calculator and the wrong answer (19.37) on a simple calculator. A scientific calculator generally comes with parentheses, so we can always get the right answer by parenthesizing, but with a simple calculator we need to remember intermediate results.

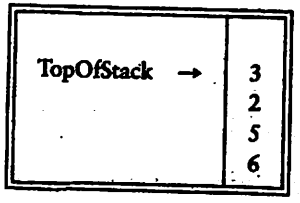
A typical evaluation sequence for this example might be to multiply 4.99 and 1.06, saving this answer as A_1. We then add 5.99 and A_1, saving the result in A_1. We multiply 6.99 and 1.06, saving the answer in A_2, and finish by adding A_1 and A_2, leaving the final answer in A_1. We can write this sequence of operations as follows:

4.99 1.06 * 5.99 + 6.99 1.06 * +

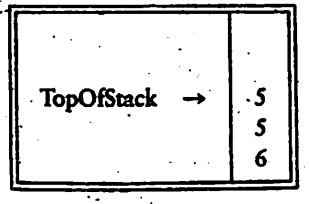
This notation is known as *postfix* or *reverse Polish* notation and is evaluated exactly as we have described above. The easiest way to do this is to use a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers (symbols) that are popped from the stack, and the result is pushed onto the stack. For instance, the postfix expression

6 5 2 3 + 8 * + 3 + *

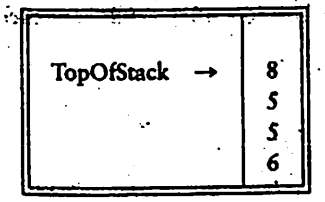
is evaluated as follows: The first four symbols are placed on the stack. The resulting stack is



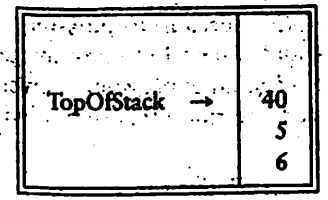
Next a '+' is read, so 3 and 2 are popped from the stack and their sum, 5, is pushed.



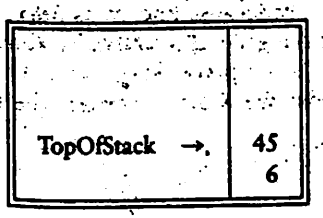
Next 8 is pushed.



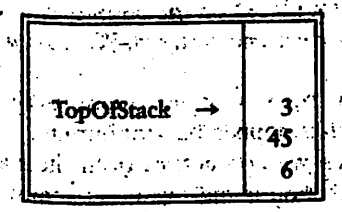
Now a '*' is seen, so 8 and 5 are popped and 5 * 8 = 40 is pushed.



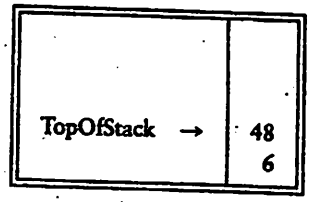
Next a '+' is seen, so 40 and 5 are popped and 5 + 40 = 45 is pushed.



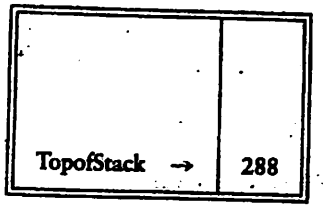
Now, 3 is pushed.



Next '+' pops 3 and 45 and pushes 45 + 3 = 48.



Finally, a '*' is seen and 48 and 6 are popped; the result, 6 * 48 = 288, is pushed.



The time to evaluate a postfix expression is $O(N)$, because processing each element in the input consists of stack operations and thus takes constant time. The algorithm to do so is very simple. Notice that when an expression is given in postfix notation, there is no need to know any precedence rules; this is an obvious advantage.

Infix to Postfix Conversion

Not only can a stack be used to evaluate a postfix expression, but we can also use a stack to convert an expression in standard form (otherwise known as *infix*) into postfix. We will concentrate on a small version of the general problem by allowing only the operators +, *, (,), and insisting on the usual precedence rules. We will further assume that the expression is legal. Suppose we want to convert the infix expression

a + b * c + (d * e + f) * g

into postfix. A correct answer is a b c * + d e * f + g * +.

When an operand is read, it is immediately placed onto the output. Operators are not immediately output, so they must be saved somewhere. The correct thing to do is to place operators that have been seen, but not placed on the output, onto the stack. We will also stack left parentheses when they are encountered. We start with an initially empty stack.

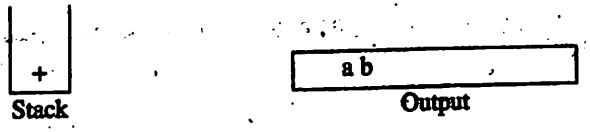
If we see a right parenthesis, then we pop the stack, writing symbols until we encounter a (corresponding) left parenthesis, which is popped but not output.

If we see any other symbol ('+', '*', '('), then we pop entries from the stack until we find an entry of lower priority. One exception is that we never remove a

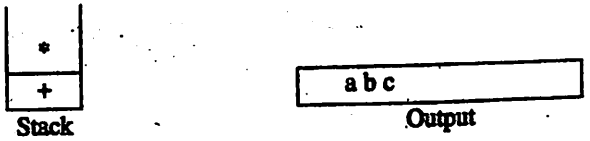
'(' from the stack except when processing a ')'. For the purposes of this operation, '+' has lowest priority and '(' highest. When the popping is done, we push the operator onto the stack.

Finally, if we read the end of input, we pop the stack until it is empty, writing symbols onto the output.

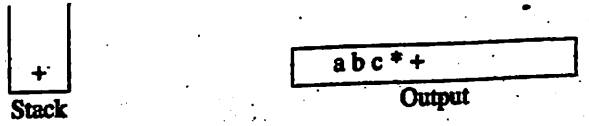
To see how this algorithm performs, we will convert the infix expression above into its postfix form. First, the symbol *a* is read, so it is passed through to the output. Then '+' is read and pushed onto the stack. Next *b* is read and passed through to the output. The state of affairs at this juncture is as follows:



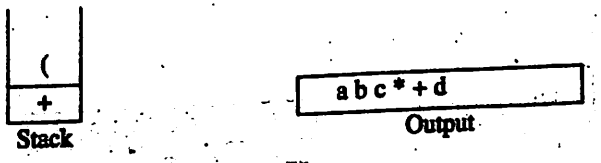
Next a '*' is read. The top entry on the operator stack has lower precedence than '*', so nothing is output and '*' is put on the stack. Next, *c* is read and output. Thus far, we have



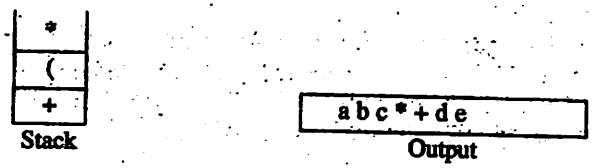
The next symbol is a '+'. Checking the stack, we find that we will pop a '*' and place it on the output; pop the other '+', which is not of *lower* but equal priority, on the stack; and then push the '+'.



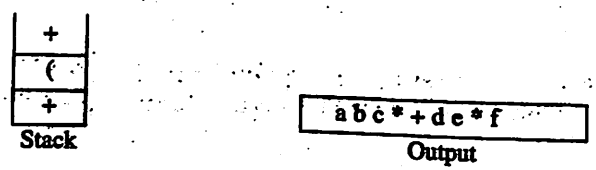
The next symbol read is a '(', which, being of highest precedence, is placed on the stack. Then *d* is read and output.



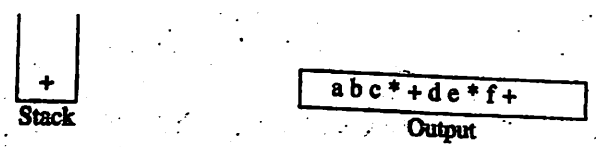
We continue by reading a '*'. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



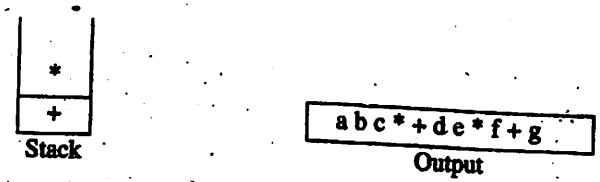
The next symbol read is a '+'. We pop and output '*' and then push '+'. Then we read and output f.



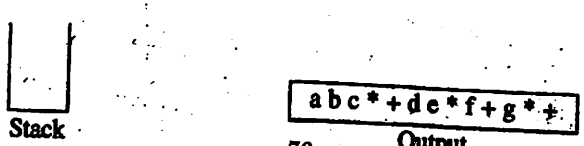
Now we read a ')', so the stack is emptied back to the '('. We output a '+'.



We read a '*' next; it is pushed onto the stack. Then g is read and output.



The input is now empty, so we pop and output symbols from the stack until it is empty.



As before, this conversion requires only $O(N)$ time and works in one pass through the input. We can add subtraction and division to this repertoire by assigning subtraction and addition equal priority and multiplication and division equal priority. A subtle point is that the expression $a - b - c$ will be converted to $ab - c-$ and not $abc - -$. Our algorithm does the right thing, because these operators associate from left to right. This is not necessarily the case in general, since exponentiation associates right to left: $2^{2^3} = 2^8 = 256$, not $4^3 = 64$. We leave as an exercise the problem of adding exponentiation to the repertoire of assignments.

Function Calls

The algorithm to check balanced symbols suggests a way to implement function calls. The problem here is that when a call is made to a new function, all the variables local to the calling routine need to be saved by the system, since otherwise the new function will overwrite the calling routine's variables. Furthermore, the current location in the routine must be saved so that the new function knows where to go after it is done. The variables have generally been assigned by the compiler to machine registers, and there are certain to be conflicts (usually all procedures get some variables assigned to register #1), especially if recursion is involved. The reason that this problem is similar to balancing symbols is that a function call and function return are essentially the same as an open parenthesis and closed parenthesis, so the same ideas should work.

When there is a function call, all the important information that needs to be saved, such as register values (corresponding to variable names) and the return address (which can be obtained from the program counter, which is typically in a register), is saved "on a piece of paper" in an abstract way and put at the top of a pile. Then the control is transferred to the new function, which is free to replace the registers with its values. If it makes other function calls, it follows the same procedure. When the function wants to return, it looks at the "paper" at the top of the pile and restores all the registers. It then makes the return jump.

Clearly, all of this work can be done using a stack, and that is exactly what happens in virtually every programming language that implements recursion. The information saved is called either an *activation record* or *stack frame*. Typically, a slight adjustment is made: The current environment is represented at the top of the stack. Thus, a return gives the previous environment (without copying). The stack in a real computer frequently grows from the high end of your memory partition downward, and on many systems there is no checking for overflow. There is always the possibility that you will run out of stack space by having too many simultaneously active functions. Needless to say, running out of stack space is always a fatal error.

In languages and systems that do not check for stack overflow, your program will crash without an explicit explanation. On these systems, strange things may happen when your stack gets too big, because your stack will run into part of your program. It could be the main program, or it could be part of your data, especially if you have a big array. If it runs into your program, your program will be corrupted; you will have nonsense instructions and will crash as soon as they are executed. If the stack runs into your data, what is likely to happen is that when you write

```
        /* Bad use of recursion: Printing a linked list */
        /* No header */

        void
        PrintList( List L )
        {
/* 1*/      if( L != NULL )
            {
/* 2*/          PrintElement( L->Element );
/* 3*/          PrintList( L->Next);
            }
        }
```

Figure 3.54 A bad use of recursion: printing a linked list

something into your data, it will destroy stack information—probably the return address—and your program will attempt to return to some weird address and crash.

In normal events, you should not run out of stack space; doing so is usually an indication of runaway recursion (forgetting a base case). On the other hand, some perfectly legal and seemingly innocuous program can cause you to run out of stack space. The routine in Figure 3.54, which prints out a linked list, is perfectly legal and actually correct. It properly handles the base case of an empty list, and the recursion is fine. This program can be *proven* correct. Unfortunately, if the list contains 20,000 elements, there will be a stack of 20,000 activation records representing the nested calls of line 3. Activation records are typically large because of all the information they contain, so this program is likely to run out of stack space. (If 20,000 elements are not enough to make the program crash, replace the number with a larger one.)

This program is an example of an extremely bad use of recursion known as *tail recursion*. Tail recursion refers to a recursive call at the last line. Tail recursion can be mechanically eliminated by changing the recursive call to a *goto* preceded by one assignment per function argument. This simulates the recursive call because nothing needs to be saved; after the recursive call finishes, there is really no need to know the saved values. Because of this, we can just go to the top of the function with the values that would have been used in a recursive call. The program in Figure 3.55 shows the improved version. Keep in mind that *you* should use the more natural *while* loop construction. The *goto* is used here to show how a compiler might automatically remove the recursion.

Removal of tail recursion is so simple that some compilers do it automatically. Even so, it is best not to find out that yours does not.

Recursion can always be completely removed (the compiler does so in converting to assembly language), but doing so can be quite tedious. The general strategy requires using a stack and is worthwhile only if you can manage to put the bare minimum on the stack. We will not dwell on this further, except to point out that although nonrecursive programs are certainly generally faster than equivalent recursive programs, the speed advantage rarely justifies the lack of clarity that results from removing the recursion.


```
/* Printing a linked list non-recursively */
/* Uses a mechanical translation */
/* No header */

void
PrintList( List L )
{
  top:
    if( L != NULL )
    {
        PrintElement( L->Element );
        L = L->Next;
        goto top;
    }
}
```

Figure 3.55 Printing a list without recursion; a compiler might do this (you should not)

3.4. The Queue ADT

Like stacks, *queues* are lists. With a queue, however, insertion is done at one end, whereas deletion is performed at the other end.

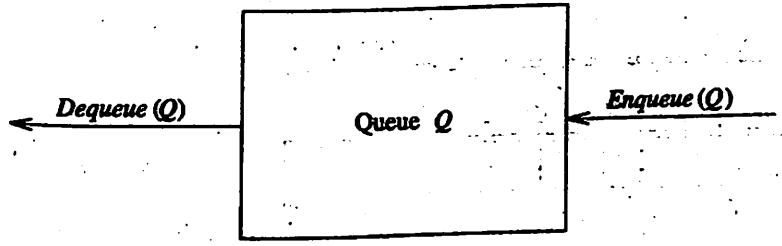
3.4.1. Queue Model

The basic operations on a queue are *Enqueue*, which inserts an element at the end of the list (called the rear), and *Dequeue*, which deletes (and returns) the element at the start of the list (known as the front). Figure 3.56 shows the abstract model of a queue.

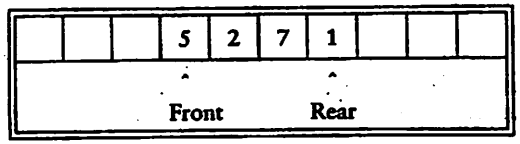
3.4.2. Array Implementation of Queues

As with stacks, any list implementation is legal for queues. Like stacks, both the linked list and array implementations give fast $O(1)$ running times for every operation. The linked list implementation is straightforward and left as an exercise. We will now discuss an array implementation of queues.

Figure 3.56 Model of a queue



For each queue data structure, we keep an array, *Queue*[], and the positions *Front* and *Rear*, which represent the ends of the queue. We also keep track of the number of elements that are actually in the queue, *Size*. All this information is part of one structure, and as usual, except for the queue routines themselves, no routine should ever access these directly. The following figure shows a queue in some intermediate state. By the way, the cells that are blanks have undefined values in them. In particular, the first three cells have elements that used to be in the queue.



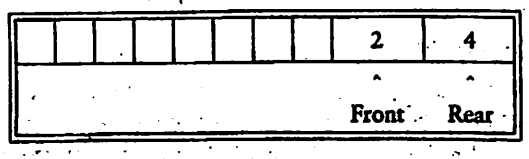
The operations should be clear. To *Enqueue* an element X, we increment *Size* and *Rear*, then set *Queue*[*Rear*] = X. To *Dequeue* an element, we set the return value to *Queue*[*Front*], decrement *Size*, and then increment *Front*. Other strategies are possible (this is discussed later). We will comment on checking for errors presently.

There is one potential problem with this implementation. After 10 *Enqueues*, the queue appears to be full, since *Rear* is now 10, and the next *Enqueue* would be in a nonexistent position. However, there might only be a few elements in the queue, because several elements may have already been dequeued. Queues, like stacks, frequently stay small even in the presence of a lot of operations.

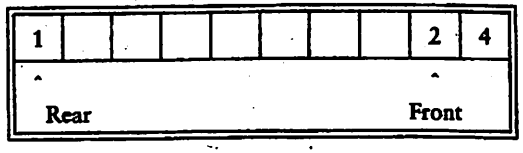
The simple solution is that whenever *Front* or *Rear* gets to the end of the array, it is wrapped around to the beginning. The following figure shows the queue during some operations. This is known as a *circular array* implementation.

The extra code required to implement the wraparound is minimal (although it probably doubles the running time). If incrementing either *Rear* or *Front* causes it to go past the array, the value is reset to the first position in the array.

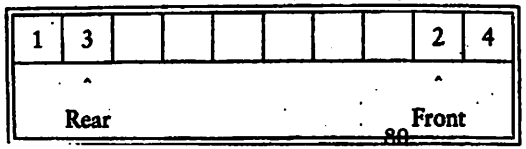
Initial State



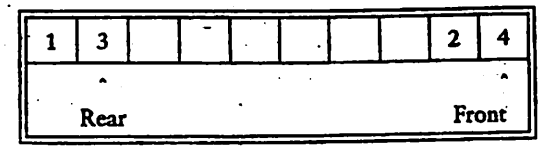
After *Enqueue*(1)



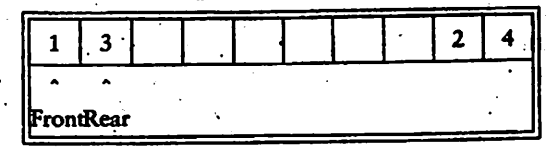
After *Enqueue*(3)



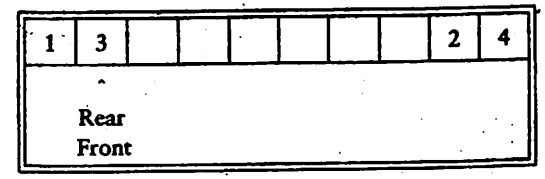
After *Dequeue*, Which Returns 2



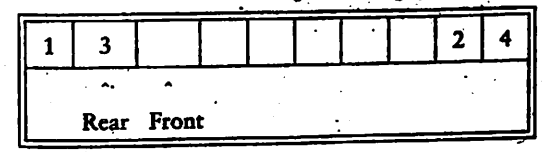
After *Dequeue*, Which Returns 4



After *Dequeue*, Which Returns 1



After *Dequeue*, Which Returns 3 and Makes the Queue Empty



There are two warnings about the circular array implementation of queues. First, it is important to check the queue for emptiness, because a *Dequeue* when the queue is empty will return an undefined value, silently.

Second, some programmers use different ways of representing the front and rear of a queue. For instance, some do not use an entry to keep track of the size, because they rely on the base case that when the queue is empty, *Rear* = *Front* − 1. The size is computed implicitly by comparing *Rear* and *Front*. This is a very tricky way to go, because there are some special cases, so be very careful if you need to modify code written this way. If the size is not part of the structure, then if the array size is *ASize*, the queue is full when there are *ASize* − 1 elements, since only *ASize* different sizes can be differentiated, and one of these is 0. Pick any style you like and make sure that all your routines are consistent. Since there are a few options for implementation, it is probably worth a comment or two in the code, if you don't use the size field.

In applications where you are sure that the number of *Enqueues* is not larger than the size of the queue, the wraparound is not necessary. As with stacks, *Dequeues* are rarely performed unless the calling routines are certain that the queue is not empty. Thus error calls are frequently skipped for this operation, except in critical code. This is generally not justifiable, because the time savings that you are likely to achieve are minimal.

We finish this section by writing some of the queue routines. We leave the others as an exercise. First, we give the queue declarations in Figure 3.57. We add a maximum size field, as was done for the array implementation of the stack; *CreateQueue* and *DisposeQueue* routines also need to be provided We also provide routines to test whether a queue is empty and to make an empty queue (Figs. 3.58 and 3.59). The reader can write the function *IsFull*, which performs the test

Figure 3.57 Type declarations for queue—array implementation

```
#ifndef _Queue_h

struct QueueRecord;
typedef struct QueueRecord *Queue;

int IsEmpty( Queue Q );
int IsFull( Queue Q );
Queue CreateQueue( int MaxElements );
void DisposeQueue( Queue Q );
void MakeEmpty( Queue Q );
void Enqueue( ElementType X, Queue Q );
ElementType Front( Queue Q );
void Dequeue( Queue Q );
ElementType FrontAndDequeue( Queue Q );

#endif  /* _Queue_h */


/* Place in implementation file */
/* Queue implementation is a dynamically allocated array */
#define MinQueueSize ( 5 )

struct QueueRecord
{
    int Capacity;
    int Front;
    int Rear;
    int Size;
    ElementType *Array;
};
```



```
int
IsEmpty( Queue Q )
{
    return Q->Size == 0;
}
```

Figure 3.58 Routine to test whether a queue is empty—array implementation

```
void
MakeEmpty( Queue Q )
{
    Q->Size = 0;
    Q->Front = 1;
    Q->Rear = 0;
}
```

Figure 3.59 Routine to make an empty queue—array implementation

```
static int
Succ( int Value, Queue Q )
{
    if( ++Value == Q->Capacity )
        Value = 0;
    return Value;
}

void
Enqueue( ElementType X, Queue Q )
{
    if( IsFull( Q ) )
        Error( "Full queue" );
    else
    {
        Q->Size++;
        Q->Rear = Succ( Q->Rear, Q );
        Q->Array[ Q->Rear ] = X;
    }
}
```

Figure 3.60 Routines to enqueue—array implementation

implied by its name. Notice that *Rear* is preinitialized to 1 before *Front*. The final operation we will write is the *Enqueue* routine. Following the exact description above, we arrive at the implementation in Figure 3.60.

3.4.3. Applications of Queues

There are several algorithms that use queues to give efficient running times. Several of these are found in graph theory, and we will discuss them in Chapter 9. For now, we will give some simple examples of queue usage.

When jobs are submitted to a printer, they are arranged in order of arrival. Thus, essentially, jobs sent to a line printer are placed on a queue.*

Virtually every real-life line is (supposed to be) a queue. For instance, lines at ticket counters are queues, because service is first-come first-served.

Another example concerns computer networks. There are many network setups of personal computers in which the disk is attached to one machine, known as the *file server*. Users on other machines are given access to files on a first-come first-served basis, so the data structure is a queue.

Further examples include the following:

- Calls to large companies are generally placed on a queue when all operators are busy.
- In large universities, where resources are limited, students must sign a waiting list if all terminals are occupied. The student who has been at a terminal the longest is forced off first, and the student who has been waiting the longest is the next user to be allowed on.

A whole branch of mathematics, known as *queueing theory*, deals with computing, probabilistically, how long users expect to wait on a line, how long the line gets, and other such questions. The answer depends on how frequently users arrive to the line and how long it takes to process a user once the user is served. Both of these parameters are given as probability distribution functions. In simple cases, an answer can be computed analytically. An example of an easy case would be a phone line with one operator. If the operator is busy, callers are placed on a waiting line (up to some maximum limit). This problem is important for businesses, because studies have shown that people are quick to hang up the phone.

If there are k operators, then this problem is much more difficult to solve. Problems that are difficult to solve analytically are often solved by a simulation. In our case, we would need to use a queue to perform the simulation. If k is large, we also need other data structures to do this efficiently. We shall see how to do this simulation in Chapter 6. We could then run the simulation for several values of k and choose the minimum k that gives a reasonable waiting time.

Additional uses for queues abound, and as with stacks, it is staggering that such a simple data structure can be so important.

*We say *essentially* because jobs can be killed. This amounts to a deletion from the middle of the queue, which is a violation of the strict definition.

Summary

This chapter describes the concept of ADTs and illustrates the concept with three of the most common abstract data types. The primary objective is to separate the implementation of the abstract data types from their function. The program must know what the operations do, but it is actually better off not knowing how it is done.

Lists, stacks, and queues are perhaps the three fundamental data structures in all of computer science, and their use is documented through a host of examples. In particular, we saw how stacks are used to keep track of procedure and function calls and how recursion is actually implemented. This is important to understand, not just because it makes procedural languages possible, but because knowing how recursion is implemented removes a good deal of the mystery that surrounds its use. Although recursion is very powerful, it is not an entirely free operation; misuse and abuse of recursion can result in programs crashing.

Exercises

3.1 Write a program to print out the elements of a singly linked list.

3.2 You are given a linked list, L, and another linked list, P, containing integers sorted in ascending order. The operation *PrintLots(L,P)* will print the elements in L that are in positions specified by P. For instance, if $P = 1, 3, 4, 6$, the first, third, fourth, and sixth elements in L are printed. Write the procedure *PrintLots(L,P)*. You should use only the basic list operations. What is the running time of your procedure?

3.3 Swap two adjacent elements by adjusting only the pointers (and not the data) using:

a. Singly linked lists.

b. Doubly linked lists.

3.4 Given two sorted lists, L_1 and L_2, write a procedure to compute $L_1 \cap L_2$ using only the basic list operations.

3.5 Given two sorted lists, L_1 and L_2, write a procedure to compute $L_1 \cup L_2$ using only the basic list operations.

3.6 Write a function to add two polynomials. Do not destroy the input. Use a linked list implementation. If the polynomials have M and N terms, respectively, what is the time complexity of your program?

3.7 Write a function to multiply two polynomials, using a linked list implementation. You must make sure that the output polynomial is sorted by exponent and has at most one term of any power.

a. Give an algorithm to solve this problem in $O(M^2N^2)$ time.

*b. Write a program to perform the multiplication in $O(M^2N)$ time, where M is the number of terms in the polynomial of fewer terms.

*c. Write a program to perform the multiplication in $O(MN \log(MN))$ time.

d. Which time bound above is the best?

3.8 Write a program that takes a polynomial, $F(X)$, and computes $(F(X))^P$. What is the complexity of your program? Propose at least one alternative solution that could be competitive for some plausible choices of $F(X)$ and P.

3.9 Write an arbitrary-precision integer arithmetic package. You should use a strategy similar to polynomial arithmetic. Compute the distribution of the digits 0 to 9 in $2^{4,000}$.

3.10 The *Josephus problem* is the following game: N people, numbered 1 to N, are sitting in a circle. Starting at person 1, a hot potato is passed. After M passes, the person holding the hot potato is eliminated, the circle closes ranks, and the game continues with the person who was sitting after the eliminated person picking up the hot potato. The last remaining person wins. Thus, if $M = 0$ and $N = 5$, players are eliminated in order, and player 5 wins. If $M = 1$ and $N = 5$, the order of elimination is 2, 4, 1, 5.

a. Write a program to solve the Josephus problem for general values of M and N. Try to make your program as efficient as possible. Make sure you dispose of cells.

b. What is the running time of your program?

c. If $M = 1$, what is the running time of your program? How is the actual speed affected by the *free* routine for large values of N ($N > 10{,}000$)?

3.11 Write a program to find a particular element in a singly linked list. Do this both recursively and nonrecursively, and compare the running times. How big does the list have to be before the recursive version crashes?

3.12 a. Write a nonrecursive procedure to reverse a singly linked list in $O(N)$ time.

*b. Write a procedure to reverse a singly linked list in $O(N)$ time using constant extra space.

3.13 You have to sort an array of student records by social security number. Write a program to do this, using radix sort with 1,000 buckets and three passes.

3.14 Write a program to read a graph into adjacency lists using:

a. Linked lists.

b. Cursors.

3.15 a. Write an array implementation of self-adjusting lists. A *self-adjusting* list is like a regular list, except that all insertions are performed at the front, and when an element is accessed by a *Find*, it is moved to the front of the list without changing the relative order of the other items.

b. Write a linked list implementation of self-adjusting lists.

*c. Suppose each element has a fixed probability, p_i, of being accessed. Show that the elements with highest access probability are expected to be close to the front.

3.16 Suppose we have an array-based list $A[0..N-1]$ and we want to delete all duplicates. *LastPosition* is initially $N-1$, but gets smaller as elements


```
/* 1*/  for( i = 0; i < LastPosition; i++ )
        {
/* 2*/      j = i + 1;
/* 3*/      while( j < LastPosition )
/* 4*/          if( A[ i ] == A[ j ] )
/* 5*/              Delete( j );
            else
/* 6*/              j++;
        }
```

Figure 3.61 Routine to remove duplicates from a list—array implementation

are deleted. Consider the pseudocode program fragment in Figure 3.61. The procedure *Delete* deletes the element in position j and collapses the list.

a. Explain how this procedure works.

b. Rewrite this procedure using general list operations.

*c. Using a standard array implementation, what is the running time of this procedure?

d. What is the running time using a linked list implementation?

*e. Give an algorithm to solve this problem in $O(N \log N)$ time.

**f. Prove that any algorithm to solve this problem requires $\Omega(N \log N)$ comparisons if only comparisons are used. (*Hint*: Look to Chapter 7.)

*g. Prove that if we allow operations besides comparisons, and the keys are real numbers, then we can solve the problem without using comparisons between elements.

3.17 An alternative to the deletion strategy we have given is to use *lazy deletion*. To delete an element, we merely mark it deleted (using an extra bit field). The number of deleted and nondeleted elements in the list is kept as part of the data structure. If there are as many deleted elements as nondeleted elements, we traverse the entire list, performing the standard deletion algorithm on all marked nodes.

a. List the advantages and disadvantages of lazy deletion.

b. Write routines to implement the standard linked list operations using lazy deletion.

3.18 Write a program to check for balancing symbols in the following languages:

a. Pascal (*begin/end*, ( ), [ ], { }).

b. C (/* */, ( ), [ ], { }).

*c. Explain how to print out an error message that is likely to reflect the probable cause.

3.19 Write a program to evaluate a postfix expression.

3.20 a. Write a program to convert an infix expression which includes '(', ')', '+', '−', '*', and '/' to postfix.

b. Add the exponentiation operator to your repertoire.

c. Write a program to convert a postfix expression to infix.

3.21 Write routines to implement two stacks using only one array. Your stack routines should not declare an overflow unless every slot in the array is used.

3.22 *a. Propose a data structure that supports the stack *Push* and *Pop* operations and a third operation *FindMin*, which returns the smallest element in the data structure, all in $O(1)$ worst case time.

*b. Prove that if we add the fourth operation *DeleteMin* which finds and removes the smallest element, then at least one of the operations must take $\Omega(\log N)$ time. (This requires reading Chapter 7.)

3.23 * Show how to implement three stacks in one array.

3.24 If the recursive routine in Section 2.4 used to compute Fibonacci numbers is run for $N = 50$, is stack space likely to run out? Why or why not?

3.25 Write the routines to implement queues using:

a. Linked lists

b. Arrays

3.26 A *deque* is a data structure consisting of a list of items, on which the following operations are possible:

Push(X,D): Insert item X on the front end of deque D.

Pop(D): Remove the front item from deque D and return it.

Inject(X,D): Insert item X on the rear end of deque D.

Eject(D): Remove the rear item from deque D and return it.

Write routines to support the deque that take $O(1)$ time per operation.

CHAPTER 4

Trees

For large amounts of input, the linear access time of linked lists is prohibitive. In this chapter we look at a simple data structure for which the running time of most operations is $O(\log N)$ on average. We also sketch a conceptually simple modification to this data structure that guarantees the above time bound in the worst case and discuss a second modification that essentially gives an $O(\log N)$ running time per operation for a long sequence of instructions.

The data structure that we are referring to is known as a *binary search tree*. *Trees* in general are very useful abstractions in computer science, so we will discuss their use in other, more general applications. In this chapter, we will

- See how trees are used to implement the file system of several popular operating systems.
- See how trees can be used to evaluate arithmetic expressions.
- Show how to use trees to support searching operations in $O(\log N)$ average time, and how to refine these ideas to obtain $O(\log N)$ worst-case bounds. We will also see how to implement these operations when the data are stored on a disk.

4.1. Preliminaries

A *tree* can be defined in several ways. One natural way to define a tree is recursively. A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node r, called the *root*, and zero or more nonempty (sub)trees T_1, T_2, ..., T_k, each of whose roots are connected by a directed *edge* from r.

The root of each subtree is said to be a *child* of r, and r is the *parent* of each subtree root. Figure 4.1 shows a typical tree using the recursive definition.

From the recursive definition, we find that a tree is a collection of N nodes, one of which is the root, and $N - 1$ edges. That there are $N - 1$ edges follows from the fact that each edge connects some node to its parent, and every node except the root has one parent (see Fig. 4.2).

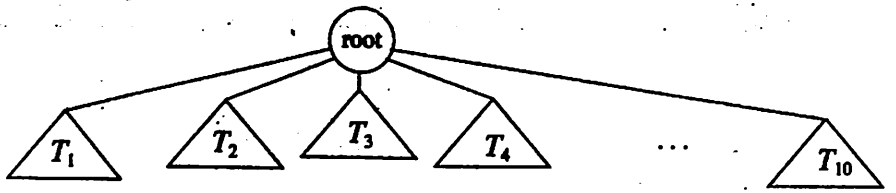


Figure 4.1 Generic tree

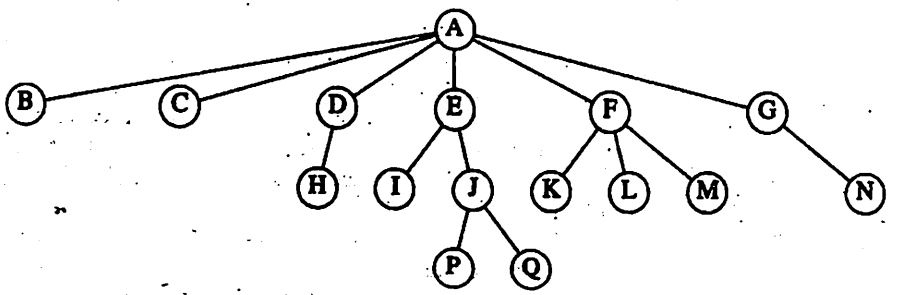


Figure 4.2 A tree

In the tree of Figure 4.2, the root is *A*. Node *F* has *A* as a parent and *K*, *L*, and *M* as children. Each node may have an arbitrary number of children, possibly zero. Nodes with no children are known as *leaves;* the leaves in the tree above are *B*, *C*, *H*, *I*, *P*, *Q*, *K*, *L*, *M*, and *N*. Nodes with the same parent are *siblings;* thus *K*, *L*, and *M* are all siblings. *Grandparent* and *grandchild* relations can be defined in a similar manner.

A *path* from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$. The *length* of this path is the number of edges on the path, namely $k - 1$. There is a path of length zero from every node to itself. Notice that in a tree there is exactly one path from the root to each node.

For any node n_i, the *depth* of n_i is the length of the unique path from the root to n_i. Thus, the root is at depth 0. The *height* of n_i is the length of the longest path from n_i to a leaf. Thus all leaves are at height 0. The height of a tree is equal to the height of the root. For the tree in Figure 4.2, *E* is at depth 1 and height 2; *F* is at depth 1 and height 1; the height of the tree is 3. The depth of a tree is equal to the depth of the deepest leaf; this is always equal to the height of the tree.

If there is a path from n_1 to n_2, then n_1 is an *ancestor* of n_2 and n_2 is a *descendant* of n_1. If $n_1 \neq n_2$, then n_1 is a *proper ancestor* of n_2 and n_2 is a *proper descendant* of n_1.

4.1.1. Implementation of Trees

One way to implement a tree would be to have in each node, besides its data, a pointer to each child of the node. However, since the number of children per node can vary so greatly and is not known in advance, it might be infeasible to make the


```
typedef struct TreeNode *PtrToNode;

struct TreeNode
{
    ElementType Element;
    PtrToNode   FirstChild;
    PtrToNode   NextSibling;
}
```

Figure 4.3 Node declarations for trees

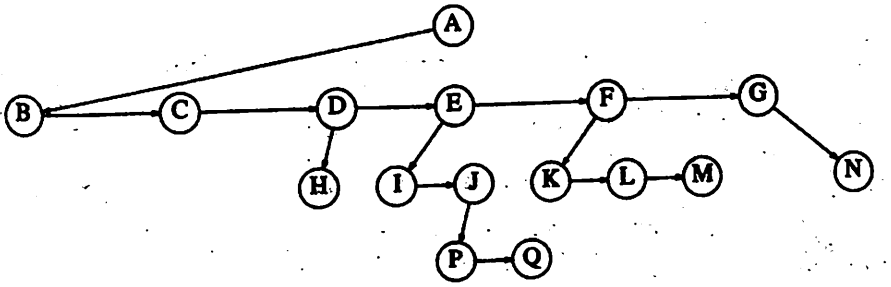


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

children direct links in the data structure, because there would be too much wasted space. The solution is simple: Keep the children of each node in a linked list of tree nodes. The declaration in Figure 4.3 is typical.

Figure 4.4 shows how a tree might be represented in this implementation. Arrows that go left to right are *FirstChild* pointers. Arrows that point downward are *NextSibling* pointers. Null pointers are not drawn, because there are too many.

In the tree of Figure 4.4, node *E* has both a pointer to a sibling (*F*) and a pointer to a child (*I*), while some nodes have neither.

4.1.2. Tree Traversals with an Application

There are many applications for trees. One of the popular uses is the directory structure in many common operating systems, including UNIX, VAX/VMS, and DOS. Figure 4.5 is a typical directory in the UNIX file system.

The root of this directory is */usr*. (The asterisk next to the name indicates that */usr* is itself a directory.) */usr* has three children, *mark*, *alex*, and *bill*, which are themselves directories. Thus, */usr* contains three directories and no regular files. The filename */usr/mark/book/ch1.r* is obtained by following the leftmost child three times. Each / after the first indicates an edge; the result is the full *pathname*. This hierarchical file system is very popular, because it allows users to organize their data logically. Furthermore, two files in different directories can share the same name, because they must have different paths from the root and thus have different

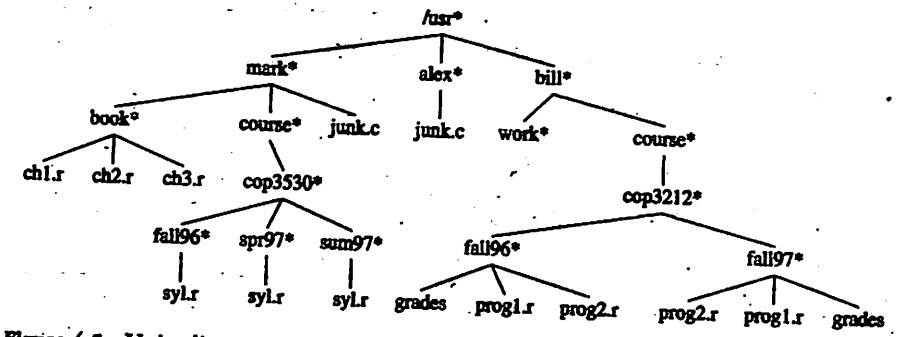


Figure 4.5 Unix directory

```
        static void
        ListDir( DirectoryOrFile D, int Depth )
        {
/* 1*/      if( D is a legitimate entry )
            {
/* 2*/          PrintName( D, Depth );
/* 3*/          if( D is a directory )
/* 4*/              for each child, C, of D
/* 5*/                  ListDir( C, Depth + 1 );
            }
        }

        void
        ListDirectory( DirectoryOrFile D )
        {
            ListDir( D, 0 );
        }
```

Figure 4.6 Routine to list a directory in a hierarchical file system

pathnames. A directory in the UNIX file system is just a file with a list of all its children, so the directories are structured almost exactly in accordance with the type declaration above.* Indeed, if the normal command to print a file is applied to a directory, then the names of the files in the directory can be seen in the output (along with other non-ASCII information).

Suppose we would like to list the names of all of the files in the directory. Our output format will be that files that are depth d_i will have their names indented by d_i tabs. Our algorithm is given in Figure 4.6.

The heart of the algorithm is the recursive procedure *ListDir*. This routine needs to be started with the directory name and a depth of 0, to signify no indenting for

*Each directory in the UNIX file system also has one entry that points to itself and another entry that points to the parent of the directory. Thus, technically, the UNIX file system is not a tree, but is treelike.

the root. This depth is an internal bookkeeping variable, and is hardly a parameter that a calling routine should be expected to know about. Thus the driver routine *ListDirectory* is used to interface the recursive routine to the outside world.

The logic of the algorithm is simple to follow. The argument to *ListDir* is some sort of reference into the tree. As long as the reference is valid, the name implied by the reference is printed out with the appropriate number of tabs. If the entry is a directory, then we process all children recursively, one by one. These children are one level deeper, and thus need to be indented an extra space. The output is in Figure 4.7.

This traversal strategy is known as a *preorder* traversal. In a preorder traversal, work at a node is performed before (*pre*) its children are processed. When this program is run, it is clear that line 2 is executed exactly once per node, since each name is output once. Since line 2 is executed at most once per node, line 3 must also be executed once per node. Furthermore, line 5 can be executed at most once for each child of each node. But the number of children is exactly one less than the

Figure 4.7 The (preorder) directory listing

```
/usr
mark
    book
        chr1.c
        chr2.c
        chr3.c
    course
        cop3530
            fall96
                syl.r
            spr97
                syl.r
            sum97
                syl.r
    junk.c
alex
    junk.c
bill
    work
    course
        cop3212
            fall96
                grades
                prog1.r
                prog2.r
            fall97
                prog2.r
                prog1.r
                grades
```

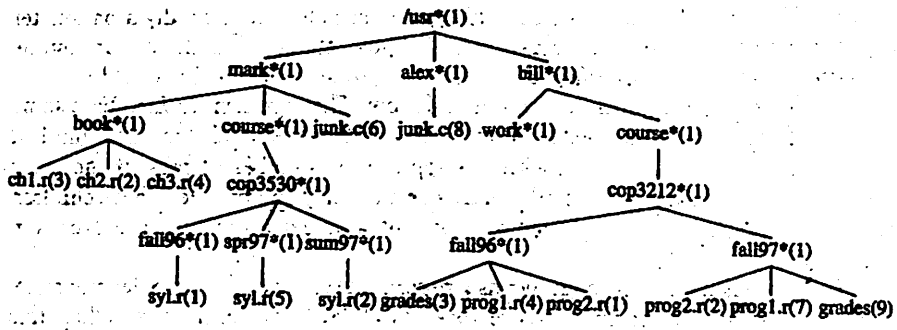


Figure 4.8 Unix directory with file sizes obtained via postorder traversal

number of nodes. Finally, the *for* loop iterates once per execution of line 5, plus once each time the loop ends. Each *for* loop terminates on a *NULL* pointer, but there is at most one of those per node. Thus, the total amount of work is constant per node. If there are N file names to be output, then the running time is $O(N)$.

Another common method of traversing a tree is the *postorder* traversal. In a postorder traversal, the work at a node is performed after (*post*) its children are evaluated. As an example, Figure 4.8 represents the same directory structure as before, with the numbers in parentheses representing the number of disk blocks taken up by each file.

Since the directories are themselves files, they have sizes too. Suppose we would like to calculate the total number of blocks used by all the files in the tree. The most natural way to do this would be to find the number of blocks contained in the subdirectories */usr/mark* (30), */usr/alex* (9), and */usr/bill* (32). The total number of blocks is then the total in the subdirectories (71) plus the one block used by */usr*, for a total of 72. The function *SizeDirectory* in Figure 4.9 implements this strategy.

Figure 4.9 Routine to calculate the size of a directory

```
        static void
        SizeDirectory( DirectoryOrFile D )
        {
            int TotalSize;

/* 1*/      TotalSize = 0;
/* 2*/      if( D is a legitimate entry )
            {
/* 3*/          TotalSize = FileSize( D );
/* 4*/          if( D is a directory )
/* 5*/              for each child, C, of D
/* 6*/                  TotalSize += SizeDirectory( C );
            }
/* 7*/      return TotalSize;
        }
```



```
            ch1.r          3
            ch2.r          2
            ch3.r          4
        book              10
                  syl.r    1
              fall96       2
                  syl.r    5
              spr97        6
                  syl.r    2
              sum97        3
          cop3530          12
      course               13
      junk.c                6
  mark                     30
      junk.c                8
  alex                      9
      work                  1
                  grades   3
                  prog1.r  4
                  prog2.r  1
              fall96       9
                  prog2.r  2
                  prog1.r  7
                  grades   9
              fall97       19
          cop3212          29
      course               30
  bill                     32
  /usr                     72
```

Figure 4.10 Trace of the *SizeDirectory* function

If D is not a directory, then *SizeDirectory* merely returns the number of blocks used by D. Otherwise, the number of blocks used by D is added to the number of blocks (recursively) found in all of the children. To see the difference between the postorder traversal strategy and the preorder traversal strategy, Figure 4.10 shows how the size of each directory or file is produced by the algorithm.

4.2. Binary Trees

A binary tree is a tree in which no node can have more than two children.

Figure 4.11 shows that a binary tree consists of a root and two subtrees, T_L and T_R, both of which could possibly be empty.

A property of a binary tree that is sometimes important is that the depth of an average binary tree is considerably smaller than N. An analysis shows that the

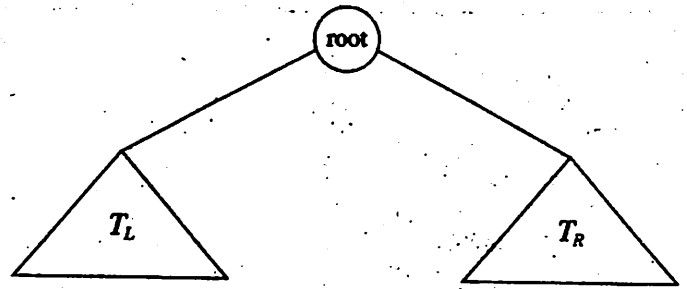


Figure 4.11 Generic binary tree

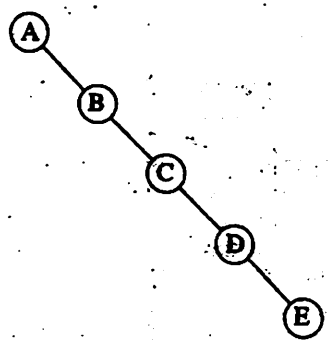


Figure 4.12 Worst-case binary tree

average depth is $O(\sqrt{N})$, and that for a special type of binary tree, namely the *binary search tree*, the average value of the depth is $O(\log N)$. Unfortunately, the depth can be as large as $N - 1$, as the example in Figure 4.12 shows.

4.2.1. Implementation

Because a binary tree has at most two children, we can keep direct pointers to them. The declaration of tree nodes is similar in structure to that for doubly linked lists, in that a node is a structure consisting of the *Key* information plus two pointers (*Left* and *Right*) to other nodes (see Fig. 4.13).

Many of the rules that apply to linked lists will apply to trees as well. In particular, when an insertion is performed, a node will have to be created by a call to *malloc*. Nodes can be freed after deletion by calling *free*.

We could draw the binary trees using the rectangular boxes that are customary for linked lists, but trees are generally drawn as circles connected by lines, because they are actually graphs. We also do not explicitly draw *NULL* pointers when


```
typedef struct TreeNode *PtrToNode;
typedef struct PtrToNode Tree;

struct TreeNode
{
    ElementType Element;
    Tree        Left;
    Tree        Right;
};
```

Figure 4.13 Binary tree node declarations

referring to trees, because every binary tree with N nodes would require $N + 1$ *NULL* pointers.

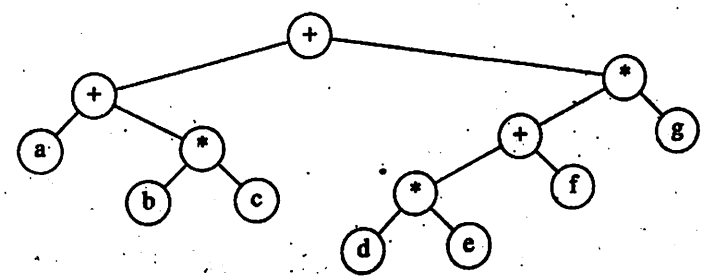
Binary trees have many important uses not associated with searching. One of the principal uses of binary trees is in the area of compiler design, which we will now explore.

4.2.2. Expression Trees

Figure 4.14 shows an example of an *expression tree*. The leaves of an expression tree are *operands*, such as constants or variable names, and the other nodes contain *operators*. This particular tree happens to be binary, because all of the operations are binary, and although this is the simplest case, it is possible for nodes to have more than two children. It is also possible for a node to have only one child, as is the case with the *unary minus* operator. We can evaluate an expression tree, T, by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees. In our example, the left subtree evaluates to $a + (b * c)$ and the right subtree evaluates to $((d * e) + f) * g$. The entire tree therefore represents $(a + (b * c)) + (((d * e) + f) * g)$.

We can produce an (overly parenthesized) infix expression by recursively producing a parenthesized left expression, then printing out the operator at the

Figure 4.14 Expression tree for $(a + b * c) + ((d * e + f) * g)$



root, and finally recursively producing a parenthesized right expression. This general strategy (left, node, right) is known as an *inorder* traversal; it is easy to remember because of the type of expression it produces.

An alternate traversal strategy is to recursively print out the left subtree, the right subtree, and then the operator. If we apply this strategy to our tree above, the output is a b c * + d e * f + g * +, which is easily seen to be the postfix representation of Section 3.3.3. This traversal strategy is generally known as a *postorder* traversal. We have seen this traversal strategy earlier in Section 4.1.

A third traversal strategy is to print out the operator first and then recursively print out the left and right subtrees. The resulting expression, + + a * b c * + * d e f g, is the less useful *prefix* notation and the traversal strategy is a *preorder* traversal, which we have also seen earlier in Section 4.1. We will return to these traversal strategies later in the chapter.

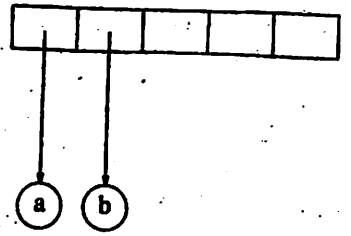
Constructing an Expression Tree

We now give an algorithm to convert a postfix expression into an expression tree. Since we already have an algorithm to convert infix to postfix, we can generate expression trees from the two common types of input. The method we describe strongly resembles the postfix evaluation algorithm of Section 3.2.3. We read our expression one symbol at a time. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack. If the symbol is an operator, we pop pointers to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1, respectively. A pointer to this new tree is then pushed onto the stack.

As an example, suppose the input is

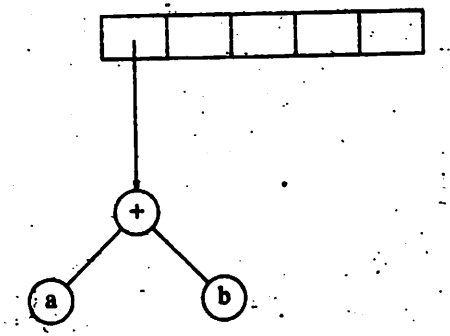
a b + c d e + * *

The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.*

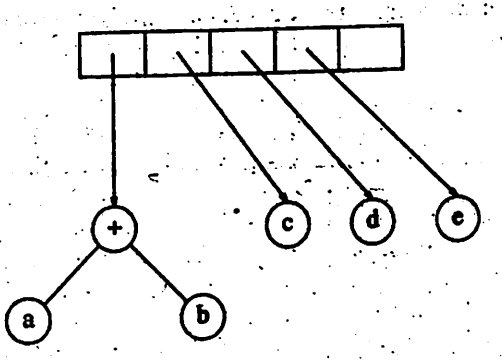


Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

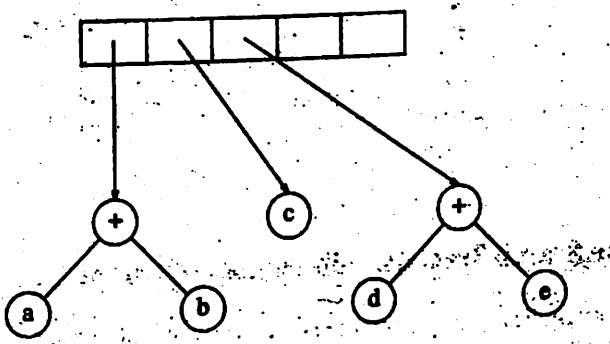
*For convenience, we will have the stack grow from left to right in the diagrams.



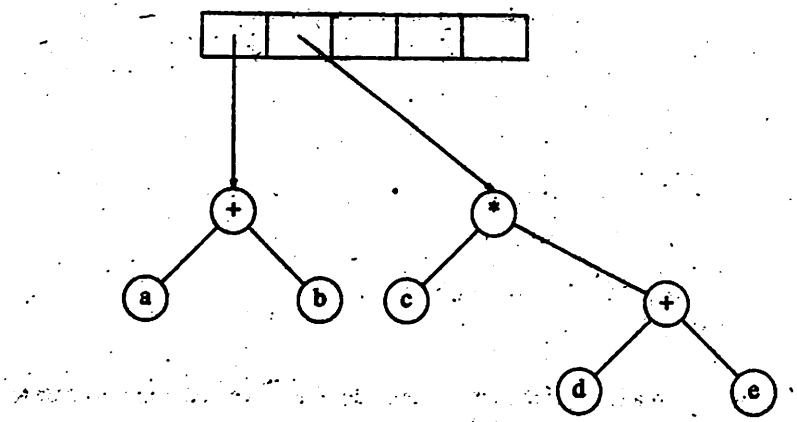
Next, c, d, and e are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



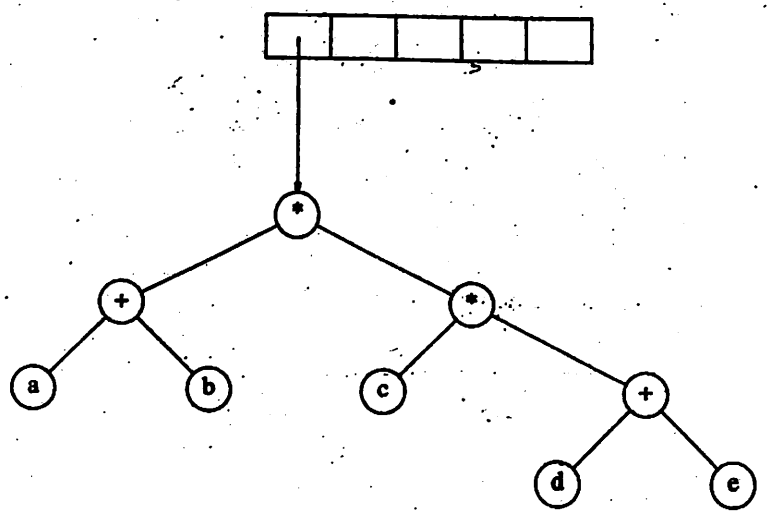
Now a '+' is read, so two trees are merged.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



4.3. The Search Tree ADT—Binary Search Trees

An important application of binary trees is their use in searching. Let us assume that each node in the tree is assigned a key value. In our examples, we will assume for simplicity that these are integers, although arbitrarily complex keys are allowed. We will also assume that all the keys are distinct, and deal with duplicates later.

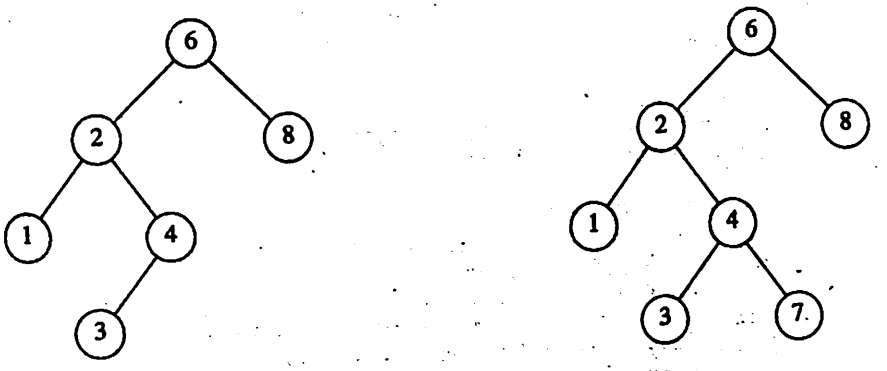


Figure 4.15 Two binary trees (only the left tree is a search tree)

The property that makes a binary tree into a binary search tree is that for every node, X, in the tree, the values of all the keys in its left subtree are smaller than the key value in X, and the values of all the keys in its right subtree are larger than the key value in X. Notice that this implies that all the elements in the tree can be ordered in some consistent manner. In Figure 4.15, the tree on the left is a binary search tree, but the tree on the right is not. The tree on the right has a node with key 7 in the left subtree of a node with key 6 (which happens to be the root).

We now give brief descriptions of the operations that are usually performed on binary search trees. Note that because of the recursive definition of trees, it is common to write these routines recursively. Because the average depth of a binary search tree is $O(\log N)$, we generally do not need to worry about running out of stack space. We repeat our type definition in Figure 4.16 and list the function prototypes. Since all the elements can be ordered, we will assume that the operators $<$, $>$, and $=$ can be applied to them, even if this might be syntactically erroneous for some types.

4.3.1. *MakeEmpty*

This operation is mainly for initialization. Some programmers prefer to initialize the first element as a one-node tree, but our implementation follows the recursive definition of trees more closely. It is also a simple routine, as evidenced by Figure 4.17.

4.3.2. *Find*

This operation generally requires returning a pointer to the node in tree T that has key X, or *NULL* if there is no such node. The structure of the tree makes this simple. If T is *NULL*, then we can just return *NULL*. Otherwise, if the key stored at T is X, we can return T. Otherwise, we make a recursive call on a subtree of T, either left or right, depending on the relationship of X to the key stored in T. The code in Figure 4.18 is an implementation of this strategy.

```
#ifndef _Tree_H

struct TreeNode;
typedef struct TreeNode *Position;
typedef struct TreeNode *SearchTree;

SearchTree MakeEmpty( SearchTree T );
Position Find( ElementType X, SearchTree T );
Position FindMin( SearchTree T );
Position FindMax( SearchTree T );
SearchTree Insert( ElementType X, SearchTree T );
SearchTree Delete( ElementType X, SearchTree T );
ElementType Retrieve( Position P );

#endif /* _Tree_H */


/* Place in the implementation file */
struct TreeNode
{
    ElementType Element;
    SearchTree  Left;
    SearchTree  Right;
};
```

Figure 4.16 Binary search tree declarations

```
SearchTree
MakeEmpty( SearchTree T )
{
    if( T != NULL )
    {
        MakeEmpty( T->Left );
        MakeEmpty( T->Right );
        free( T );
    }
    return NULL;
}
```

Figure 4.17 Routine to make an empty tree


```
Position
Find( ElementType X, SearchTree T )
{
    if( T == NULL )
        return NULL;
    if( X < T->Element )
        return Find( X, T->Left );
    else
    if( X > T->Element )
        return Find( X, T->Right );
    else
        return T;
}
```

Figure 4.18 *Find* operation for binary search trees

Notice the order of the tests. It is crucial that the test for an empty tree be performed first, since otherwise the indirections would be on a *NULL* pointer. The remaining tests are arranged with the least likely case last. Also note that both recursive calls are actually tail recursions and can be easily removed with an assignment and a *goto*. The use of tail recursion is justifiable here because the simplicity of algorithmic expression compensates for the decrease in speed, and the amount of stack space used is expected to be only $O(\log N)$.

4.3.3. *FindMin and FindMax*

These routines return the position of the smallest and largest elements in the tree, respectively. Although returning the exact values of these elements might seem more reasonable, this would be inconsistent with the *Find* operation. It is important that similar-looking operations do similar things. To perform a *FindMin*, start at the root and go left as long as there is a left child. The stopping point is the smallest element. The *FindMax* routine is the same, except that branching is to the right child.

Figure 4.19 Recursive implementation of *FindMin* for binary search trees

```
Position
FindMin( SearchTree T )
{
    if( T == NULL )
        return NULL;
    else
    if( T->Left == NULL )
        return T;
    else
        return FindMin( T->Left );
}
```

```
Position
FindMax( SearchTree T )
{
    if( T != NULL )
        while( T->Right != NULL )
            T = T->Right;

    return T;
}
```

Figure 4.20 Nonrecursive implementation of *FindMax* for binary search trees

This is so easy that many programmers do not bother using recursion. We will code the routines both ways by doing *FindMin* recursively and *FindMax* nonrecursively (see Figs. 4.19 and 4.20).

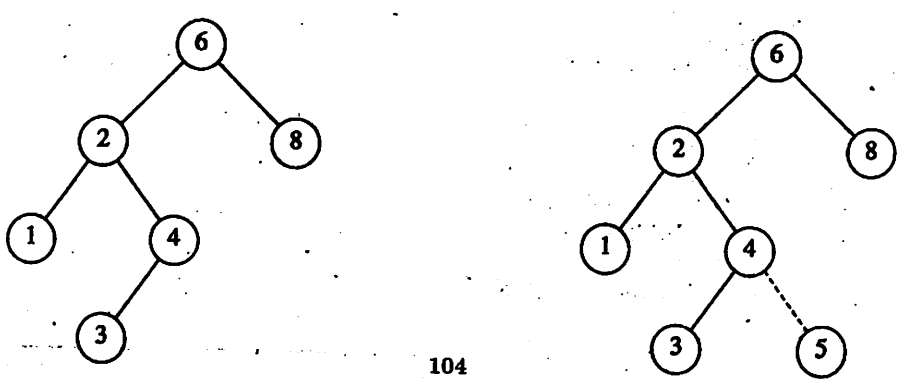
Notice how we carefully handle the degenerate case of an empty tree. Although this is always important to do, it is especially crucial in recursive programs. Also notice that it is safe to change T in *FindMax*, since we are only working with a copy. Always be extremely careful, however, because a statement such as T->*Right* = T->*Right*->*Right* will make changes.

4.3.4. *Insert*

The insertion routine is conceptually simple. To insert X into tree T, proceed down the tree as you would with a *Find*. If X is found, do nothing (or "update" something). Otherwise, insert X at the last spot on the path traversed. Figure 4.21 shows what happens. To insert 5, we traverse the tree as though a *Find* were occurring. At the node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct spot.

Duplicates can be handled by keeping an extra field in the node record indicating the frequency of occurrence. This adds some extra space to the entire tree, but is

Figure 4.21 Binary search trees before and after inserting 5




```
        SearchTree
        Insert( ElementType X, SearchTree T )
        {
/* 1*/      if( T == NULL )
            {
                /* Create and return a one-node tree */
/* 2*/          T = malloc( sizeof( struct TreeNode ) );
/* 3*/          if( T == NULL )
/* 4*/              FatalError( "Out of space!!!" );
                else
                {
/* 5*/                  T->Element = X;
/* 6*/                  T->Left = T->Right = NULL;
                }
            }
            else
/* 7*/      if( X < T->Element )
/* 8*/          T->Left = Insert( X, T->Left );
            else
/* 9*/      if( X > T->Element )
/*10*/          T->Right = Insert( X, T->Right );
            /* Else X is in the tree already; we'll do nothing */

/*11*/      return T;  /* Do not forget this line!! */
        }
```

Figure 4.22 Insertion into a binary search tree

better than putting duplicates in the tree (which tends to make the tree very deep). Of course this strategy does not work if the key is only part of a larger structure. If that is the case, then we can keep all of the structures that have the same key in an auxiliary data structure, such as a list or another search tree.

Figure 4.22 shows the code for the insertion routine. Since T points to the root of the tree, and the root changes on the first insertion, *Insert* is written as a function that returns a pointer to the root of the new tree. Lines 8 and 10 recursively insert and attach X into the appropriate subtree.

4.3.5. *Delete*

As is common with many data structures, the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities.

If the node is a leaf, it can be deleted immediately. If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node (we will draw the pointer directions explicitly for clarity). See Figure 4.23. Notice that the deleted node is now unreferenced and can be disposed of only if a pointer to it has been saved.

The complicated case deals with a node with two children. The general strategy is to replace the data of this node with the smallest data of the right subtree (which

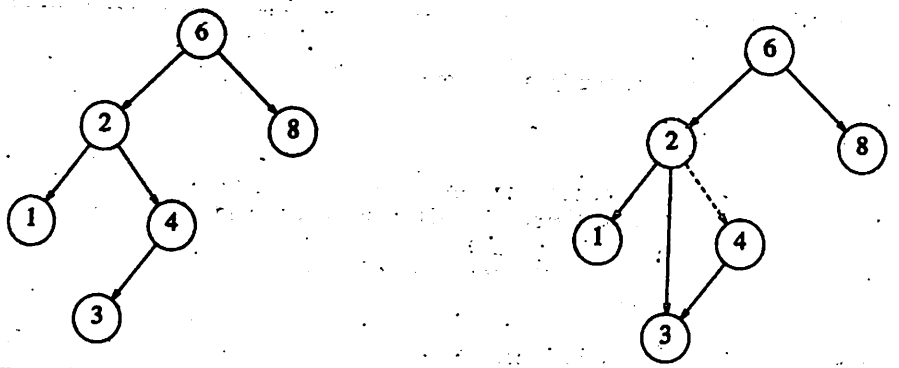


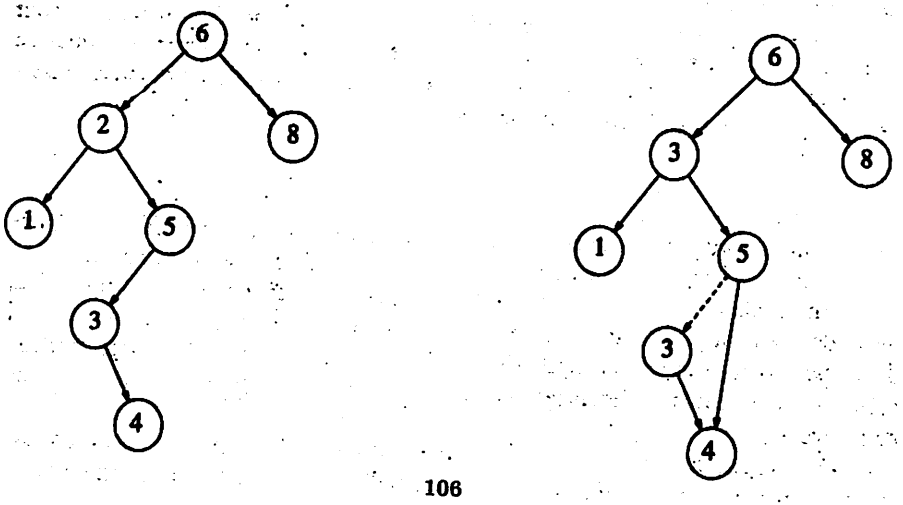
Figure 4.23 Deletion of a node (4) with one child, before and after

is easily found) and recursively delete that node (which is now empty). Because the smallest node in the right subtree cannot have a left child, the second *Delete* is an easy one. Figure 4.24 shows an initial tree and the result of a deletion. The node to be deleted is the left child of the root; the key value is 2. It is replaced with the smallest data in its right subtree (3), and then that node is deleted as before.

The code in Figure 4.25 performs deletion. It is inefficient, because it makes two passes down the tree to find and delete the smallest node in the right subtree when this is appropriate. It is easy to remove this inefficiency, by writing a special *DeleteMin* function, and we have left it in only for simplicity.

If the number of deletions is expected to be small, then a popular strategy to use is *lazy deletion:* When an element is to be deleted, it is left in the tree and merely *marked* as being deleted. This is especially popular if duplicate keys are

Figure 4.24 Deletion of a node (2) with two children, before and after




```
SearchTree
Delete( ElementType X, SearchTree T )
{
    Position TmpCell;

    if( T == NULL )
        Error( "Element not found" );
    else
    if( X < T->Element )  /* Go left */
        T->Left = Delete( X, T->Left );
    else
    if( X > T->Element )  /* Go right */
        T->Right = Delete( X, T->Left );
    else  /* Found element to be deleted */
    if( T->Left && T->Right )  /* Two children */
    {
        /* Replace with smallest in right subtree */
        TmpCell = FindMin( T->Right );
        T->Element = TmpCell->Element;
        T->Right = Delete( T->Element, T->Right );
    }
    else  /* One or zero children */
    {
        TmpCell = T;
        if( T->Left == NULL ) /* Also handles 0 children */
            T = T->Right;
        else if( T->Right == NULL )
            T = T->Left;
        free( TmpCell );
    }

    return T;
}
```

Figure 4.25 Deletion routine for binary search trees

present, because then the field that keeps count of the frequency of appearance can be decremented. If the number of real nodes in the tree is the same as the number of "deleted" nodes, then the depth of the tree is only expected to go up by a small constant (why?), so there is a very small time penalty associated with lazy deletion. Also, if a deleted key is reinserted, the overhead of allocating a new cell is avoided.

4.3.6. *Average-Case Analysis*

Intuitively, we expect that all of the operations of the previous section, except *MakeEmpty*, should take $O(\log N)$ time, because in constant time we descend a

level in the tree, thus operating on a tree that is now roughly half as large. Indeed, the running time of all the operations, except *MakeEmpty*, is $O(d)$, where d is the depth of the node containing the accessed key.

We prove in this section that the average depth over all nodes in a tree is $O(\log N)$ on the assumption that all trees are equally likely.

The sum of the depths of all nodes in a tree is known as the *internal path length*. We will now calculate the average internal path length of a binary search tree, where the average is taken over all possible insertion sequences into binary search trees.

Let $D(N)$ be the internal path length for some tree T of N nodes. $D(1) = 0$. An N-node tree consists of an i-node left subtree and an $(N - i - 1)$-node right subtree, plus a root at depth zero for $0 \leq i < N$. $D(i)$ is the internal path length of the left subtree with respect to its root. In the main tree, all these nodes are one level deeper. The same holds for the right subtree. Thus, we get the recurrence

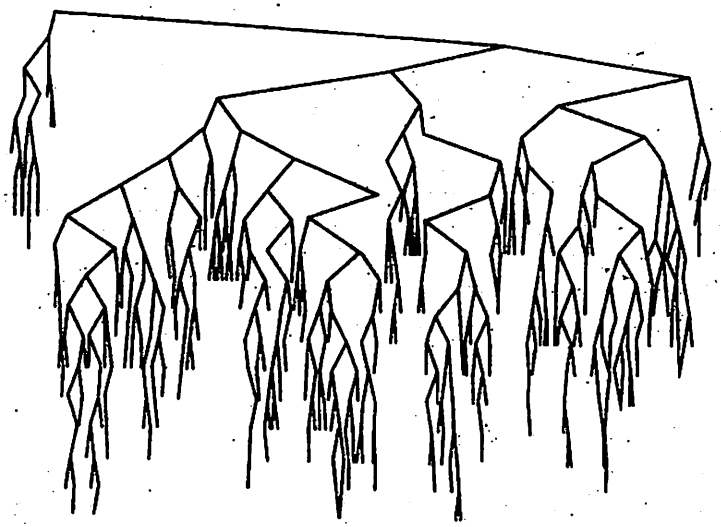
$$D(N) = D(i) + D(N - i - 1) + N - 1$$

If all subtree sizes are equally likely, which is true for binary search trees (since the subtree size depends only on the relative rank of the first element inserted into the tree), but not binary trees, then the average value of both $D(i)$ and $D(N - i - 1)$ is $(1/N)\sum_{j=0}^{N-1} D(j)$. This yields

$$D(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} D(j)\right] + N - 1$$

This recurrence will be encountered and solved in Chapter 7, obtaining an average value of $D(N) = O(N \log N)$. Thus, the expected depth of any node is $O(\log N)$. As an example, the randomly generated 500-node tree shown in Figure 4.26 has nodes at expected depth 9.98.

Figure 4.26 A randomly generated binary search tree



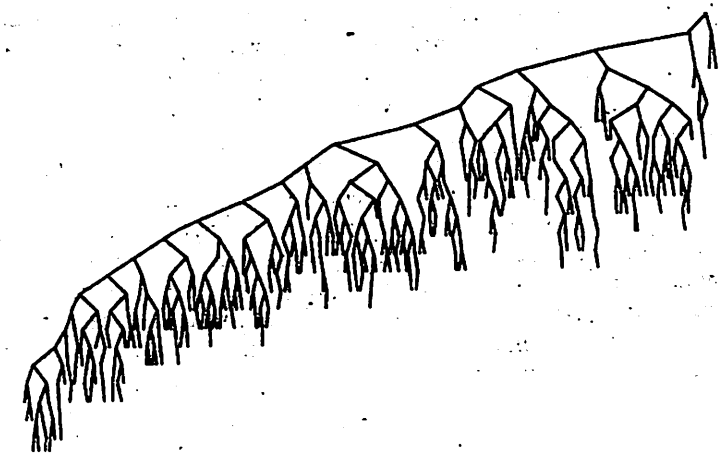


Figure 4.27 Binary search tree after $\Theta(N^2)$ *Insert/Delete* pairs

It is tempting to say immediately that this result implies that the average running time of all the operations discussed in the previous section is $O(\log N)$, but this is not entirely true. The reason for this is that because of deletions, it is not clear that all binary search trees are equally likely. In particular, the deletion algorithm described above favors making the left subtrees deeper than the right, because we are always replacing a deleted node with a node from the right subtree. The exact effect of this strategy is still unknown, but it seems only to be a theoretical novelty. It has been shown that if we alternate insertions and deletions $\Theta(N^2)$ times, then the trees will have an expected depth of $\Theta(\sqrt{N})$. After a quarter-million random *Insert/Delete* pairs, the tree that was somewhat right-heavy in Figure 4.26 looks decidedly unbalanced (average depth = 12.51). See Figure 4.27.

We could try to eliminate the problem by randomly choosing between the smallest element in the right subtree and the largest in the left when replacing the deleted element. This apparently eliminates the bias and should keep the trees balanced, but nobody has actually proved this. In any event, this phenomenon appears to be mostly a theoretical novelty, because the effect does not show up at all for small trees, and stranger still, if $o(N^2)$ *Insert/Delete* pairs are used, then the tree seems to gain balance!

The main point of this discussion is that deciding what "average" means is generally extremely difficult and can require assumptions that may or may not be valid. In the absence of deletions, or when lazy deletion is used, it can be shown that all binary search trees are equally likely and we can conclude that the average running times of the operations above are $O(\log N)$. Except for strange cases like the one discussed above, this result is very consistent with observed behavior.

If the input comes into a tree presorted, then a series of *Insert*s will take quadratic time and give a very expensive implementation of a linked list, since the tree will consist only of nodes with no left children. One solution to the problem is to insist on an extra structural condition called *balance*: no node is allowed to get too deep.

There are quite a few general algorithms to implement balanced trees. Most are quite a bit more complicated than a standard binary search tree, and all take longer on average for updates. They do, however, provide protection against the embarrassingly simple cases. Below, we will sketch one of the oldest forms of balanced search trees, the AVL tree.

A second, newer method is to forego the balance condition and allow the tree to be arbitrarily deep, but after every operation, a restructuring rule is applied that tends to make future operations efficient. These types of data structures are generally classified as *self-adjusting*. In the case of a binary search tree, we can no longer guarantee an $O(\log N)$ bound on any single operation, but can show that any *sequence* of M operations takes total time $O(M \log N)$ in the worst case. This is generally sufficient protection against a bad worst case. The data structure we will discuss is known as a *splay tree;* its analysis is fairly intricate and is discussed in Chapter 11.

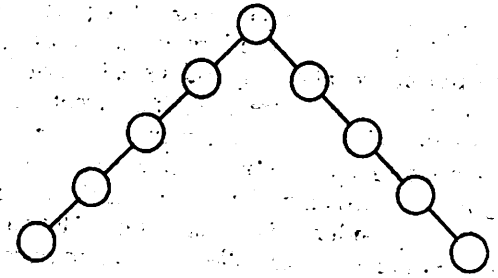
4.4. AVL Trees

An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a *balance* condition. The balance condition must be easy to maintain, and it ensures that the depth of the tree is $O(\log N)$. The simplest idea is to require that the left and right subtrees have the same height. As Figure 4.28 shows, this idea does not force the tree to be shallow.

Another balance condition would insist that every node must have left and right subtrees of the same height. If the height of an empty subtree is defined to be -1 (as is usual), then only perfectly balanced trees of $2^k - 1$ nodes would satisfy this criterion. Thus, although this guarantees trees of small depth, the balance condition is too rigid to be useful and needs to be relaxed.

An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. (The height of an empty tree is defined to be -1.) In Figure 4.29 the tree on the left is an AVL tree, but the tree on the right is not. Height information is kept for each node (in the node structure). It can be shown that the height of an AVL tree is at most roughly

Figure 4.28 A bad binary tree. Requiring balance at the root is not enough.



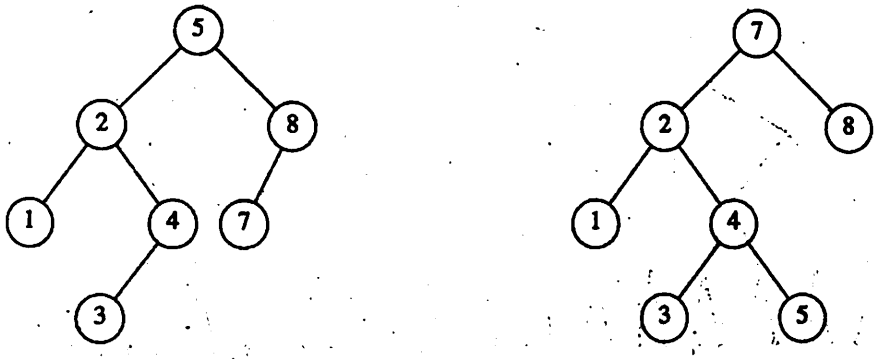


Figure 4.29 Two binary search trees. Only the left tree is AVL.

$1.44 \log(N + 2) - .328$, but in practice it is only slightly more than $\log N$. As an example, the AVL tree of height 9 with the fewest nodes (143) is shown in Figure 4.30. This tree has as a left subtree an AVL tree of height 7 of minimum size. The right subtree is an AVL tree of height 8 of minimum size. This tells us that the minimum number of nodes, $S(h)$, in an AVL tree of height h is given by $S(h) = S(h-1) + S(h-2) + 1$. For $h = 0$, $S(h) = 1$. For $h = 1$, $S(h) = 2$. The function $S(h)$ is closely related to the Fibonacci numbers, from which the bound claimed above on the height of an AVL tree follows.

Thus, all the tree operations can be performed in $O(\log N)$ time, except possibly insertion (we will assume lazy deletion). When we do an insertion, we need to update all the balancing information for the nodes on the path back to the root, but the reason that insertion is potentially difficult is that inserting a node could violate the AVL tree property. (For instance, inserting 6 into the AVL tree in Figure 4.29 would destroy the balance condition at the node with key 8.) If this is the case, then the property has to be restored before the insertion step is considered over. It turns out that this can always be done with a simple modification to the tree, known as a *rotation*.

After an insertion, only nodes that are on the path from the insertion point to the root might have their balance altered because only those nodes have their subtrees altered. As we follow the path up to the root and update the balancing information, we may find a node whose new balance violates the AVL condition. We will show how to rebalance the tree at the first (i.e., deepest) such node, and we will prove that this rebalancing guarantees that the entire tree satisfies the AVL property.

Let us call the node that must be rebalanced α. Since any node has at most two children, and a height imbalance requires that α's two subtrees' height differ by two, it is easy to see that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of α.
2. An insertion into the right subtree of the left child of α.
3. An insertion into the left subtree of the right child of α.
4. An insertion into the right subtree of the right child of α.

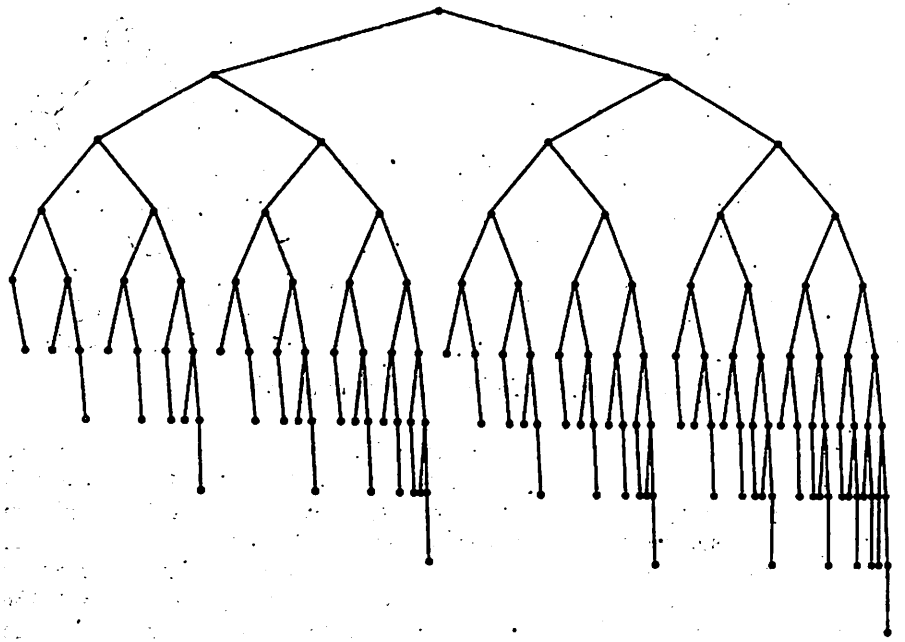


Figure 4.30 Smallest AVL tree of height 9

Cases 1 and 4 are mirror image symmetries with respect to α, as are cases 2 and 3. Consequently, as a matter of theory, there are two basic cases. From a programming perspective, of course, there are still four cases.

The first case, in which the insertion occurs on the "outside" (i.e., left–left or right–right), is fixed by a *single rotation* of the tree. The second case, in which the insertion occurs on the "inside" (i.e., left–right or right–left) is handled by the slightly more complex *double rotation*. These are fundamental operations on the tree that we'll see used several times in balanced-tree algorithms. The remainder of this section describes these rotations, proves that they suffice to maintain balance, and gives a casual implementation of the AVL tree. Chapter 12 describes other balanced-tree methods with an eye toward a more careful implementation.

4.4.1. Single Rotation

Figure 4.31 shows the *single rotation* that fixes case 1. The before picture is on the left, and the after is on the right. Let us analyze carefully what is going on. Node k_2 violates the AVL balance property because its left subtree is two levels deeper than its right subtree (the dashed lines in the middle of the diagram mark the levels). The situation depicted is the only possible case 1 scenario that allows k_2 to satisfy the AVL property before an insertion but violate it afterwards. Subtree X has grown to an extra level, causing it to be exactly two levels deeper than Z. Y cannot be at the same level as the new X because then k_2 would have been out of balance *before*

the insertion, and Y cannot be at the same level as Z because then k_1 would be the first node on the path toward the root that was in violation of the AVL balancing condition.

To ideally rebalance the tree, we would like to move X up a level and Z down a level. Note that this is actually more than the AVL property would require. To do this, we rearrange nodes into an equivalent tree as shown in the second part of Figure 4.31. Here is an abstract scenario: visualize the tree as being flexible, grab the child node k_1, close your eyes, and shake it, letting gravity take hold. The result is that k_1 will be the new root. The binary search tree property tells us that in the original tree $k_2 > k_1$, so k_2 becomes the right child of k_1 in the new tree. X and Z remain as the left child of k_1 and right child of k_2, respectively. Subtree Y, which holds items that are between k_1 and k_2 in the original tree, can be placed as k_2's left child in the new tree and satisfy all the ordering requirements.

As a result of this work, which requires only a few pointer changes, we have another binary search tree that is an AVL tree. This happens because X moves up one level, Y stays at the same level, and Z moves down one level. k_2 and k_1 not only satisfy the AVL requirements, but they also have subtrees that are exactly the same height. Furthermore, the new height of the entire subtree is *exactly the same* as the height of the original subtree prior to the insertion that caused X to grow. Thus no further updating of heights on the path to the root is needed, and consequently *no further rotations are needed*. Figure 4.32 shows that after the insertion of 6 into

Figure 4.31 Single rotation to fix case 1

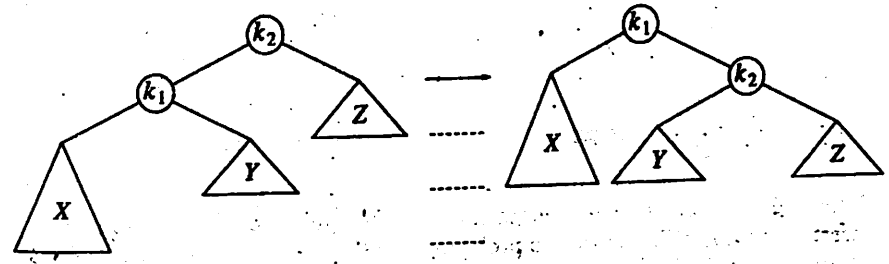
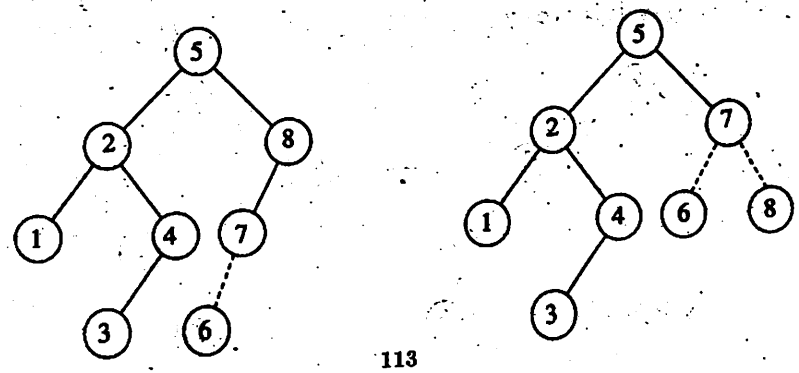


Figure 4.32 AVL property destroyed by insertion of 6, then fixed by a single rotation



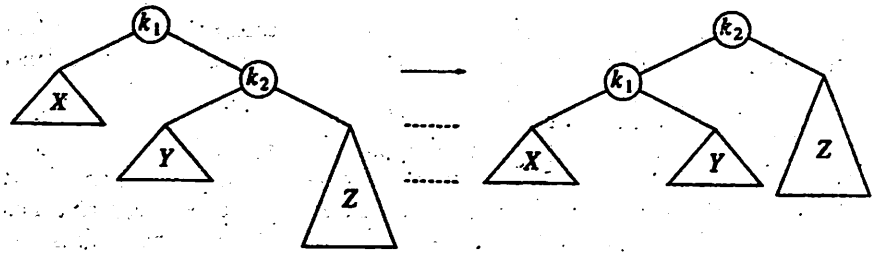
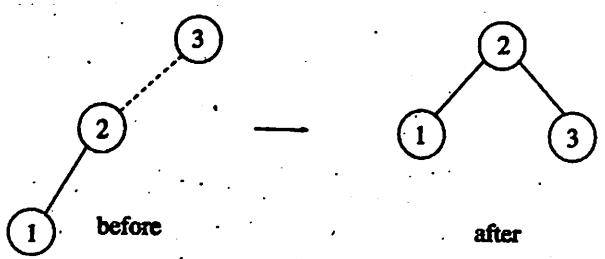


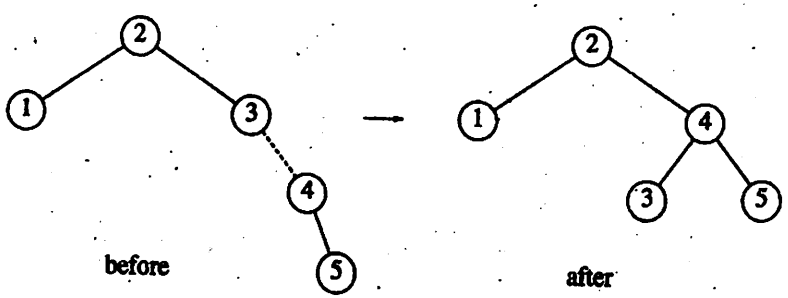
Figure 4.33 Single rotation fixes case 4

the original AVL tree on the left, node 8 becomes unbalanced. Thus, we do a single rotation between 7 and 8, obtaining the tree on the right.

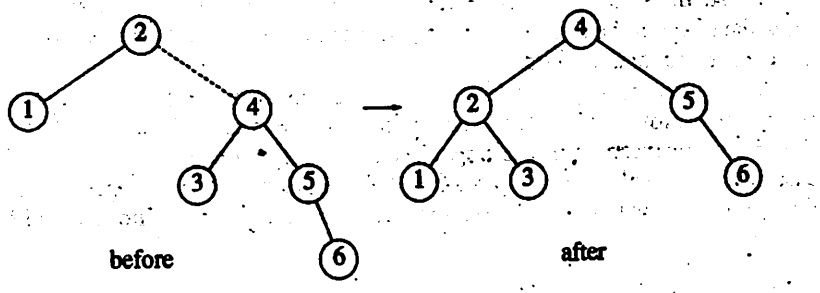
As we mentioned earlier, case 4 represents a symmetric case. Figure 4.33 shows how a single rotation is applied. Let us work through a rather long example. Suppose we start with an initially empty AVL tree and insert the keys 3, 2, 1, and then 4 through 7 in sequential order. The first problem occurs when it is time to insert key 1 because the AVL property is violated at the root. We perform a single rotation between the root and its left child to fix the problem. Here are the before and after trees:



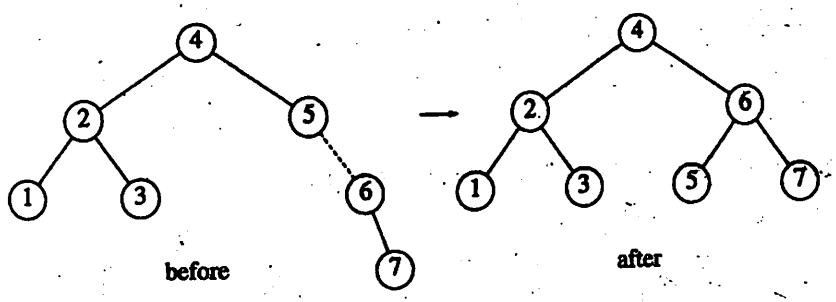
A dashed line joins the two nodes that are the subject of the rotation. Next we insert the key 4, which causes no problems, but the insertion of 5 creates a violation at node 3 that is fixed by a single rotation. Besides the local change caused by the rotation, the programmer must remember that the rest of the tree has to be informed of this change. Here this means that 2's right child must be reset to point to 4 instead of 3. Forgetting to do so is easy and would destroy the tree (4 would be inaccessible).



Next we insert 6. This causes a balance problem at the root, since its left subtree is of height 0 and its right subtree would be height 2. Therefore, we perform a single rotation at the root between 2 and 4.



The rotation is performed by making 2 a child of 4 and 4's original left subtree the new right subtree of 2. Every key in this subtree must lie between 2 and 4, so this transformation makes sense. The next key we insert is 7, which causes another rotation:



4.4.2. *Double Rotation*

The algorithm described above has one problem: as Figure 4.34 shows, it does not work for cases 2 or 3. The problem is that subtree Y is too deep, and a single rotation does not make it any less deep. The *double rotation* that solves the problem is shown in Figure 4.35.

The fact that subtree Y in Figure 4.34 has had an item inserted into it guarantees that it is nonempty. Thus, we may assume that it has a root and two subtrees. Consequently, the tree may be viewed as four subtrees connected by three nodes. As the diagram suggests, exactly one of tree B or C is two levels deeper than D (unless all are empty), but we cannot be sure which one. It turns out not to matter; in Figure 4.35, both B and C are drawn at $1\frac{1}{2}$ levels below D.

To rebalance, we see that we cannot leave k_3 as the root, and a rotation between k_3 and k_1 was shown in Figure 4.34 to not work, so the only alternative is to place k_2 as the new root. This forces k_1 to be k_2's left child and k_3 to be its right child,

and it also completely determines the resulting locations of the four subtrees. It is easy to see that the resulting tree satisfies the AVL tree property, and as was the case with the single rotation, it restores the height to what it was before the insertion, thus guaranteeing that all rebalancing and height updating is complete. Figure 4.36 shows that the symmetric case 3 can also be fixed by a double rotation. In both cases the effect is the same as rotating between α's child and grandchild, and then between α and its new child.

We will continue our previous example by inserting the keys 10 through 16 in reverse order, followed by 8 and then 9. Inserting 16 is easy, since it does not destroy the balance property, but inserting 15 causes a height imbalance at node 7. This is case 3, which is solved by a right–left double rotation. In our example, the right–left double rotation will involve 7, 16, and 15. In this case, k_1 is the node with key 7, k_3

Figure 4.34 Single rotation fails to fix case 2

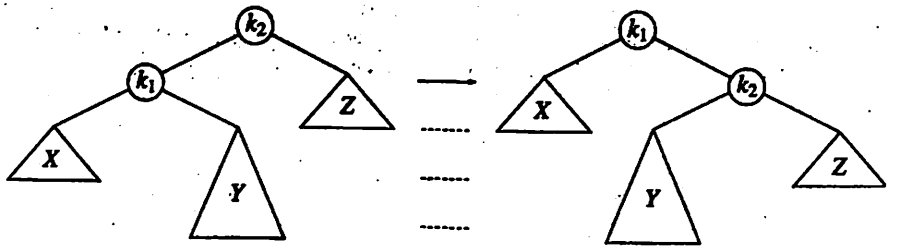


Figure 4.35 Left–right double rotation to fix case 2

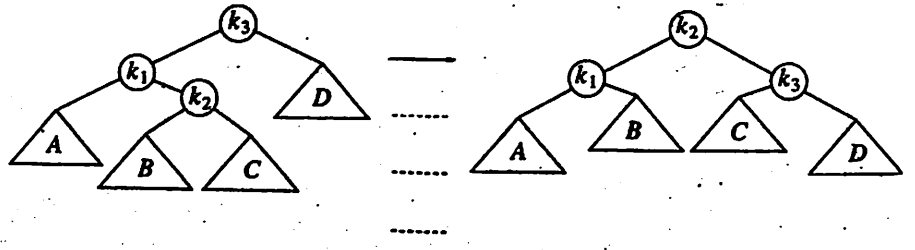
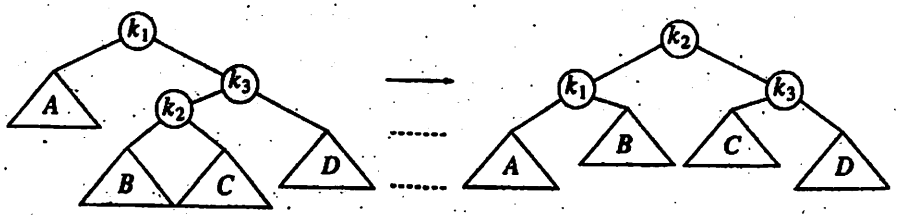
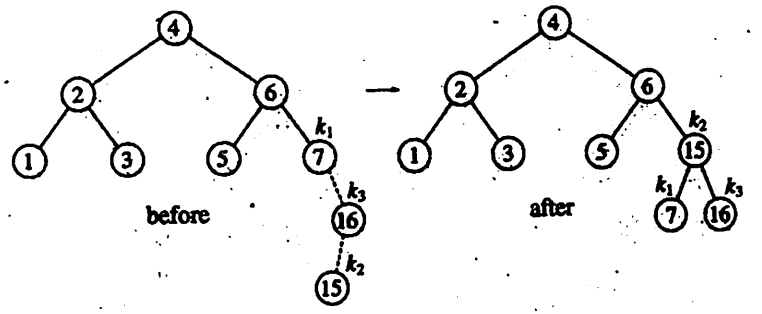


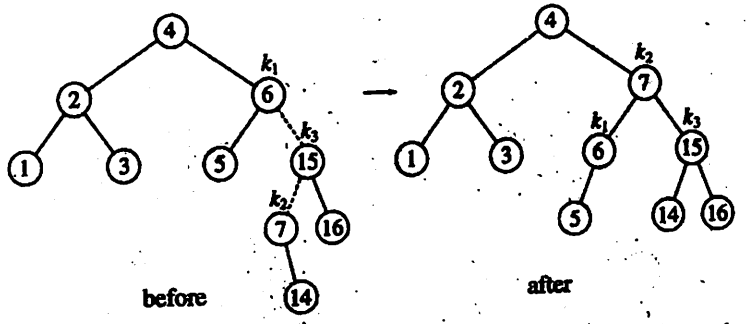
Figure 4.36 Right–left double rotation to fix case 3



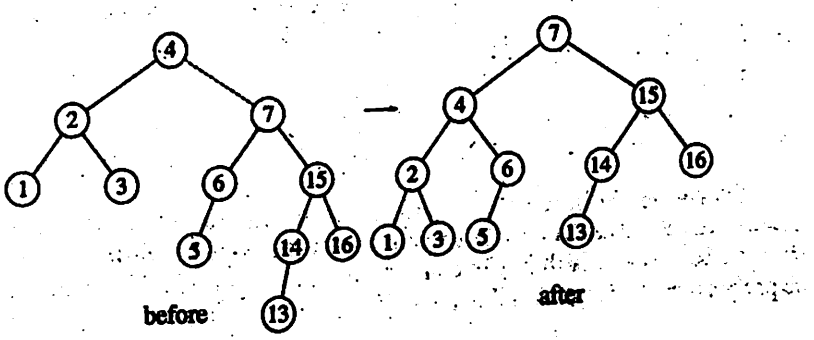
is the node with key 16, and k_2 is the node with key 15. Subtrees A, B, C, and D are empty.



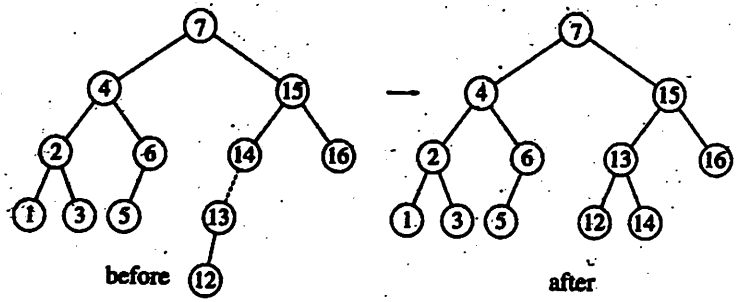
Next we insert 14, which also requires a double rotation. Here the double rotation that will restore the tree is again a right–left double rotation that will involve 6, 15, and 7. In this case, k_1 is the node with key 6, k_2 is the node with key 7, and k_3 is the node with key 15. Subtree A is the tree rooted at the node with key 5; subtree B is the empty subtree that was originally the left child of the node with key 7, subtree C is the tree rooted at the node with key 14, and finally, subtree D is the tree rooted at the node with key 16.



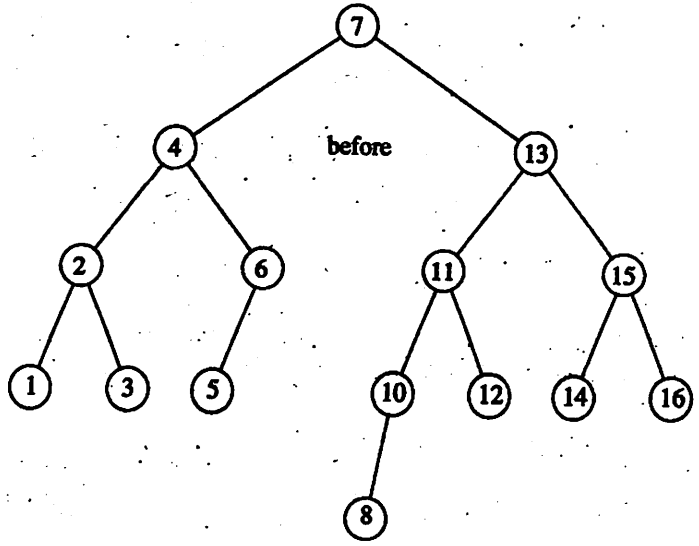
If 13 is now inserted, there is an imbalance at the root. Since 13 is not between 4 and 7 we know that the single rotation will work.



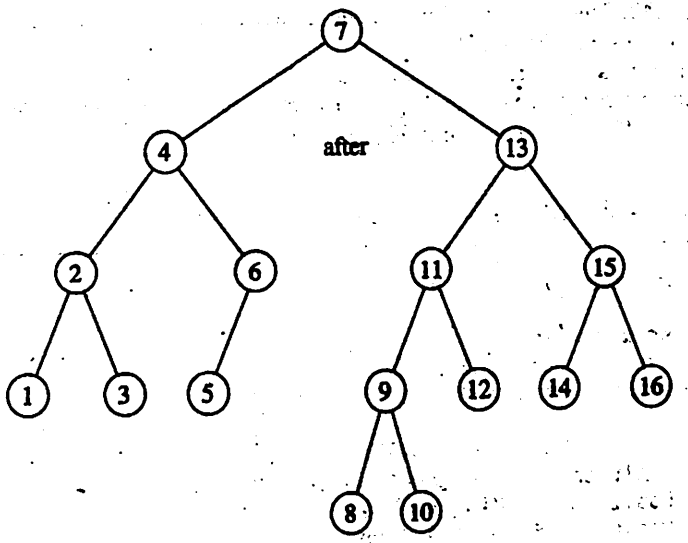
Insertion of 12 will also require a single rotation:



To insert 11, a single rotation needs to be performed, and the same is true for the subsequent insertion of 10. We insert 8 without a rotation creating an almost perfectly balanced tree:



Finally, we will insert 9 to show the symmetric case of the double rotation. Notice that 9 causes the node containing 10 to become unbalanced. Since 9 is between 10 and 8 (which is 10's child on the path to 9), a double rotation needs to be performed, yielding the following tree:



Let us summarize what happens. The programming details are fairly straightforward except that there are several cases. To insert a new node with key X into an AVL tree T, we recursively insert X into the appropriate subtree of T (let us call this T_{LR}). If the height of T_{LR} does not change, then we are done. Otherwise, if a height imbalance appears in T, we do the appropriate single or double rotation depending on X and the keys in T and T_{LR}, update the heights (making the connection from the rest of the tree above), and are done. Since one rotation always suffices, a carefully coded nonrecursive version generally turns out to be significantly faster than the recursive version. However, nonrecursive versions are quite difficult to code correctly, so many programmers implement AVL trees recursively.

Another efficiency issue concerns storage of the height information. Since all that is really required is the difference in height, which is guaranteed to be small, we could get by with two bits (to represent +1, 0, −1) if we really try. Doing so will avoid repetitive calculation of balance factors but results in some loss of clarity. The resulting code is somewhat more complicated than if the height were stored at each node. If a recursive routine is written, then speed is probably not the main consideration. In this case, the slight speed advantage obtained by storing balance factors hardly seems worth the loss of clarity and relative simplicity. Furthermore, since most machines will align this to at least an 8-bit boundary anyway, there is not likely to be any difference in the amount of space used. Eight bits will allow us to store absolute heights of up to 255. Since the tree is balanced, it is inconceivable that this would be insufficient (see the exercises).

With all this, we are ready to write the AVL routines. We will do only a partial job and leave the rest as an exercise. First, we need the declarations. These are given in Figure 4.37. We also need a quick function to return the height of a node. This function is necessary to handle the annoying case of a *NULL* pointer. This is shown

in Figure 4.38. The basic insertion routine is easy to write, since it consists mostly of function calls (see Fig. 4.39).

For the trees in Figure 4.40, *SingleRotateWithLeft* converts the tree on the left to the tree on the right, returning a pointer to the new root. *SingleRotateWithRight* is symmetric. The code is shown in Figure 4.41.

Figure 4.37 Node declaration for AVL trees

```
#ifndef _AvlTree_H

struct AvlNode;
typedef struct AvlNode *Position;
typedef struct AvlNode *AvlTree;

AvlTree MakeEmpty( AvlTree T );
Position Find( ElementType X, AvlTree T );
Position FindMin( AvlTree T );
Position FindMax( AvlTree T );
AvlTree Insert( ElementType X, AvlTree T );
AvlTree Delete( ElementType X, AvlTree T );
ElementType Retrieve( Position P );

#endif  /* _AvlTree_H */


/* Place in the implementation file */
struct AvlNode
{
    ElementType Element;
    AvlTree  Left;
    AvlTree  Right;
    int      Height;
};
```

Figure 4.38 Function to compute height of an AVL node

```
static int
Height( Position P )
{
    if( P == NULL )
        return -1;
    else
        return P->Height;
}
```



```
AvlTree
Insert( ElementType X, AvlTree T )
{
    if( T == NULL )
    {
        /* Create and return a one-node tree */
        T = malloc( sizeof( struct AvlNode ) );
        if( T == NULL )
            FatalError( "Out of space!!!" );
        else
        {
            T->Element = X; T->Height = 0;
            T->Left = T->Right = NULL;
        }
    }
    else
    if( X < T->Element )
    {
        T->Left = Insert( X, T->Left );
        if( Height( T->Left ) - Height( T->Right ) == 2 )
            if( X < T->Left->Element )
                T = SingleRotateWithLeft( T );
            else
                T = DoubleRotateWithLeft( T );
    }
    else
    if( X > T->Element )
    {
        T->Right = Insert( X, T->Right );
        if( Height( T->Right ) - Height( T->Left ) == 2 )
            if( X > T->Right->Element )
                T = SingleRotateWithRight( T );
            else
                T = DoubleRotateWithRight( T );
    }
    /* Else X is in the tree already; we'll do nothing */

    T->Height = Max( Height( T->Left ), Height( T->Right ) ) +
    return T;
}
```

Figure 4.39 Insertion into an AVL tree

The last function we will write will perform the double rotation pictured in Figure 4.42, for which the code is shown in Figure 4.43.

Deletion in AVL trees is somewhat more complicated than insertion. Lazy deletion is probably the best strategy if deletions are relatively infrequent.

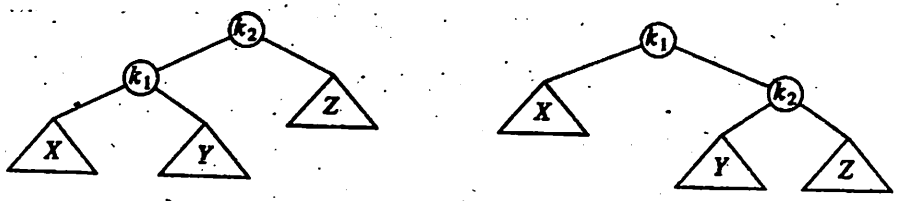


Figure 4.40 Single Rotation

```
/* This function can be called only if K2 has a left child */
/* Perform a rotate between a node (K2) and its left child */
/* Update heights, then return new root */

static Position
SingleRotateWithLeft( Position K2 )
{
    Position K1;

    K1 = K2->Left;
    K2->Left = K1->Right;
    K1->Right = K2;

    K2->Height = Max( Height( K2->Left ),
                      Height( K2->Right ) ) + 1;
    K1->Height = Max( Height( K1->Left ), K2->Height ) + 1;

    return K1;  /* New root */
}
```

Figure 4.41 Routine to perform single rotation

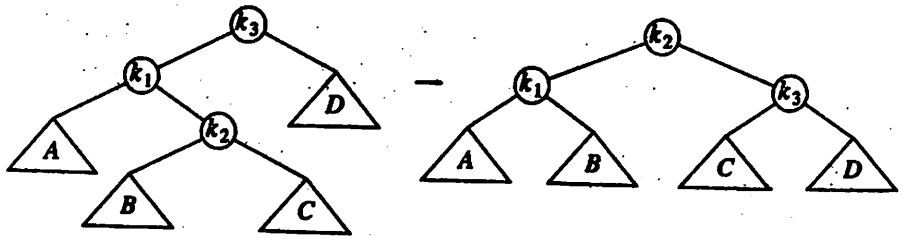


Figure 4.42 Double rotation


```
/* This function can be called only if K3 has a left */
/* child and K3's left child has a right child */
/* Do the left-right double rotation */
/* Update heights, then return new root */

static Position
DoubleRotateWithLeft( Position K3 )
{
    /* Rotate between K1 and K2 */
    K3->Left = SingleRotateWithRight( K3->Left );

    /* Rotate between K3 and K2 */
    return SingleRotateWithLeft( K3 );
}
```

Figure 4.43 Routine to perform double rotation

4.5. Splay Trees

We now describe a relatively simple data structure, known as a *splay tree*, that guarantees that any M consecutive tree operations starting from an empty tree take at most $O(M \log N)$ time. Although this guarantee does not preclude the possibility that any *single* operation might take $O(N)$ time, and thus the bound is not as strong as an $O(\log N)$ worst-case bound per operation, the net effect is the same: There are no bad input sequences. Generally, when a sequence of M operations has total worst-case running time of $O(MF(N))$, we say that the *amortized* running time is $O(F(N))$. Thus, a splay tree has an $O(\log N)$ amortized cost per operation. Over a long sequence of operations, some may take more, some less.

Splay trees are based on the fact that the $O(N)$ worst-case time per operation for binary search trees is not bad, as long at it occurs relatively infrequently. Any one access, even if it takes $O(N)$, is still likely to be extremely fast. The problem with binary search trees is that it is possible, and not uncommon, for a whole sequence of bad accesses to take place. The cumulative running time then becomes noticeable. A search tree data structure with $O(N)$ worst-case time, but a *guarantee* of at most $O(M \log N)$ for any M consecutive operations, is certainly satisfactory, because there are no bad sequences.

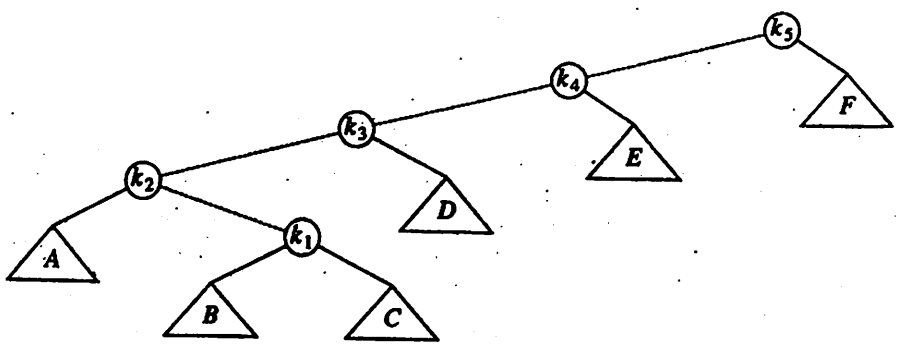
If any particular operation is allowed to have an $O(N)$ worst-case time bound, and we still want an $O(\log N)$ amortized time bound, then it is clear that whenever a node is accessed, it must be moved. Otherwise, once we find a deep node, we could keep performing *Finds* on it. If the node does not change location, and each access costs $O(N)$, then a sequence of M accesses will cost $O(M \cdot N)$.

The basic idea of the splay tree is that after a node is accessed, it is pushed to the root by a series of AVL tree rotations. Notice that if a node is deep, there are

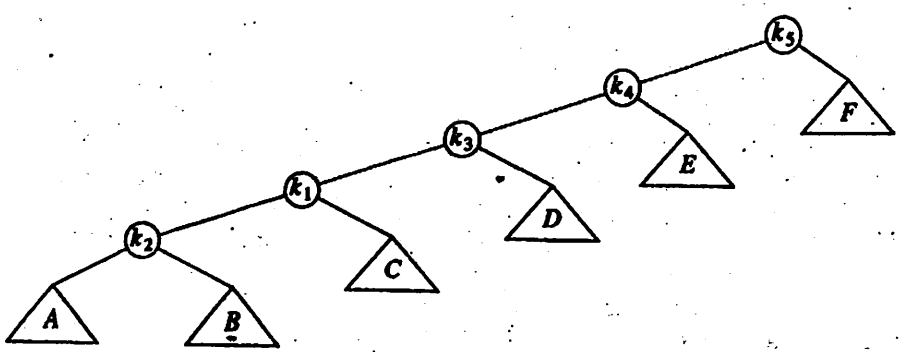
many nodes on the path that are also relatively deep, and by restructuring we can make future accesses cheaper on all these nodes. Thus, if the node is unduly deep, then we want this restructuring to have the side effect of balancing the tree (to some extent). Besides giving a good time bound in theory, this method is likely to have practical utility, because in many applications when a node is accessed, it is likely to be accessed again in the near future. Studies have shown that this happens much more often than one would expect. Splay trees also do not require the maintenance of height or balance information, thus saving space and simplifying the code to some extent (especially when careful implementations are written).

4.5.1. *A Simple Idea (That Does Not Work)*

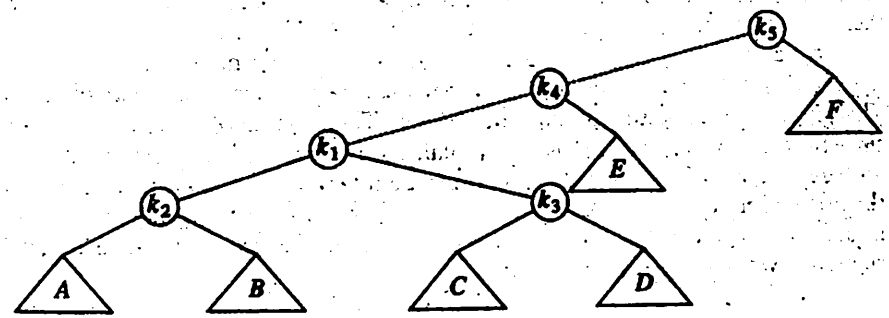
One way of performing the restructuring described above is to perform single rotations, bottom up. This means that we rotate every node on the access path with its parent. As an example, consider what happens after an access (a *Find*) on k_1 in the following tree.



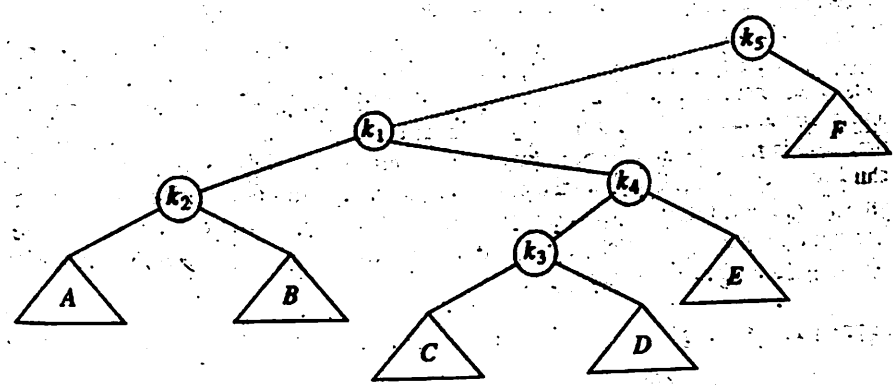
The access path is dashed. First, we would perform a single rotation between k_1 and its parent, obtaining the following tree.

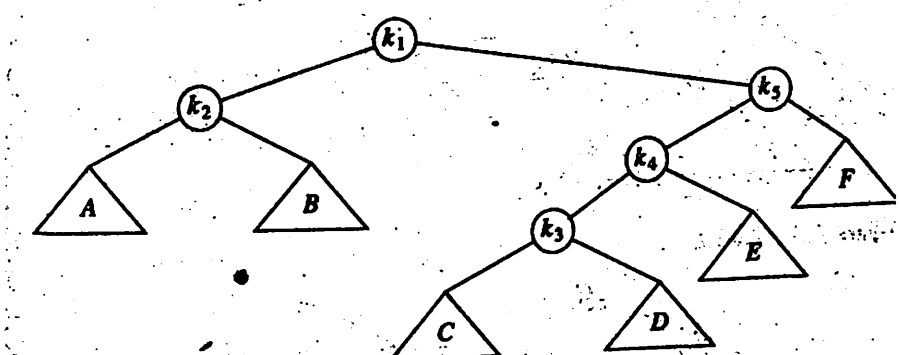


Then, we rotate between k_1 and k_3, obtaining the next tree.



Then two more rotations are performed until we reach the root.





These rotations have the effect of pushing k_1 all the way to the root, so that future accesses on k_1 are easy (for a while). Unfortunately, it has pushed another node (k_3) almost as deep as k_1 used to be. An access on that node will then push

another node deep, and so on. Although this strategy makes future accesses of k_1 cheaper, it has not significantly improved the situation for the other nodes on the (original) access path. It turns out that it is possible to prove that using this strategy, there is a sequence of M operations requiring $\Omega(M \cdot N)$ time, so this idea is not quite good enough. The simplest way to show this is to consider the tree formed by inserting keys 1, 2, 3, . . . , N into an initially empty tree (work this example out). This gives a tree consisting of only left children. This is not necessarily bad, though, since the time to build this tree is $O(N)$ total. The bad part is that accessing the node with key 1 takes $N - 1$ units of time. After the rotations are complete, an access of the node with key 2 takes $N - 2$ units of time. The total for accessing all the keys in order is $\sum_{i=1}^{N-1} i = \Omega(N^2)$. After they are accessed, the tree reverts to its original state, and we can repeat the sequence.

4.5.2. *Splaying*

The splaying strategy is similar to the rotation idea above, except that we are a little more selective about how rotations are performed. We will still rotate bottom up along the access path. Let X be a (nonroot) node on the access path at which we are rotating. If the parent of X is the root of the tree, we merely rotate X and the root. This is the last rotation along the access path. Otherwise, X has both a parent (P) and a grandparent (G), and there are two cases, plus symmetries, to consider. The first case is the *zig-zag* case (see Fig. 4.44). Here X is a right child and P is a left child (or vice versa). If this is the case, we perform a double rotation, exactly like an AVL double rotation. Otherwise, we have a *zig-zig* case: X and P are either both left children or both right children. In that case, we transform the tree on the left of Figure 4.45 to the tree on the right.

Figure 4.44 Zig-zag

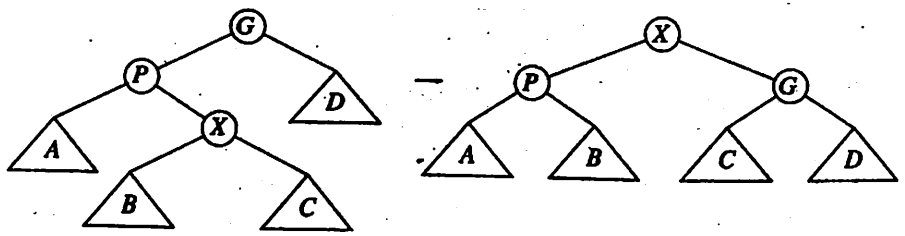
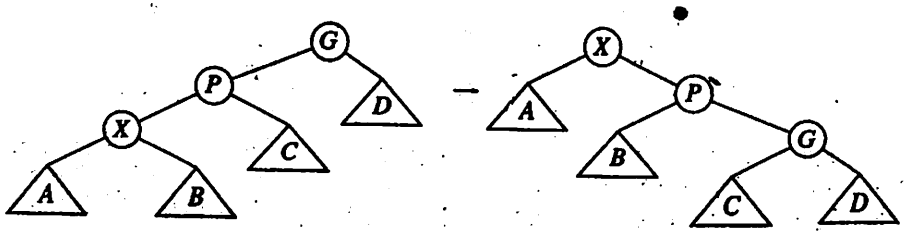
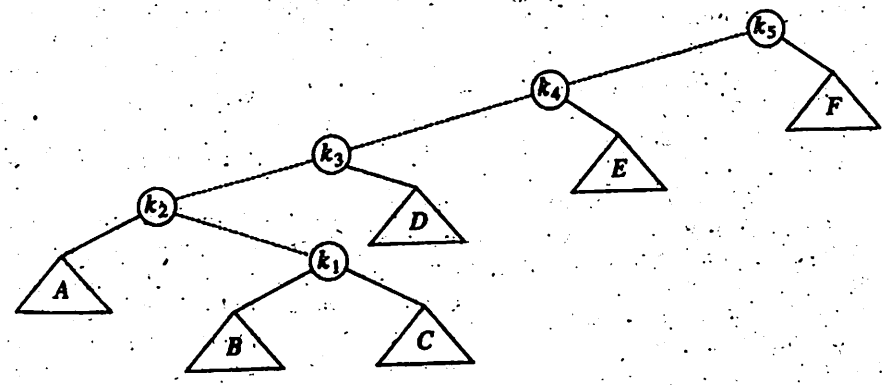


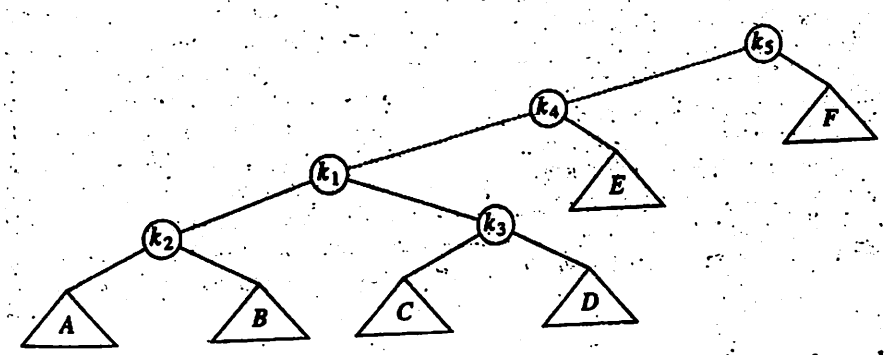
Figure 4.45 Zig-zig



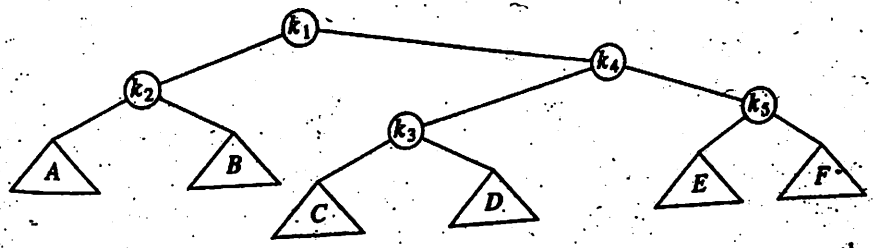
As an example, consider the tree from the last example, with a *Find* on k_1:



The first splay step is at k_1, and is clearly a *zig-zag*, so we perform a standard AVL double rotation using k_1, k_2, and k_3. The resulting tree follows.



The next splay step at k_1 is a *zig-zig*, so we do the *zig-zig* rotation with k_1, k_4, and k_5, obtaining the final tree.



Although it is hard to see from small examples, splaying not only moves the accessed node to the root, but also has the effect of roughly halving the depth of

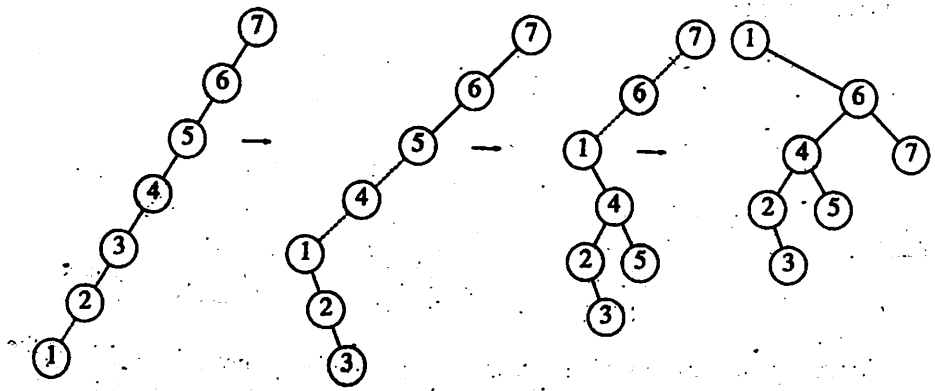
most nodes on the access path (some shallow nodes are pushed down at most two levels).

To see the difference that splaying makes over simple rotation, consider again the effect of inserting keys 1, 2, 3, ..., N into an initially empty tree. This takes a total of $O(N)$, as before, and yields the same tree as simple rotations. Figure 4.46 shows the result of splaying at the node with key 1. The difference is that after an access of the node with key 1, which takes $N - 1$ units, the access on the node with key 2 will only take about $N/2$ units instead of $N - 2$ units; there are no nodes quite as deep as before.

An access on the node with key 2 will bring nodes to within $N/4$ of the root, and this is repeated until the depth becomes roughly $\log N$ (an example with $N = 7$ is too small to see the effect well). Figures 4.47 to 4.55 show the result of accessing keys 1 through 9 in a 32-node tree that originally contains only left children. Thus we do not get the same bad behavior from splay trees that is prevalent in the simple rotation strategy. (Actually, this turns out to be a very good case. A rather complicated proof shows that for this example, the N accesses take a total of $O(N)$ time.)

These figures highlight the fundamental and crucial property of splay trees. When access paths are long, thus leading to a longer-than-normal search time, the rotations tend to be good for future operations. When accesses are cheap, the rotations are not as good and can be bad. The extreme case is the initial tree formed by the insertions. All the insertions were constant-time operations leading to a bad initial tree. At that point in time, we had a very bad tree, but we were running ahead of schedule and had the compensation of less total running time. Then a couple of really horrible accesses left a nearly balanced tree, but the cost was that we had to give back some of the time that had been saved. The main theorem, which we will prove in Chapter 11, is that we never fall behind a pace of $O(\log N)$ per operation: We are always on schedule, even though there are occasionally bad operations.

Figure 4.46 Result of splaying at node 1



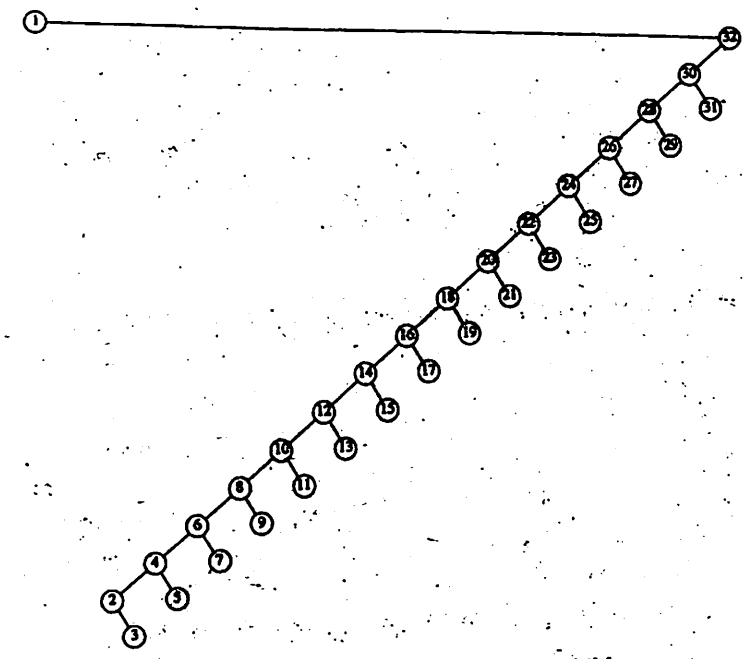


Figure 4.47 Result of splaying at node 1 a tree of all left children

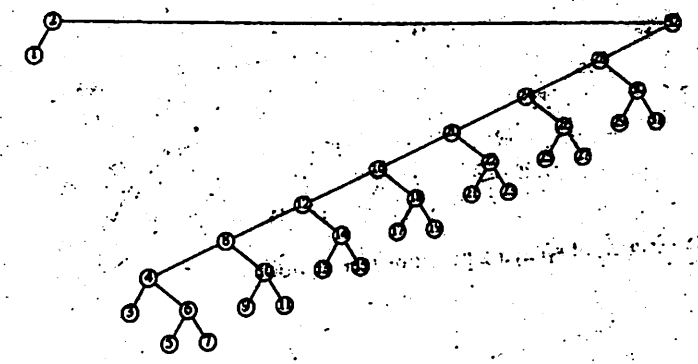


Figure 4.48 Result of splaying previous tree at node 2

We can perform deletion by accessing the node to be deleted. This puts the node at the root. If it is deleted, we get two subtrees T_L and T_R (left and right). If we find the largest element in T_L (which is easy), then this element is rotated to the root of T_L, and T_L will now have a root with no right child. We can finish the deletion by making T_R the right child.

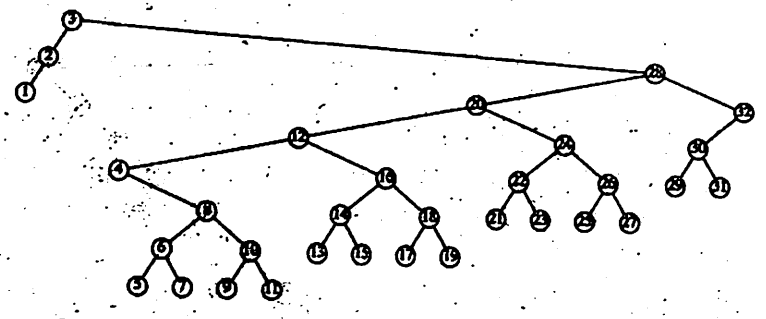


Figure 4.49 Result of splaying previous tree at node 3

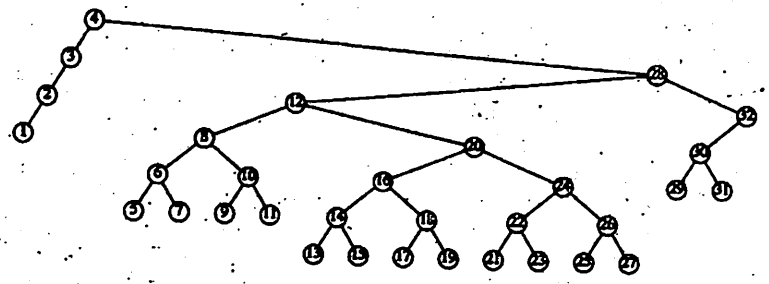


Figure 4.50 Result of splaying previous tree at node 4

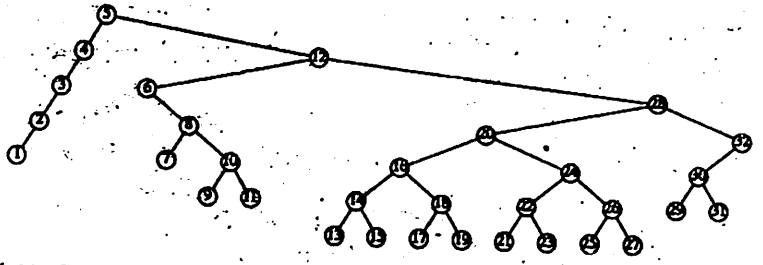


Figure 4.51 Result of splaying previous tree at node 5

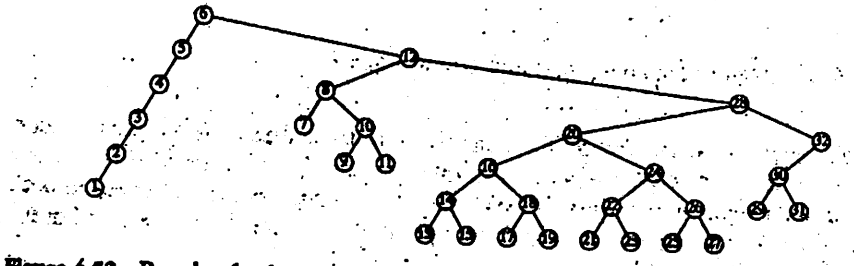


Figure 4.52 Result of splaying previous tree at node 6

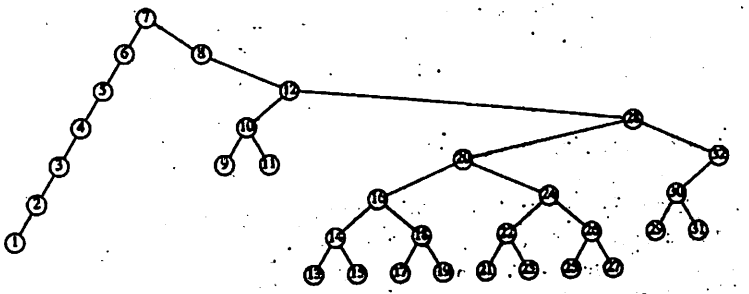


Figure 4.53 Result of splaying previous tree at node 7

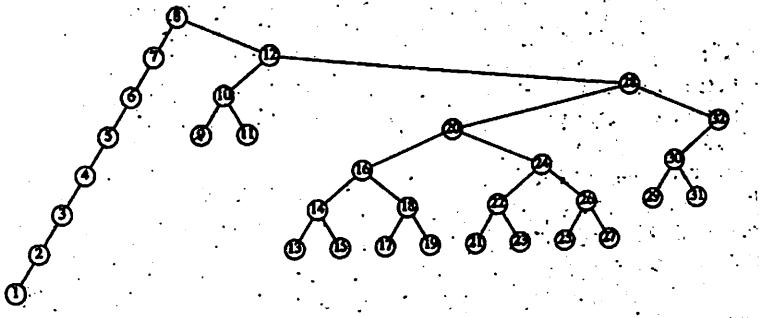


Figure 4.54 Result of splaying previous tree at node 8

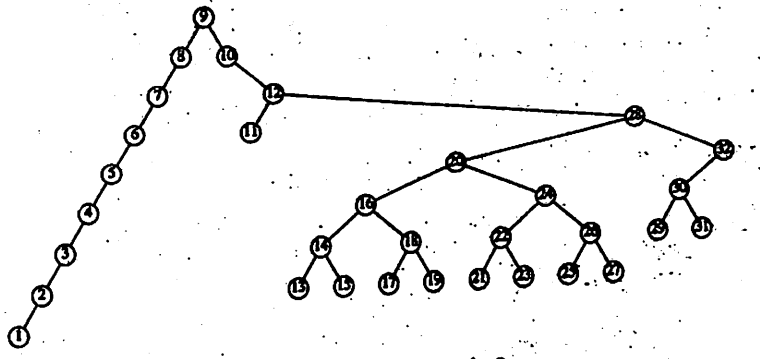


Figure 4.55 Result of splaying previous tree at node 9

The analysis of splay trees is difficult, because it must take into account the ever-changing structure of the tree. On the other hand, splay trees are much simpler to program than AVL trees, since there are fewer cases to consider and no balance information to maintain. Some empirical evidence suggests that this translates into faster code in practice, although the case for this is far from complete. Finally, we

point out that there are several variations of splay trees that can perform even better in practice. One variation is completely coded in Chapter 12.

4.6. Tree Traversals (Revisited)

Because of the ordering information in a binary search tree, it is simple to list all the keys in sorted order. The recursive procedure in Figure 4.56 does this.

Convince yourself that this procedure works. As we have seen before, this kind of routine when applied to trees is known as an *inorder* traversal (which makes sense, since it lists the keys in order). The general strategy of an inorder traversal is to process the left subtree first, then perform processing at the current node, and finally process the right subtree. The interesting part about this algorithm, aside from its simplicity, is that the total running time is $O(N)$. This is because there is constant work being performed at every node in the tree. Each node is visited once, and the work performed at each node is testing against *NULL*, setting up two procedure calls, and doing a *PrintElement*. Since there is constant work per node and N nodes, the running time is $O(N)$.

Sometimes we need to process both subtrees first before we can process a node. For instance, to compute the height of a node, we need to know the height of the subtrees first. The code in Figure 4.57 computes this. Since it is always a good idea to check the special cases—and crucial when recursion is involved—notice that the routine will declare the height of a leaf to be zero, which is correct. This general order of traversal, which we have also seen before, is known as a *postorder* traversal. Again, the total running time is $O(N)$, because constant work is performed at each node.

The third popular traversal scheme that we have seen is *preorder* traversal. Here, the node is processed before the children. This could be useful, for example, if you wanted to label each node with its depth.

The common idea in all of these routines is that you handle the *NULL* case first, and then the rest. Notice the lack of extraneous variables. These routines pass only

Figure 4.56 Routine to print a binary search tree in order

```
void
PrintTree( SearchTree T )
{
    if( T != NULL )
    {
        PrintTree( T->Left );
        PrintElement( T->Element );
        PrintTree( T->Right );
    }
}
```



```
int
Height( Tree T )
{
    if( T == NULL )
        return -1;
    else
        return 1 + Max( Height( T->Left ),
                        Height( T->Right ) );
}
```

Figure 4.57 Routine to compute the height of a tree using a postorder traversal

the tree, and do not declare or pass any extra variables. The more compact the code, the less likely that a silly bug will turn up. A fourth, less often used, traversal (which we have not seen yet) is *level-order* traversal. In a level-order traversal, all nodes at depth D are processed before any node at depth $D + 1$. Level-order traversal differs from the other traversals in that it is not done recursively; a queue is used, instead of the implied stack of recursion.

4.7. B-Trees

Although all of the search trees we have seen so far are binary, there is a popular search tree that is not binary. This tree is known as a *B-tree*.

A B-tree of order M is a tree with the following structural properties:

- The root is either a leaf or has between 2 and M children.
- All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
- All leaves are at the same depth.

All data are stored at the leaves. Contained in each interior node are pointers $P_1, P_2, \ldots, P_M$ to the children, and values $k_1, k_2, \ldots, k_{M-1}$, representing the smallest key found in the subtrees $P_2, P_3, \ldots, P_M$, respectively. Of course, some of these pointers might be *NULL*, and the corresponding k_i would then be undefined. For every node, all the keys in subtree P_1 are smaller than the keys in subtree P_2, and so on. The leaves contain all the actual data, which are either the keys themselves or pointers to records containing the keys. We will assume the former to keep our examples simple. There are various definitions of B-trees that change this structure in mostly minor ways, but this definition is one of the popular forms. (A popular alternative structure allows the actual data to be stored in both leaves and internal nodes, as is done in binary search trees.) We will also insist (for now) that the number of keys in a (nonroot) leaf is also between $\lceil M/2 \rceil$ and M.

The tree in Figure 4.58 is an example of a B-tree of order 4.

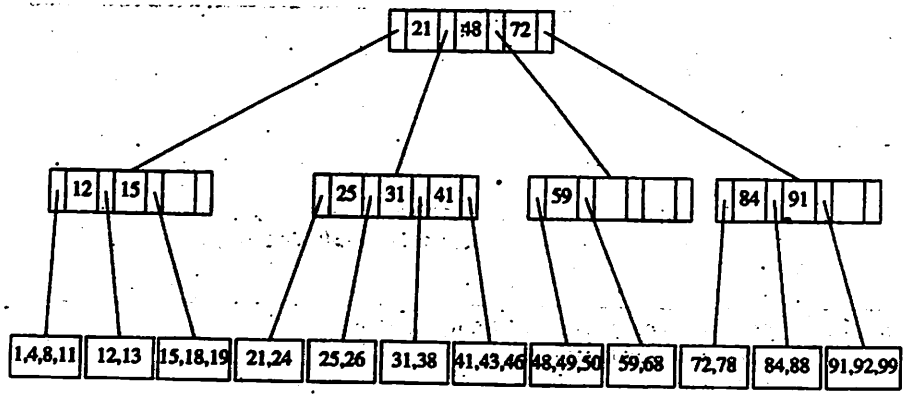
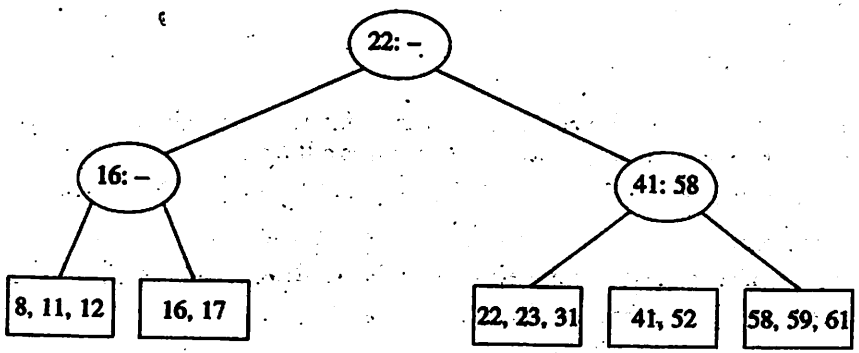


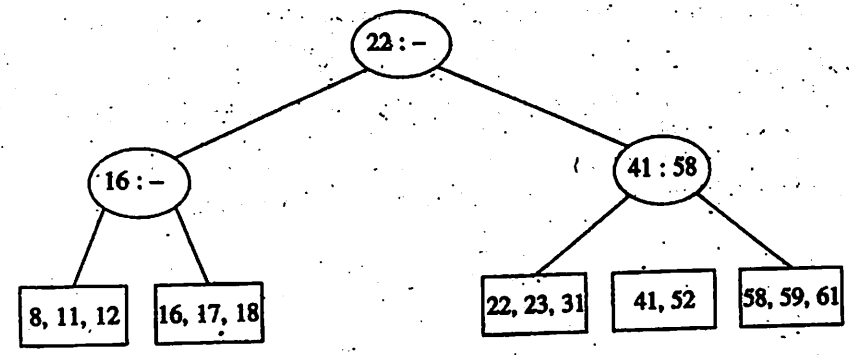
Figure 4.58 B-tree of order 4

A B-tree of order 4 is more popularly known as a 2–3–4 tree, and a B-tree of order 3 is known as a 2–3 tree. We will describe the operation of B-trees by using the special case of 2–3 trees. Our starting point is the 2–3 tree that follows.

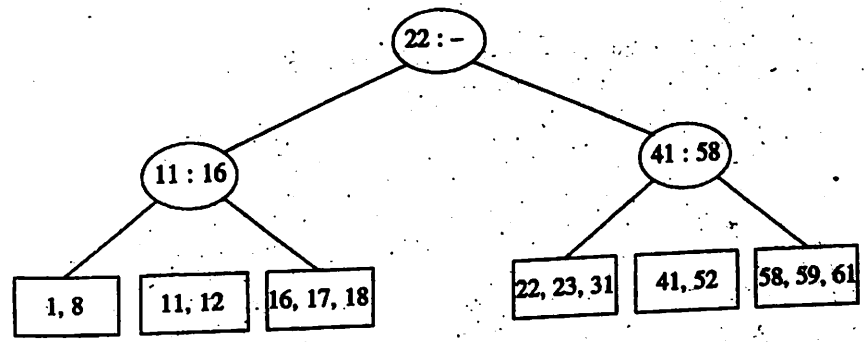


We have drawn interior nodes (nonleaves) in ellipses, which contain the two pieces of data for each node. A dash line as a second piece of information in an interior node indicates that the node has only two children. Leaves are drawn in boxes, which contain the keys. The keys in the leaves are ordered. To perform a *Find*, we start at the root and branch in one of (at most) three directions, depending on the relation of the key we are looking for to the two (possibly one) values stored at the node.

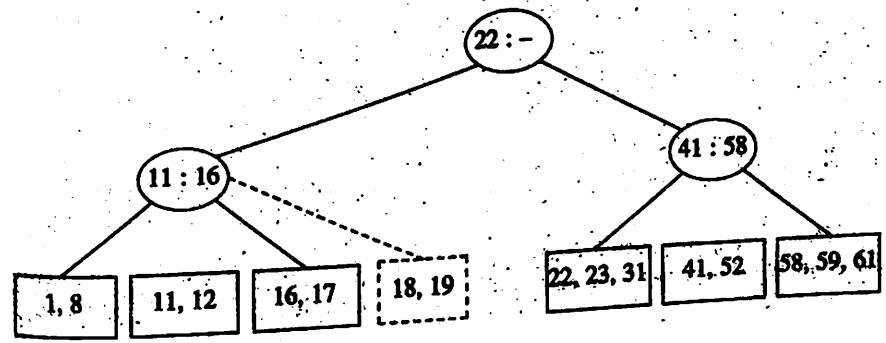
To perform an *Insert* on a previously unseen key, X, we follow the path as though we were performing a *Find*. When we get to a leaf node, we have found the correct place to put X. Thus, to insert a node with key 18, we can just add it to a leaf without causing any violations of the 2–3 tree properties. The result is shown in the following figure.



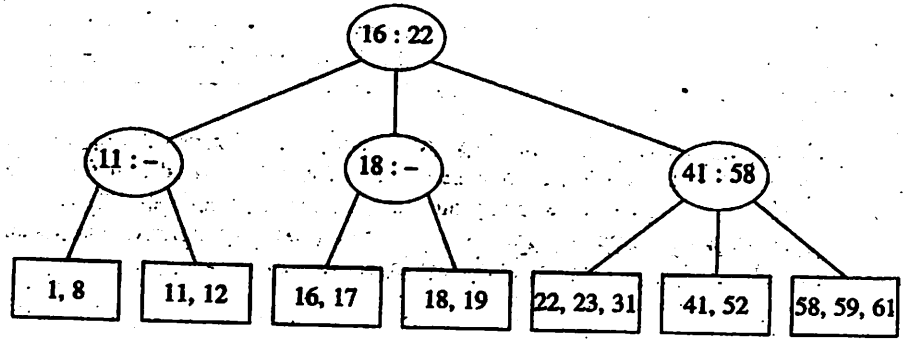
Unfortunately, since a leaf can hold only two or three keys, this might not always be possible. If we now try to insert 1 into the tree, we find that the node where it belongs is already full. Placing our new key into this node would give it a fourth element, which is not allowed. This can be solved by making two nodes of two keys each and adjusting the information in the parent.



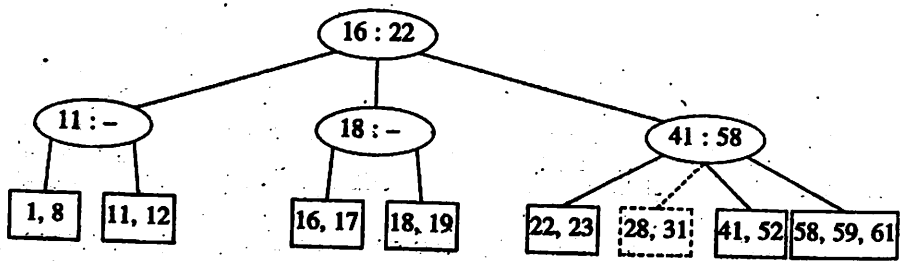
Unfortunately, this idea does not always work, as can be seen by an attempt to insert 19 into the current tree. If we make two nodes of two keys each, we obtain the following tree.

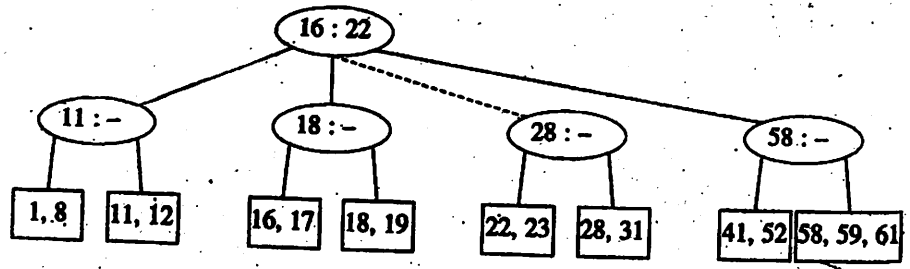


This tree has an internal node with four children, but we only allow three per node. The solution is simple. We merely split this node into two nodes with two children. Of course, this node might be one of three children itself, and thus splitting it would create a problem for its parent (which would now have four children), but we can keep on splitting nodes on the way up to the root until we either get to the root or find a node with only two children. In our case, we can get by with splitting only the first internal node we see, obtaining the following tree.



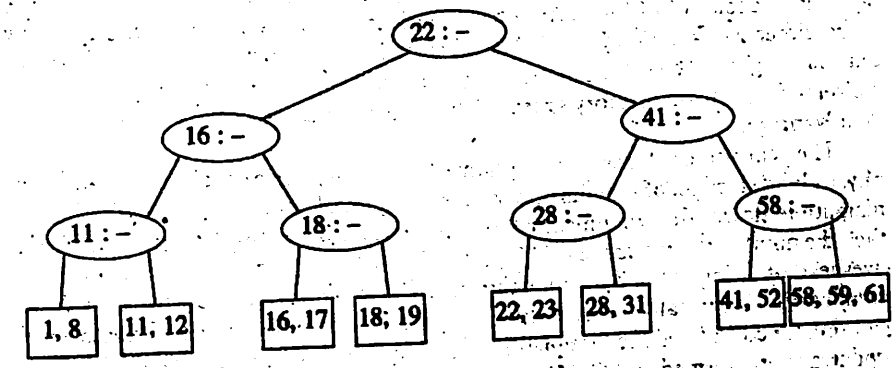
If we now insert an element with key 28, we create a leaf with four children, which is split into two leaves of two children:





This creates an internal node with four children, which is then split into two children. What we have done here is split the root into two nodes. When we do this,

we have a special case, which we finish by creating a new root. This is how (the only way) a 2–3 tree gains height.



Notice also that when a key is inserted, the only changes to internal nodes occur on the access path. These changes can be made in time proportional to the length of this path; but be forewarned that there are quite a few cases to handle, and it is easy to do this wrong.

There are other ways to handle the case where a node becomes overloaded with children, but the method we have described is probably the simplest. When attempting to add a fourth key to a leaf, instead of splitting the node into two we can first attempt to find a sibling with only two keys. For instance, to insert 70 into the tree above, we could move 58 to the leaf containing 41 and 52, place 70 with 59 and 61, and adjust the entries in the internal nodes. This strategy can also be applied to internal nodes and tends to keep more nodes full. The cost of this is slightly more complicated routines, but less space tends to be wasted.

We can perform deletion by finding the key to be deleted and removing it. If this key was one of only two keys in a node, then its removal leaves only one key. We can fix this by combining this node with a sibling. If the sibling has three keys, we can steal one and have both nodes with two keys. If the sibling has only two keys, we combine the two nodes into a single node with three keys. The parent of this node now loses a child, so we might have to percolate this strategy all the way to the top. If the root loses its second child, then the root is also deleted and the tree becomes one level shallower. As we combine nodes, we must remember to update the information kept at the internal nodes.

With general B-trees of order M, when a key is inserted, the only difficulty arises when the node that is to accept the key already has M keys. This key gives the node $M + 1$ keys, which we can split into two nodes with $\lceil (M + 1)/2 \rceil$ and $\lfloor (M + 1)/2 \rfloor$ keys, respectively. As this gives the parent an extra node, we have to check whether this node can be accepted by the parent and split the parent if it already has M children. We repeat this until we find a parent with less than M children. If we split the root, we create a new root with two children.

The depth of a B-tree is at most $\lceil \log_{\lceil M/2 \rceil} N \rceil$. At each node on the path, we perform $O(\log M)$ work to determine which branch to take (using a binary search),

but an *Insert* or *Delete* could require $O(M)$ work to fix up all the information at the node. The worst-case running time for each of the *Insert* and *Delete* operations is thus $O(M \log_M N) = O((M/\log M)\log N)$, but a *Find* takes only $O(\log N)$. The best (legal) choice of M for running time considerations has been shown empirically to be either $M = 3$ or $M = 4$; this agrees with the bounds above, which show that as M gets larger, the insertion and deletion times increase. If we are only concerned with main memory speed, higher order B-trees, such as 5–9 trees, are not an advantage.

The real use of B-trees lies in database systems, where the tree is kept on a physical disk instead of main memory. Accessing a disk is typically several orders of magnitude slower than any main memory operation. If we use a B-tree of order M, then the number of *disk accesses* is $O(\log_M N)$. Although each disk access carries the overhead of $O(\log M)$ to determine the direction to branch, the time to perform this computation is typically much smaller than the time to read a block of memory and can thus be considered inconsequential (as long as M is chosen reasonably). Even if updates are performed and $O(M)$ computing time is required at each node, these too are generally not significant. The value of M is then chosen to be the largest value that still allows an interior node to fit into one disk block, and is typically in the range $32 \leq M \leq 256$. The maximum number of elements that are stored in a leaf is chosen so that if the leaf is full, it fits in one block. This means that a record can always be found in very few disk accesses, since a typical B-tree will have a depth of only 2 or 3, and the root (and possibly the first level) can be kept in main memory.

Analysis suggests that a B-tree will be $\ln 2 = 69$ percent full. Better space utilization can be obtained if, instead of always splitting a node when the tree obtains its $(M + 1)$th entry, the routine searches for a sibling that can take the extra child. The details can be found in the references.

Summary

We have seen uses of trees in operating systems, compiler design, and searching. Expression trees are a small example of a more general structure known as a *parse tree*, which is a central data structure in compiler design. Parse trees are not binary, but are relatively simple extensions of expression trees (although the algorithms to build them are not quite so simple).

Search trees are of great importance in algorithm design. They support almost all the useful operations, and the logarithmic average cost is very small. Nonrecursive implementations of search trees are somewhat faster, but the recursive versions are sleeker, more elegant, and easier to understand and debug. The problem with search trees is that their performance depends heavily on the input being random. If this is not the case, the running time increases significantly, to the point where search trees become expensive linked lists.

We saw several ways to deal with this problem. AVL trees work by insisting that all nodes' left and right subtrees differ in heights by at most one. This ensures that the tree cannot get too deep. The operations that do not change the tree, as insertion does, can all use the standard binary search tree code. Operations that change the

tree must restore the tree. This can be somewhat complicated, especially in the case of deletion. We showed how to restore the tree after insertions in $O(\log N)$ time.

We also examined the splay tree. Nodes in splay trees can get arbitrarily deep, but after every access the tree is adjusted in a somewhat mysterious manner. The net effect is that any sequence of M operations takes $O(M \log N)$ time, which is the same as a balanced tree would take.

B-trees are balanced M-way (as opposed to 2-way or binary) trees, which are well suited for disks; a special case is the 2–3 tree, which is another common method of implementing balanced search trees.

In practice, the running time of all the balanced tree schemes is worse (by a constant factor) than the simple binary search tree, but this is generally acceptable in view of the protection being given against easily obtained worst-case input. Chapter 12 discusses some additional search tree data structures and provides detailed implementations.

A final note: By inserting elements into a search tree and then performing an inorder traversal, we obtain the elements in sorted order. This gives an $O(N \log N)$ algorithm to sort, which is a worst-case bound if any sophisticated search tree is used. We shall see better ways in Chapter 7, but none that have a lower time bound.

Exercises

Questions 4.1 to 4.3 refer to the tree in Figure 4.59.

4.1 For the tree in Figure 4.59:
   a. Which node is the root?
   b. Which nodes are leaves?

4.2 For each node in the tree of Figure 4.59:
   a. Name the parent node.
   b. List the children.
   c. List the siblings.
   d. Compute the depth.
   e. Compute the height.

4.3 What is the depth of the tree in Figure 4.59?

4.4 Show that in a binary tree of N nodes, there are $N + 1$ *NULL* pointers representing children.

4.5 Show that the maximum number of nodes in a binary tree of height H is $2^{H+1} - 1$.

4.6 A *full node* is a node with two children. Prove that the number of full nodes plus one is equal to the number of leaves in a nonempty binary tree.

4.7 Suppose a binary tree has leaves $l_1, l_2, \ldots, l_M$ at depths $d_1, d_2, \ldots, d_M$, respectively. Prove that $\sum_{i=1}^{M} 2^{-d_i} \leq 1$ and determine when the equality is true.

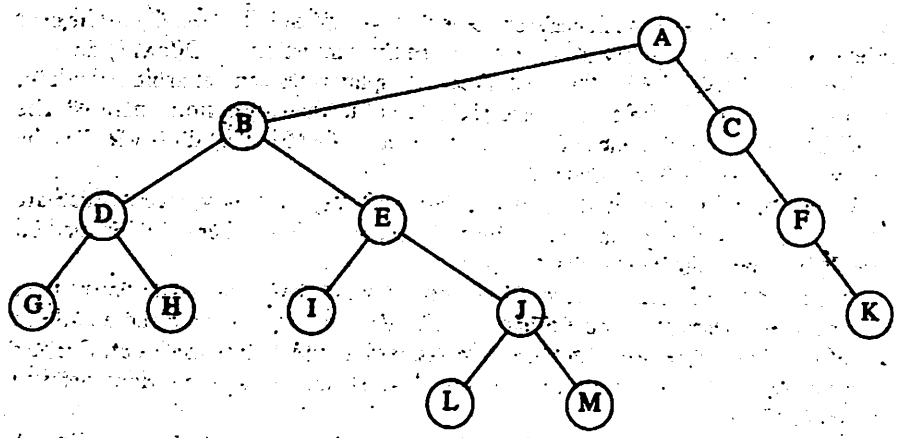


Figure 4.59

4.8 Give the prefix, infix, and postfix expressions corresponding to the tree in Figure 4.60.

4.9 a. Show the result of inserting 3, 1, 4, 6, 9, 2, 5, 7 into an initially empty binary search tree.

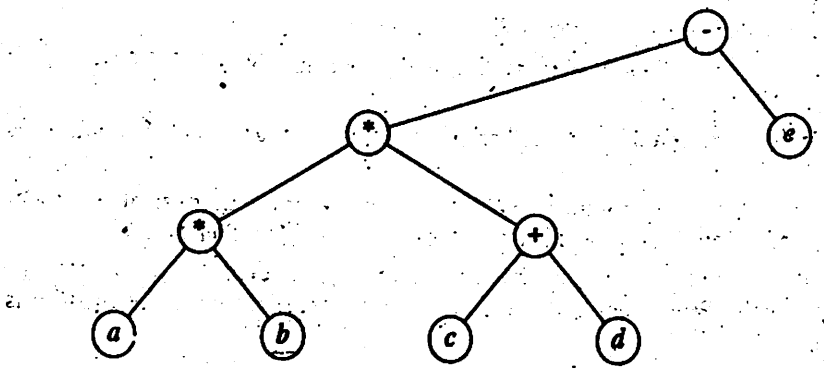
b. Show the result of deleting the root.

4.10 Write routines to implement the basic binary search tree operations.

4.11 Binary search trees can be implemented with cursors, using a strategy similar to a cursor linked list implementation. Write the basic binary search tree routines using a cursor implementation.

4.12 Suppose you want to perform an experiment to verify the problems that can be caused by random *Insert/Delete* pairs. Here is a strategy that is not perfectly random, but close enough. You build a tree with N elements by inserting N

Figure 4.60 Tree for Exercise 4.8



elements chosen at random from the range 1 to $M = \alpha N$. You then perform N^2 pairs of insertions followed by deletions. Assume the existence of a routine, *RandomInteger*(A, B), which returns a uniform random integer between A and B inclusive.

a. Explain how to generate a random integer between 1 and M that is not already in the tree (so a random insert can be performed). In terms of N and α, what is the running time of this operation?

b. Explain how to generate a random integer between 1 and M that is already in the tree (so a random delete can be performed). What is the running time of this operation?

c. What is a good choice of α? Why?

4.13 Write a program to evaluate empirically the following strategies for deleting nodes with two children:

a. Replace with the largest node, X, in T_L and recursively delete X.

b. Alternately replace with the largest node in T_L and the smallest node in T_R, and recursively delete the appropriate node.

c. Replace with either the largest node in T_L or the smallest node in T_R (recursively deleting the appropriate node), making the choice randomly.

Which strategy seems to give the most balance? Which takes the least CPU time to process the entire sequence?

4.14 ** Prove that the depth of a random binary search tree (depth of the deepest node) is $O(\log N)$, on average.

4.15 *a. Give a precise expression for the minimum number of nodes in an AVL tree of height H.

b. What is the minimum number of nodes in an AVL tree of height 15?

4.16 Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree.

4.17 * Keys $1, 2, \ldots, 2^k - 1$ are inserted in order into an initially empty AVL tree. Prove that the resulting tree is perfectly balanced.

4.18 Write the remaining procedures to implement AVL single and double rotations.

4.19 Write a nonrecursive function to insert into an AVL tree.

4.20 * How can you implement (nonlazy) deletion in AVL trees?

4.21 a. How many bits are required per node to store the height of a node in an N-node AVL tree?

b. What is the smallest AVL tree that overflows an 8-bit height counter?

4.22 Write the functions to perform the double rotation without the inefficiency of doing two single rotations.

4.23 Show the result of accessing the keys 3, 9, 1, 5 in order in the splay tree in Figure 4.61.

4.24 Show the result of deleting the element with key 6 in the resulting splay tree for the previous exercise.

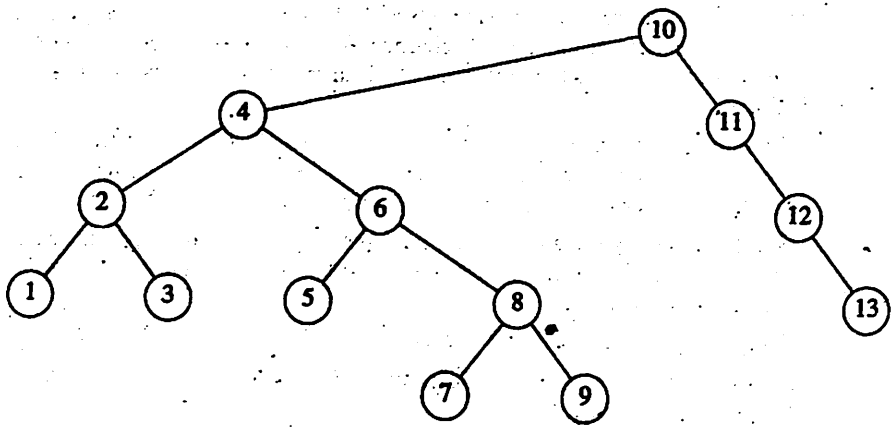


Figure 4.61

4.25 Nodes 1 through $N = 1024$ form a splay tree of left children.

a. What is the internal path length of the tree (exactly)?

*b. Calculate the internal path length after each of *Find*(1), *Find*(2), *Find*(3), *Find*(4), *Find*(5), *Find*(6).

*c. If the sequence of successive *Finds* is continued, when is the internal path length minimized?

4.26 a. Show that if all nodes in a splay tree are accessed in sequential order, the resulting tree consists of a chain of left children.

**b. Show that if all nodes in a splay tree are accessed in sequential order, then the total access time is $O(N)$, regardless of the initial tree.

4.27 Write a program to perform random operations on splay trees. Count the total number of rotations performed over the sequence. How does the running time compare to AVL trees and unbalanced binary search trees?

4.28 Write efficient functions that take only a pointer to the root of a binary tree, T, and compute:

a. The number of nodes in T.

b. The number of leaves in T.

c. The number of full nodes in T.

What is the running time of your routines?

4.29 Write a function to generate an N-node random binary search tree with distinct keys 1 through N. What is the running time of your routine?

4.30 Write a function to generate the AVL tree of height H with fewest nodes. What is the running time of your function?

4.31 Write a function to generate a perfectly balanced binary search tree of height H with keys 1 through $2^{H+1} - 1$. What is the running time of your function?

4.32 Write a function that takes as input a binary search tree, T, and two keys k_1 and k_2, which are ordered so that $k_1 \leq k_2$, and prints all elements X in the tree such that $k_1 \leq Key(X) \leq k_2$. Do not assume any information about the type of keys except that they can be ordered (consistently). Your program should run in $O(K + \log N)$ average time, where K is the number of keys printed. Bound the running time of your algorithm.

4.33 The larger binary trees in this chapter were generated automatically by a program. This was done by assigning an (x, y) coordinate to each tree node, drawing a circle around each coordinate (this is hard to see in some pictures), and connecting each node to its parent. Assume you have a binary search tree stored in memory (perhaps generated by one of the routines above) and that each node has two extra fields to store the coordinates.

a. The x coordinate can be computed by assigning the inorder traversal number. Write a routine to do this for each node in the tree.

b. The y coordinate can be computed by using the negative of the depth of the node. Write a routine to do this for each node in the tree.

c. In terms of some imaginary unit, what will the dimensions of the picture be? How can you adjust the units so that the tree is always roughly two-thirds as high as it is wide?

d. Prove that using this system no lines cross, and that for any node, X, all elements in X's left subtree appear to the left of X and all elements in X's right subtree appear to the right of X.

4.34 Write a general-purpose tree-drawing program that will convert a tree into the following graph-assembler instructions:

a. *Circle*(X, Y)

b. *DrawLine*(i, j)

The first instruction draws a circle at (X, Y), and the second instruction connects the ith circle to the jth circle (circles are numbered in the order drawn). You should either make this a program and define some sort of input language or make this a function that can be called from any program. What is the running time of your routine?

4.35 Write a routine to list out the nodes of a binary tree in *level-order*. List the root, then nodes at depth 1, followed by nodes at depth 2, and so on. You must do this in linear time. Prove your time bound.

4.36 a. Show the result of inserting the following keys into an initially empty 2–3 tree: 3, 1, 4, 5, 9, 2, 6, 8, 7, 0.

b. Show the result of deleting 0 and then 9 from the 2–3 tree created in part (a).

4.37 *a. Write a routine to perform insertion into a B-tree.

*b. Write a routine to perform deletion from a B-tree. When a key is deleted, is it necessary to update information in the internal nodes?

*c. Modify your insertion routine so that if an attempt is made to add into a node that already has M entries, a search is performed for a sibling with less than M children before the node is split.

4.38 A *B^*-tree* of order M is a B-tree in which each interior node has between $2M/3$ and M children. Describe a method to perform insertion into a B^*-tree.

4.39 Show how the tree in Figure 4.62 is represented using a child/sibling pointer implementation.

4.40 Write a procedure to traverse a tree stored with child/sibling links.

4.41 Two binary trees are similar if they are both empty or both nonempty and have similar left and right subtrees. Write a function to decide whether two binary trees are similar. What is the running time of your program?

4.42 Two trees, T_1 and T_2, are *isomorphic* if T_1 can be transformed into T_2 by swapping left and right children of (some of the) nodes in T_1. For instance, the two trees in Figure 4.63 are isomorphic because they are the same if the children of A, B, and G, but not the other nodes, are swapped.

a. Give a polynomial time algorithm to decide if two trees are isomorphic.

*b. What is the running time of your program (there is a linear solution)?

4.43 *a. Show that via AVL single rotations, any binary search tree T_1 can be transformed into another search tree T_2 (with the same keys).

*b. Give an algorithm to perform this transformation using $O(N \log N)$ rotations on average.

**c. Show that this transformation can be done with $O(N)$ rotations, worst-case.

Figure 4.62 Tree for Exercise 4.39

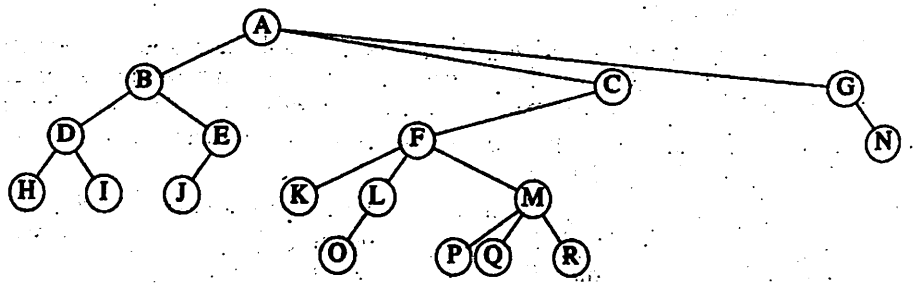
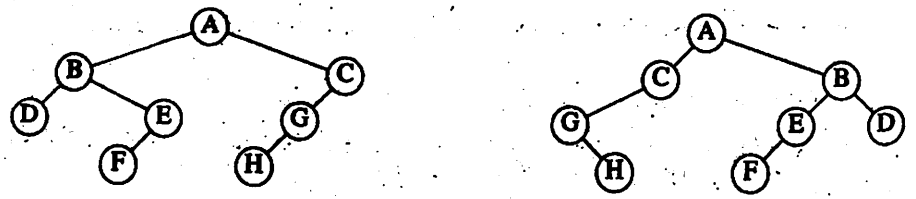


Figure 4.63 Two isomorphic trees



4.44 Suppose we want to add the operation *FindKth* to our repertoire. The operation *FindKth*(T, i) returns the element in tree T with ith smallest key. Assume all elements have distinct keys. Explain how to modify the binary search tree to support this operation in $O(\log N)$ average time, without sacrificing the time bounds of any other operation.

4.45 Since a binary search tree with N nodes has $N + 1$ *NULL* pointers, half the space allocated in a binary search tree for pointer information is wasted. Suppose that if a node has a *NULL* left child, we make its left child point to its inorder predecessor, and if a node has a *NULL* right child, we make its right child point to its inorder successor. This is known as a *threaded tree* and the extra pointers are called *threads*.

a. How can we distinguish threads from real children pointers?

b. Write routines to perform insertion and deletion into a tree threaded in the manner described above.

c. What is the advantage of using threaded trees?

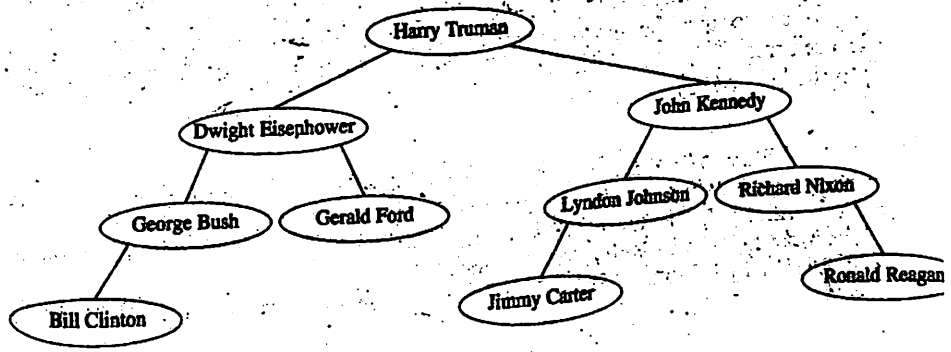
4.46 A binary search tree presupposes that searching is based on only one key per record. Suppose we would like to be able to perform searching based on either of two keys, Key_1 or Key_2.

a. One method is to build two separate binary search trees. How many extra pointers does this require?

b. An alternative method is a 2-d *tree*. A 2-d tree is similar to a binary search tree, except that branching at even levels is done with respect to Key_1, and branching at odd levels is done with Key_2. Figure 4.64 shows a 2-d tree, with the first and last names as keys, for post-WWII presidents. The presidents' names were inserted chronologically (Truman, Eisenhower, Kennedy, Johnson, Nixon, Ford, Carter, Reagan, Bush, Clinton). Write a routine to perform insertion into a 2-d tree.

c. Write an efficient procedure that prints all records in the tree that simultaneously satisfy the constraints $Low_1 \le Key_1 \le High_1$ and $Low_2 \le Key_2 \le High_2$.

Figure 4.64 A 2-d tree



d. Show how to extend the 2-d tree to handle more than two search keys. The resulting strategy is known as a k-d tree.

References

More information on binary search trees, and in particular the mathematical properties of trees, can be found in the two books by Knuth, [23] and [24].

Several papers deal with the lack of balance caused by biased deletion algorithms in binary search trees. Hibbard's paper [20] proposed the original deletion algorithm and established that one deletion preserves the randomness of the trees. A complete analysis has been performed only for trees with three nodes [21] and four nodes [5]. Eppinger's paper [15] provided early empirical evidence of nonrandomness, and the papers by Culberson and Munro, [11], [12] provided some analytical evidence (but not a complete proof for the general case of intermixed insertions and deletions).

AVL trees were proposed by Adelson-Velskii and Landis [1]. Simulation results for AVL trees, and variants in which the height imbalance is allowed to be at most k for various values of k, are presented in [22]. A deletion algorithm for AVL trees can be found in [24]. Analysis of the average search cost in AVL trees is incomplete, but some results are contained in [25].

[3] and [9] considered self-adjusting trees like the type in Section 4.5.1. Splay trees are described in [29].

B-trees first appeared in [6]. The implementation described in the original paper allows data to be stored in internal nodes as well as leaves. The data structure we have described is sometimes known as a B^+-tree. A survey of the different types of B-trees is presented in [10]. Empirical results of the various schemes are reported in [18]. Analysis of 2–3 trees and B-trees can be found in [4], [14], and [33].

Exercise 4.14 is deceptively difficult. A solution can be found in [16]. Exercise 4.26 is from [32]. Information on B^*-trees, described in Exercise 4.38, can be found in [13]. Exercise 4.42 is from [2]. A solution to Exercise 4.43 using $2N - 6$ rotations is given in [30]. Using threads, à la Exercise 4.45, was first proposed in [28]. k-d trees were first proposed in [7]. Their major drawback is that both deletion and balancing are difficult. [8] discusses k-d trees and other methods used for multidimensional searching; a short discussion is also provided in Chapter 12.

Other popular balanced search trees are red black trees [19] and weight-balanced trees [27]. More balanced tree schemes can be found in Chapter 12, as well as in the books [17], [26], and [31].

1. G. M. Adelson-Velskii and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet. Mat. Doklady*, 3 (1962), 1259–1263.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
3. B. Allen and J. I. Munro, "Self Organizing Search Trees," *Journal of the ACM*, 25 (1978), 526–535.

4. R. A. Baeza-Yates; "Expected Behaviour of B^+-trees under Random Insertions," *Acta Informatica*, 26 (1989), 439–471.
5. R. A. Baeza-Yates, "A Trivial Algorithm Whose Analysis Isn't: A Continuation," *BIT*, 29 (1989), 88–113.
6. R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices," *Acta Informatica*, 1 (1972), 173–189.
7. J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, 18 (1975), 509–517.
8. J. L. Bentley and J. H. Friedman, "Data Structures for Range Searching," *Computing Surveys*, 11 (1979), 397–409.
9. J. R. Bitner, "Heuristics that Dynamically Organize Data Structures," *SIAM Journal on Computing*, 8 (1979), 82–110.
10. D. Comer, "The Ubiquitous B-tree," *Computing Surveys*, 11 (1979), 121–137.
11. J. Culberson and J. I. Munro, "Explaining the Behavior of Binary Search Trees under Prolonged Updates: A Model and Simulations," *Computer Journal*, 32 (1989), 68–75.
12. J. Culberson and J. I. Munro, "Analysis of the Standard Deletion Algorithms in Exact Fit Domain Binary Search Trees," *Algorithmica*, 5 (1990) 295–311.
13. K. Culik, T. Ottman, and D. Wood, "Dense Multiway Trees," *ACM Transactions on Database Systems*, 6 (1981), 486–512.
14. B. Eisenbath, N. Ziviana, G. H. Gonnet, K. Melhorn, and D. Wood, "The Theory of Fringe Analysis and its Application to 2-3 Trees and B-trees," *Information and Control*, 55 (1982), 125–174.
15. J. L. Eppinger, "An Empirical Study of Insertion and Deletion in Binary Search Trees," *Communications of the ACM*, 26 (1983), 663–669.
16. P. Flajolet and A. Odlyzko, "The Average Height of Binary Trees and Other Simple Trees," *Journal of Computer and System Sciences*, 25 (1982), 171–213.
17. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2d ed., Addison-Wesley, Reading, Mass., 1991.
18. E. Gudes and S. Tsur, "Experiments with B-tree Reorganization," *Proceedings of ACM SIGMOD Symposium on Management of Data* (1980), 200–206.
19. L. J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proceedings of the Nineteenth Annual IEEE Symposium on Foundations of Computer Science* (1978), 8–21.
20. T. H. Hibbard, "Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting," *Journal of the ACM*, 9 (1962), 13–28.
21. A. T. Jonassen and D. E. Knuth, "A Trivial Algorithm Whose Analysis Isn't," *Journal of Computer and System Sciences*, 16 (1978), 301–322.
22. P. L. Karlton, S. H. Fuller, R. E. Scroggs, and E. B. Kaehler, "Performance of Height Balanced Trees," *Communications of the ACM*, 19 (1976), 23–28.
23. D. E. Knuth, *The Art of Computer Programming: Vol. 1: Fundamental Algorithms*, 2d ed., Addison-Wesley, Reading, Mass., 1973.
24. D. E. Knuth, *The Art of Computer Programming: Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.
25. K. Melhorn, "A Partial Analysis of Height-Balanced Trees under Random Insertions and Deletions," *SIAM Journal of Computing*, 11 (1982), 748–760.
26. K. Melhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, Berlin, 1984.
27. J. Nievergelt and E. M. Reingold, "Binary Search Trees of Bounded Balance," *SIAM Journal on Computing*, 2 (1973), 33–43.

28. A. J. Perlis and C. Thornton, "Symbol Manipulation in Threaded Lists," *Communications of the ACM*, 3 (1960), 195–204.
29. D. D. Sleator and R. E. Tarjan, "Self-adjusting Binary Search Trees," *Journal of ACM*, 32 (1985), 652–686.
30. D. D. Sleator, R. E. Tarjan, and W. P. Thurston, "Rotation Distance, Triangulations, and Hyperbolic Geometry," *Journal of AMS* (1988), 647–682.
31. H. F. Smith, *Data Structures—Form and Function*, Harcourt Brace Jovanovich, Orlando, Fla., 1987.
32. R. E. Tarjan, "Sequential Access in Splay Trees Takes Linear Time," *Combinatorica*, 5 (1985), 367–378.
33. A. C. Yao, "On Random 2–3 Trees," *Acta Informatica*, 9 (1978), 159–170.

CHAPTER 5

Hashing

In Chapter 4, we discussed the search tree ADT, which allowed various operations on a set of elements. In this chapter, we discuss the *hash table* ADT, which supports only a subset of the operations allowed by binary search trees.

The implementation of hash tables is frequently called *hashing*. Hashing is a technique used for performing insertions, deletions, and finds in constant average time. Tree operations that require any ordering information among the elements are not supported efficiently. Thus, operations such as *FindMin*, *FindMax*, and the printing of the entire table in sorted order in linear time are not supported.

The central data structure in this chapter is the *hash table*. We will

- See several methods of implementing the hash table.
- Compare these methods analytically.
- Show numerous applications of hashing.
- Compare hash tables with binary search trees.

5.1. General Idea

The ideal hash table data structure is merely an array of some fixed size, containing the keys. Typically, a key is a string with an associated value (for instance, salary information). We will refer to the table size as *TableSize*, with the understanding that this is part of a hash data structure and not merely some variable floating around globally. The common convention is to have the table run from 0 to *TableSize* − 1; we will see why shortly.

Each key is mapped into some number in the range 0 to *TableSize* − 1 and placed in the appropriate cell. The mapping is called a *hash function*, which ideally should be simple to compute and should ensure that any two distinct keys get different cells. Since there are a finite number of cells and a virtually inexhaustible supply of keys, this is clearly impossible, and thus we seek a hash function that distributes the keys evenly among the cells. Figure 5.1 is typical of a perfect situation. In this example, *john* hashes to 3, *phil* hashes to 4, *dave* hashes to 6, and *mary* hashes to 7.

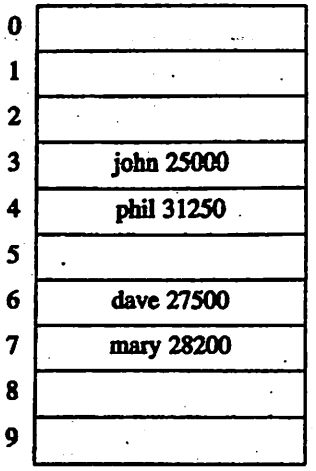


Figure 5.1 An ideal hash table

This is the basic idea of hashing. The only remaining problems deal with choosing a function, deciding what to do when two keys hash to the same value (this is known as a *collision*), and deciding on the table size.

5.2. Hash Function

If the input keys are integers, then simply returning *Key* mod *TableSize* is generally a reasonable strategy, unless *Key* happens to have some undesirable properties. In this case, the choice of hash function needs to be carefully considered. For instance, if the table size is 10 and the keys all end in zero, then the standard hash function is a bad choice. For reasons we shall see later, and to avoid situations like the one above, it is usually a good idea to ensure that the table size is prime. When the input keys are random integers, then this function is not only very simple to compute but also distributes the keys evenly.

Usually, the keys are strings; in this case, the hash function needs to be chosen carefully.

One option is to add up the ASCII values of the characters in the string. In Figure 5.2 we declare the type *Index*, which is returned by the hash function. The routine in Figure 5.3 implements this strategy and uses the typical C method of stepping through a string.

The hash function depicted in Figure 5.3 is simple to implement and computes an answer quickly. However, if the table size is large, the function does not distribute

Figure 5.2 Type returned by hash function

```
typedef unsigned int Index;
```



```
Index
Hash( const char *Key, int TableSize )
{
    unsigned int HashVal = 0;

/* 1*/      while( *Key != '\0' )
/* 2*/          HashVal += *Key++;

/* 3*/      return HashVal % TableSize;
}
```

Figure 5.3 A simple hash function

the keys well. For instance, suppose that *TableSize* = 10,007 (10,007 is a prime number). Suppose all the keys are eight or fewer characters long. Since a *char* has an integer value that is always at most 127, the hash function can only assume values between 0 and 1,016, which is 127 * 8. This is clearly not an equitable distribution!

Another hash function is shown in Figure 5.4. This hash function assumes that *Key* has at least two characters plus the *NULL* terminator. The value 27 represents the number of letters in the English alphabet, plus the blank, and 729 is 27^2. This function examines only the first three characters, but if these are random and the table size is 10,007, as before, then we would expect a reasonably equitable distribution. Unfortunately, English is not random. Although there are $26^3 = 17{,}576$ possible combinations of three characters (ignoring blanks), a check of a reasonably large on-line dictionary reveals that the number of different combinations is actually only 2,851. Even if none of *these* combinations collide, only 28 percent of the table can actually be hashed to. Thus this function, although easily computable, is also not appropriate if the hash table is reasonably large.

Figure 5.5 shows a third attempt at a hash function. This hash function involves all characters in the key and can generally be expected to distribute well (it computes $\sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 32^i$, and brings the result into proper range). The code computes a polynomial function (of 32) by use of Horner's rule. For instance, another way of computing $h_k = k_1 + 27k_2 + 27^2k_3$ is by the formula $h_k = ((k_3) * 27 + k_2) * 27 + k_1$. Horner's rule extends this to an nth degree polynomial.

Figure 5.4 Another possible hash function—not too good

```
Index
Hash( const char *Key, int TableSize )
{
    return ( Key[ 0 ] + 27 * Key[ 1 ] + 729 * Key[ 2 ] )
                % TableSize;
}
```

```
        Index
        Hash( const char *Key, int TableSize )
        {
            unsigned int HashVal = 0;

/* 1*/      while( *Key != '\0' )
/* 2*/          HashVal = ( HashVal << 5 ) + *Key++;

/* 3*/      return HashVal % TableSize;
        }
```

Figure 5.5 A good hash function

We have used 32 instead of 27, because multiplication by 32 is not really a multiplication, but amounts to bit-shifting by 5. In line 2, the addition could be replaced with a bitwise Exclusive Or, for increased speed.

The hash function described in Figure 5.5 is not necessarily the best with respect to table distribution, but does have the merit of extreme simplicity (and speed if overflows are allowed). If the keys are very long, the hash function will take too long to compute. Furthermore, the early characters will wind up being left-shifted out of the eventual answer. A common practice in this case is not to use all the characters. The length and properties of the keys would then influence the choice. For instance, the keys could be a complete street address. The hash function might include a couple of characters from the street address and perhaps a couple of characters from the city name and ZIP code. Some programmers implement their hash function by using only the characters in the odd spaces, with the idea that the time saved computing the hash function will make up for a slightly less evenly distributed function.

The main programming detail left is collision resolution. If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a *collision* and need to resolve it. There are several methods for dealing with this. We will discuss two of the simplest: separate chaining and open addressing.

5.3. Separate Chaining

The first strategy, commonly known as *separate chaining*, is to keep a list of all elements that hash to the same value. For convenience, our lists have headers. This makes the list implementation the same as in Chapter 3. If space is tight, it might be preferable to avoid their use. We assume for this section that the keys are the first 10 perfect squares and that the hashing function is simply $Hash(X) = X \bmod 10$. (The table size is not prime but is used here for simplicity.) Figure 5.6 should make this clear.

To perform a *Find*, we use the hash function to determine which list to traverse. We then traverse this list in the normal manner, returning the position where the

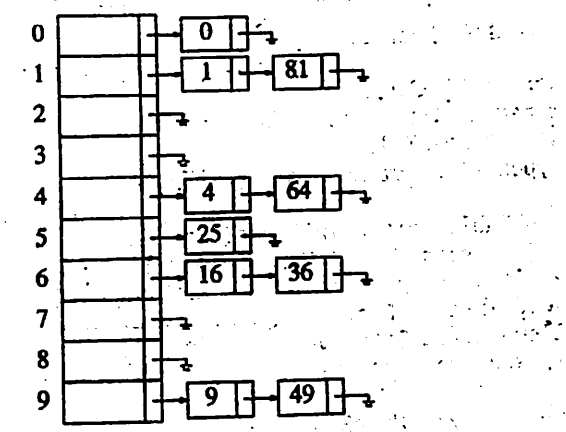


Figure 5.6 A separate chaining hash table

item is found. To perform an *Insert*, we traverse down the appropriate list to check whether the element is already in place (if duplicates are expected, an extra field is usually kept, and this field would be incremented in the event of a match). If the element turns out to be new, it is inserted either at the front of the list or at the end of the list, whichever is easiest. This is an issue most easily addressed while the code is being written. Sometimes new elements are inserted at the front of the list, since it is convenient and also because frequently it happens that recently inserted elements are the most likely to be accessed in the near future.

The type declarations required to implement separate chaining are shown in Figure 5.7. The *ListNode* structure is the same as the linked list declarations of Chapter 3. The hash table structure contains an array of linked lists (and the number of lists in the array), which are dynamically allocated when the table is initialized. The *HashTable* type is just a pointer to this structure.

Notice that the *TheLists* field is actually a pointer to a pointer to a *ListNode* structure. If typedefs and abstraction are not used, this can be quite confusing.

Figure 5.8 shows the initialization function, which uses the same ideas that were seen in the array implementation of stacks. Lines 4 through 6 allocate a hash table structure. If space is available, then *H* will point to a structure containing an integer and a pointer to a list. Line 7 sets the table size to a prime number, and lines 8 through 10 attempt to allocate an array of lists. Since a *List* is defined to be a pointer, the result is an array of pointers.

If our *List* implementation were not using headers, we could stop here. Since our implementation uses headers, we must allocate one header per list and set its *Next* field to *NULL*. This is done in lines 11 through 15. Of course, lines 12 through 15 could be replaced with the statement

```
H->TheLists[ i ] = MakeEmpty( );
```

Although we have not used this option, because in this instance it is preferable to make the code as self-contained as possible, it is certainly worth considering. An

```
#ifndef _HashSep_H

struct ListNode;
typedef struct ListNode *Position;
struct HashTbl;
typedef struct HashTbl *HashTable;

HashTable InitializeTable( int TableSize );
void DestroyTable( HashTable H );
Position Find( ElementType Key, HashTable H );
void Insert( ElementType Key, HashTable H );
ElementType Retrieve( Position P );
/* Routines such as Delete and MakeEmpty are omitted */

#endif  /* _HashSep_H */



/* Place in the implementation file */
struct ListNode
{
    ElementType Element;
    Position    Next;
};

typedef Position List;

/* List *TheList will be an array of lists, allocated later */
/* The lists use headers (for simplicity), */
/* though this wastes space */
struct HashTbl
{
    int TableSize;
    List *TheLists;
};
```

Figure 5.7 Type declaration for separate chaining hash table

inefficiency of our code is that the *malloc* on line 12 is performed *H->TableSize* times. This can be avoided by replacing line 12 with one call to *malloc* before the loop occurs:

```
H->TheLists = malloc (H->TableSize * sizeof (struct ListNode));
```

Line 16 returns *H*.

The call *Find(Key, H)* will return a pointer to the cell containing *Key*. The code to implement this is shown in Figure 5.9. Notice that lines 2 through 5 are identical to the code to perform a *Find* that is given in Chapter 3. Thus, the list ADT implementation in Chapter 3 could be used here. Remember that if *ElementType*


```
        HashTable
        InitializeTable( int TableSize )
        {
            HashTable H;
            int i;

/* 1*/      if( TableSize < MinTableSize )
            {
/* 2*/          Error( "Table size too small" );
/* 3*/          return NULL;
            }

            /* Allocate table */
/* 4*/      H = malloc( sizeof( struct HashTbl ) );
/* 5*/      if( H == NULL )
/* 6*/          FatalError( "Out of space!!!" );

/* 7*/      H->TableSize = NextPrime( TableSize );

            /* Allocate array of lists */
/* 8*/      H->TheLists = malloc( sizeof( List ) * H->TableSize );
/* 9*/      if( H->TheLists == NULL )
/*10*/          FatalError( "Out of space!!!" );

            /* Allocate list headers */
/*11*/      for( i = 0; i < H->TableSize; i++ )
            {
/*12*/          H->TheLists[ i ] = malloc( sizeof( struct ListNode ) );
/*13*/          if( H->TheLists[ i ] == NULL )
/*14*/              FatalError( "Out of space!!!" );
                else
/*15*/              H->TheLists[ i ]->Next = NULL;
            }

/*16*/      return H;
        }
```

Figure 5.8 Initialization routine for separate chaining hash table

is a string, comparison and assignment must be done with *strcmp* and *strcpy*, respectively.

Next comes the insertion routine. If the item to be inserted is already present, then we do nothing; otherwise we place it at the front of the list (see Fig. 5.10).* The element can be placed anywhere in the list; this is most convenient in our case.

*Since the table in Figure 5.6 was created by inserting at the end of the list, the code in Figure 5.10 will produce a table with the lists in Figure 5.6 reversed.

```
        Position
        Find( ElementType Key, HashTable H )
        {
            Position P;
            List L;

/* 1*/      L = H->TheLists[ Hash( Key, H->TableSize ) ];
/* 2*/      P = L->Next;
/* 3*/      while( P != NULL && P->Element != Key )
                                /* Probably need strcmp!! */
/* 4*/          P = P->Next;
/* 5*/      return P;
        }
```

Figure 5.9 *Find* routine for separate chaining hash table

Notice that the code to insert at the front of the list is essentially identical to the code in Chapter 3 that implements a *Push* using linked lists. Again, if the ADTs in Chapter 3 have already been carefully implemented, they can be used here.

The insertion routine coded in Figure 5.10 is somewhat poorly coded, because it computes the hash function twice. Redundant calculations are always bad, so this code should be rewritten if it turns out that the hash routines account for a significant portion of a program's running time.

Figure 5.10 *Insert* routine for separate chaining hash table

```
        void
        Insert( ElementType Key, HashTable H )
        {
            Position Pos, NewCell;
            List L;

/* 1*/      Pos = Find( Key, H );
/* 2*/      if( Pos == NULL )   /* Key is not found */
            {
/* 3*/          NewCell = malloc( sizeof( struct ListNode ) );
/* 4*/          if( NewCell == NULL )
/* 5*/              FatalError( "Out of space!!!" );
                else
                {
/* 6*/              L = H->TheLists[ Hash( Key, H->TableSize ) ];
/* 7*/              NewCell->Next = L->Next;
/* 8*/              NewCell->Element = Key;  /* Probably need strcpy!
/* 9*/              L->Next = NewCell;
                }
            }
        }
```


The deletion routine is a straightforward implementation of deletion in a linked list, so we will not bother with it here. If the repertoire of hash routines does not include deletions, it is probably best to not use headers, since their use would provide no simplification and would waste considerable space. We leave this as an exercise, too.

Any scheme could be used besides linked lists to resolve the collisions; a binary search tree or even another hash table would work, but we expect that if the table is large and the hash function is good, all the lists should be short, so it is not worthwhile to try anything complicated.

We define the load factor, λ, of a hash table to be the ratio of the number of elements in the hash table to the table size. In the example above, $\lambda = 1.0$. The average length of a list is λ. The effort required to perform a search is the constant time required to evaluate the hash function plus the time to traverse the list. In an unsuccessful search, the number of links to traverse is λ (excluding the final *NULL* link) on average. A successful search requires that about $1 + (\lambda/2)$ links be traversed; there is a guarantee that one link must be traversed (since the search is successful), and we also expect to go halfway down a list to find our match. This analysis shows that the table size is not really important, but the load factor is. The general rule for separate chaining hashing is to make the table size about as large as the number of elements expected (in other words, let $\lambda \approx 1$). It is also a good idea, as mentioned before, to keep the table size prime to ensure a good distribution.

5.4. Open Addressing

Separate chaining hashing has the disadvantage of requiring pointers. This tends to slow the algorithm down a bit because of the time required to allocate new cells, and also essentially requires the implementation of a second data structure. *Open addressing hashing* is an alternative to resolving collisions with linked lists. In an open addressing hashing system, if a collision occurs, alternative cells are tried until an empty cell is found. More formally, cells $h_0(X), h_1(X), h_2(X), \ldots$ are tried in succession, where $h_i(X) = (Hash(X) + F(i)) \bmod TableSize$, with $F(0) = 0$. The function, F, is the collision resolution strategy. Because all the data go inside the table, a bigger table is needed for open addressing hashing than for separate chaining hashing. Generally, the load factor should be below $\lambda = 0.5$ for open addressing hashing. We now look at three common collision resolution strategies.

5.4.1. Linear Probing

In linear probing, F is a linear function of i, typically $F(i) = i$. This amounts to trying cells sequentially (with wraparound) in search of an empty cell. Figure 5.11 shows the result of inserting keys {89, 18, 49, 58, 69} into a hash table using the same hash function as before and the collision resolution strategy, $F(i) = i$.

The first collision occurs when 49 is inserted; it is put in the next available spot, namely, spot 0, which is open. The key 58 collides with 18, 89, and then 49

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.11 Open addressing hash table with linear probing, after each insertion

before an empty cell is found three away. The collision for 69 is handled in a similar manner. As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large. Worse, even if the table is relatively empty, blocks of occupied cells start forming. This effect, known as *primary clustering*, means that any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

Although we will not perform the calculations here, it can be shown that the expected number of probes using linear probing is roughly $\frac{1}{2}(1 + 1/(1-\lambda)^2)$ for insertions and unsuccessful searches, and $\frac{1}{2}(1 + 1/(1-\lambda))$ for successful searches. The calculations are somewhat involved. It is easy to see from the code that insertions and unsuccessful searches require the same number of probes. A moment's thought suggests that, on average, successful searches should take less time than unsuccessful searches.

The corresponding formulas, if clustering is not a problem, are fairly easy to derive. We will assume a very large table and that each probe is independent of the previous probes. These assumptions are satisfied by a *random* collision resolution strategy and are reasonable unless λ is very close to 1. First, we derive the expected number of probes in an unsuccessful search. This is just the expected number of probes until we find an empty cell. Since the fraction of empty cells is $1 - \lambda$, the number of cells we expect to probe is $1/(1-\lambda)$. The number of probes for a successful search is equal to the number of probes required when the particular element was inserted. When an element is inserted, it is done as a result of an unsuccessful search. Thus, we can use the cost of an unsuccessful search to compute the average cost of a successful search.

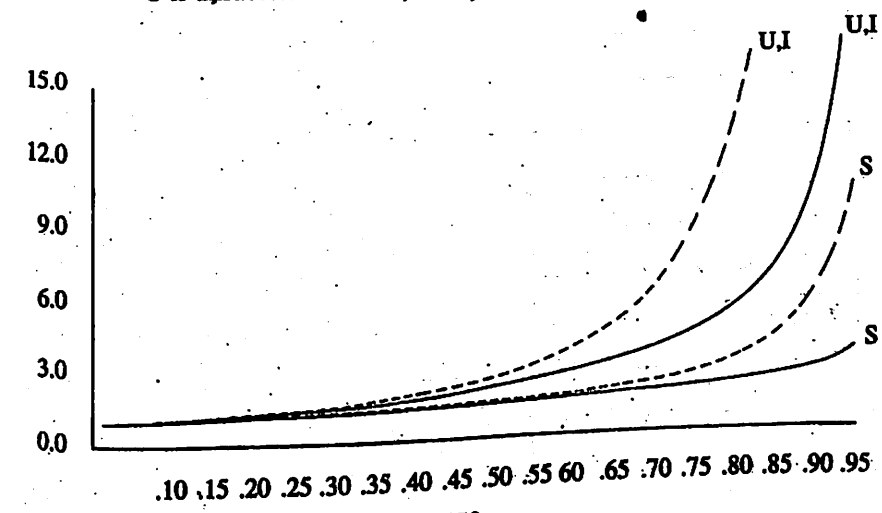
The caveat is that λ changes from 0 to its current value, so that earlier insertions are cheaper and should bring the average down. For instance, in the table above, $\lambda = 0.5$, but the cost of accessing 18 is determined when 18 is inserted. At that point, $\lambda = 0.2$. Since 18 was inserted into a relatively empty table, accessing it should be easier than accessing a recently inserted element such as 69. We can estimate the average by using an integral to calculate the mean value of the insertion time, obtaining

$$I(\lambda) = \frac{1}{\lambda}\int_0^{\lambda} \frac{1}{1-x}dx = \frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$

These formulas are clearly better than the corresponding formulas for linear probing. Clustering is not only a theoretical problem but actually occurs in real implementations. Figure 5.12 compares the performance of linear probing (dashed curves) with what would be expected from more random collision resolution. Successful searches are indicated by an *S*, and unsuccessful searches and insertions are marked with *U* and *I*, respectively.

If $\lambda = 0.75$, then the formula above indicates that 8.5 probes are expected for an insertion in linear probing. If $\lambda = 0.9$, then 50 probes are expected, which is unreasonable. This compares with 4 and 10 probes for the respective load factors if clustering were not a problem. We see from these formulas that linear probing can be a bad idea if the table is expected to be more than half full. If $\lambda = 0.5$, however, only 2.5 probes are required on average for insertion, and only 1.5 probes are required, on average, for a successful search.

Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (*S* is successful search, *U* is unsuccessful search, and *I* is insertion)



5.4.2. Quadratic Probing

Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. Quadratic probing is what you would expect—the collision function is quadratic. The popular choice is $F(i) = i^2$. Figure 5.13 shows the resulting open addressing hash table with this collision function on the same input used in the linear probing example.

When 49 collides with 89, the next position attempted is one cell away. This cell is empty, so 49 is placed there. Next 58 collides at position 8. Then the cell one away is tried, but another collision occurs. A vacant cell is found at the next cell tried, which is $2^2 = 4$ away. The key 58 is thus placed in cell 2. The same thing happens for 69.

For linear probing it is a bad idea to let the hash table get nearly full, because performance degrades. For quadratic probing, the situation is even more drastic: There is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternative locations to resolve collisions.

Indeed, we prove now that if the table is half empty and the table size is prime, then we are always guaranteed to be able to insert a new element.

THEOREM 5.1.

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

Figure 5.13 Open addressing hash table with quadratic probing, after each insertion

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

PROOF:

Let the table size, *TableSize*, be an (odd) prime greater than 3. We show that the first $\lfloor TableSize/2 \rfloor$ alternative locations are all distinct. Two of these locations are $h(X) + i^2 \pmod{TableSize}$ and $h(X) + j^2 \pmod{TableSize}$, where $0 < i, j \leq \lfloor TableSize/2 \rfloor$. Suppose, for the sake of contradiction, that these locations are the same, but $i \neq j$. Then

$$h(X) + i^2 = h(X) + j^2 \pmod{TableSize}$$
$$i^2 = j^2 \pmod{TableSize}$$
$$i^2 - j^2 = 0 \pmod{TableSize}$$
$$(i - j)(i + j) = 0 \pmod{TableSize}$$

Since *TableSize* is prime, it follows that either $(i - j)$ or $(i + j)$ is equal to 0 (mod *TableSize*). Since i and j are distinct, the first option is not possible. Since $0 < i, j < \lfloor TableSize/2 \rfloor$, the second option is also impossible. Thus, the first $\lfloor TableSize/2 \rfloor$ alternative locations are distinct. Since the element to be inserted can also be placed in the cell to which it hashes (if there are no collisions), any element has $\lceil TableSize/2 \rceil$ locations into which it can go. If at most $\lfloor TableSize/2 \rfloor$ positions are taken, then an empty spot can always be found.

If the table is even one more than half full, the insertion could fail (although this is extremely unlikely). Therefore, it is important to keep this in mind. It is also crucial that the table size be prime.* If the table size is not prime, the number of alternative locations can be severely reduced. As an example, if the table size were 16, then the only alternative locations would be at distances 1, 4, or 9 away.

Standard deletion cannot be performed in an open addressing hash table, because the cell might have caused a collision to go past it. For instance, if we remove 89, then virtually all of the remaining *Finds* will fail. Thus, open addressing hash tables require lazy deletion, although in this case there really is no laziness implied.

The type declarations required to implement open addressing hashing are shown in Figure 5.14. Instead of an array of lists, we have an array of hash table entry cells, which, as in separate chaining hashing, are allocated dynamically. Initializing the table (Fig. 5.15) consists of allocating space (lines 1 through 10) and then setting the *Info* field to *Empty* for each cell.

As with separate chaining hashing, *Find*(*Key*, *H*) will return the position of *Key* in the hash table. If *Key* is not present, then *Find* will return the last cell. This cell is where *Key* would be inserted if needed. Further, because it is marked *Empty*, it is easy to tell that the *Find* failed. We assume for convenience that the hash table is at least twice as large as the number of elements in the table, so quadratic resolution will always work. Otherwise, we would need to test i (*CollisionNum*) before line 4.

*If the table size is a prime of the form $4k + 3$, and the quadratic collision resolution strategy $F(i) = \pm i^2$ is used, then the entire table can be probed. The cost is a slightly more complicated routine.

```
#ifndef _HashQuad_H

typedef unsigned int Index;
typedef Index Position;

struct HashTbl;
typedef struct HashTbl *HashTable;

HashTable InitializeTable( int TableSize );
void DestroyTable( HashTable H );
Position Find( ElementType Key, HashTable H );
void Insert( ElementType Key, HashTable H );
ElementType Retrieve( Position P, HashTable H );
HashTable Rehash( HashTable H );
/* Routines such as Delete and MakeEmpty are omitted */

#endif /* _HashQuad_H */


/* Place in the implementation file */
enum KindOfEntry { Legitimate, Empty, Deleted };

struct HashEntry
{
    ElementType  Element;
    enum KindOfEntry Info;
};

typedef struct HashEntry Cell;

/* Cell *TheCells will be an array of */
/* HashEntry cells, allocated later */
struct HashTbl
{
    int TableSize;
    Cell *TheCells;
};
```

Figure 5.14 Type declaration for open addressing hash tables

In the implementation in Figure 5.16, elements that are marked as deleted count as being in the table. This can cause problems, because the table can get too full prematurely. We shall discuss this item presently.

Lines 4 through 6 represent the fast way of doing quadratic resolution. From the definition of the quadratic resolution function, $F(i) = F(i-1) + 2i - 1$, so the next cell to try can be determined with a multiplication by 2 (really a bit


```
        HashTable
        InitializeTable( int TableSize )
        {
            HashTable H;
            int i;

/* 1*/      if( TableSize < MinTableSize )
            {
/* 2*/          Error( "Table size too small" );
/* 3*/          return NULL;
            }

            /* Allocate table */
/* 4*/      H = malloc( sizeof( struct HashTbl ) );
/* 5*/      if( H == NULL )
/* 6*/          FatalError( "Out of space!!!" );

/* 7*/      H->TableSize = NextPrime( TableSize );

            /* Allocate array of Cells */
/* 8*/      H->TheCells = malloc( sizeof( Cell ) * H->TableSize );
/* 9*/      if( H->TheCells == NULL )
/*10*/          FatalError( "Out of space!!!" );

/*11*/      for( i = 0; i < H->TableSize; i++ )
/*12*/          H->TheCells[ i ].Info = Empty;

/*13*/      return H;
        }
```

Figure 5.15 Routine to initialize open addressing hash table

shift) and a decrement. If the new location is past the array, it can be put back in range by subtracting *TableSize*. This is faster than the obvious method, because it avoids the multiplication and division that seem to be required. An important warning: The order of testing at line 3 is important. Don't switch it!

The final routine is insertion. As with separate chaining hashing, we do nothing if *Key* is already present. It is a simple modification to do something else. Otherwise, we place it at the spot suggested by the *Find* routine. The code is shown in Figure 5.17.

Although quadratic probing eliminates primary clustering, elements that hash to the same position will probe the same alternative cells. This is known as *secondary clustering*. Secondary clustering is a slight theoretical blemish. Simulation results suggest that it generally causes less than an extra half probe per search. The following technique eliminates this, but does so at the cost of extra multiplications and divisions.

```
Position
Find( ElementType Key, HashTable H )
{
    Position CurrentPos;
    int CollisionNum;

/* 1*/  CollisionNum = 0;
/* 2*/  CurrentPos = Hash( Key, H->TableSize );
/* 3*/  while( H->TheCells[ CurrentPos ].Info != Empty &&
            H->TheCells[ CurrentPos ].Element != Key )
                /* Probably need strcmp!! */
        {
/* 4*/      CurrentPos += 2 * ++CollisionNum - 1;
/* 5*/      if( CurrentPos >= H->TableSize )
/* 6*/          CurrentPos -= H->TableSize;
        }
/* 7*/  return CurrentPos;
}
```

Figure 5.16 *Find* routine for hashing with quadratic probing

5.4.3. Double Hashing

The last collision resolution method we will examine is *double hashing*. For double hashing, one popular choice is $F(i) = i \cdot hash_2(X)$. This formula says that we apply a second hash function to X and probe at a distance $hash_2(X), 2hash_2(X), \ldots$, and so on. A poor choice of $hash_2(X)$ would be disastrous. For instance, the obvious choice $hash_2(X) = X \bmod 9$ would not help if 99 were inserted into the input in the previous examples. Thus, the function must never evaluate to zero. It is also important to make sure all cells can be probed (this is not possible in the example below, because

Figure 5.17 *Insert* routine for hash tables with quadratic probing

```
void
Insert( ElementType Key, HashTable H )
{
    Position Pos;

    Pos = Find( Key, H );
    if( H->TheCells[ Pos ].Info != Legitimate )
    {
            /* OK to insert here */
        H->TheCells[ Pos ].Info = Legitimate;
        H->TheCells[ Pos ].Element = Key;
                /* Probably need strcpy! */
    }
}
```


	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.18 Open addressing hash table with double hashing, after each insertion

the table size is not prime). A function such as $hash_2(X) = R - (X \bmod R)$, with R a prime smaller than *TableSize*, will work well. If we choose $R = 7$, then Figure 5.18 shows the results of inserting the same keys as before.

The first collision occurs when 49 is inserted. $hash_2(49) = 7 - 0 = 7$, so 49 is inserted in position 6. $hash_2(58) = 7 - 2 = 5$, so 58 is inserted at location 3. Finally, 69 collides and is inserted at a distance $hash_2(69) = 7 - 6 = 1$ away. If we tried to insert 60 in position 0, we would have a collision. Since $hash_2(60) = 7 - 4 = 3$, we would then try positions 3, 6, 9, and then 2 until an empty spot is found. It is generally possible to find some bad case, but there are not too many here.

As we have said before, the size of our sample hash table is not prime. We have done this for convenience in computing the hash function, but it is worth seeing why it is important to make sure the table size is prime when double hashing is used. If we attempt to insert 23 into the table, it would collide with 58. Since $hash_2(23) = 7 - 2 = 5$, and the table size is 10, we essentially have only one alternative location, and it is already taken. Thus, if the table size is not prime, it is possible to run out of alternative locations prematurely. However, if double hashing is correctly implemented, simulations imply that the expected number of probes is almost the same as for a random collision resolution strategy. This makes double hashing theoretically interesting. Quadratic probing, however, does not require the use of a second hash function and is thus likely to be simpler and faster in practice.

5.5. Rehashing

If the table gets too full, the running time for the operations will start taking too long and *Inserts* might fail for open addressing hashing with quadratic resolution. This

can happen if there are too many removals intermixed with insertions. A solution, then, is to build another table that is about twice as big (with an associated new hash function) and scan down the entire original hash table, computing the new hash value for each (nondeleted) element and inserting it in the new table.

As an example, suppose the elements 13, 15, 24, and 6 are inserted into an open addressing hash table of size 7. The hash function is $h(X) = X \bmod 7$. Suppose linear probing is used to resolve collisions. The resulting hash table appears in Figure 5.19.

If 23 is inserted into the table, the resulting table in Fig. 5.20 will be over 70 percent full. Because the table is so full, a new table is created. The size of this table is 17, because this is the first prime that is twice as large as the old table size. The new hash function is then $h(X) = X \bmod 17$. The old table is scanned, and elements 6, 15, 23, 24, and 13 are inserted into the new table. The resulting table appears in Figure 5.21.

This entire operation is called *rehashing*. This is obviously a very expensive operation; the running time is $O(N)$, since there are N elements to rehash and the

Figure 5.19 Open addressing hash table with linear probing with input 13, 15, 6, 24

0	6
1	15
2	
3	24
4	
5	
6	13

Figure 5.20 Open addressing hash table with linear probing after 23 is inserted

0	6
1	15
2	23
3	24
4	
5	
6	13

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Figure 5.21 Open addressing hash table after rehashing

table size is roughly $2N$, but it is actually not all that bad, because it happens very infrequently. In particular, there must have been $N/2$ *Inserts* prior to the last rehash, so it essentially adds a constant cost to each insertion.* If this data structure is part of the program, the effect is not noticeable. On the other hand, if the hashing is performed as part of an interactive system, then the unfortunate user whose insertion caused a rehash could see a slowdown.

Rehashing can be implemented in several ways with quadratic probing. One alternative is to rehash as soon as the table is half full. The other extreme is to rehash only when an insertion fails. A third, middle-of-the-road strategy is to rehash when the table reaches a certain load factor. Since performance does degrade as the load factor increases, the third strategy, implemented with a good cutoff, could be best.

Rehashing frees the programmer from worrying about the table size and is important because hash tables cannot be made arbitrarily large in complex programs. The exercises ask you to investigate the use of rehashing in conjunction with lazy

*This is why the new table is made twice as large as the old table.

```
        HashTable
        Rehash( HashTable H )
        {
            int i, OldSize;
            Cell *OldCells;

/* 1*/      OldCells = H->TheCells;
/* 2*/      OldSize  = H->TableSize;

            /* Get a new, empty table */
/* 3*/      H = InitializeTable( 2 * OldSize );

            /* Scan through old table, reinserting into new */
/* 4*/      for( i = 0; i < OldSize; i++ )
/* 5*/          if( OldCells[ i ].Info == Legitimate )
/* 6*/              Insert( OldCells[ i ].Element, H );

/* 7*/      free( OldCells );

/* 8*/      return H;
        }
```

Figure 5.22 Rehashing for open addressing hash tables

deletion. Rehashing can be used in other data structures as well. For instance, if the queue data structure of Chapter 3 became full, we could declare a double-sized array and copy everything over, freeing the original.

Figure 5.22 shows that rehashing is simple to implement.

5.6. Extendible Hashing

Our last topic in this chapter deals with the case where the amount of data is too large to fit in main memory. As we saw in Chapter 4, the main consideration then is the number of disk accesses required to retrieve data.

As before, we assume that at any point we have N records to store; the value of N changes over time. Furthermore, at most M records fit in one disk block. We will use $M = 4$ in this section.

If either open addressing hashing or separate chaining hashing is used, the major problem is that collisions could cause several blocks to be examined during a *Find*, even for a well-distributed hash table. Furthermore, when the table gets too full, an extremely expensive rehashing step must be performed, which requires $O(N)$ disk accesses.

A clever alternative, known as extendible hashing, allows a *Find* to be performed in two disk accesses. Insertions also require few disk accesses.

We recall from Chapter 4 that a B-tree has depth $O(\log_{M/2} N)$. As M increases, the depth of a B-tree decreases. We could in theory choose M to be so large that

the depth of the B-tree would be 1. Then any *Find* after the first would take one disk access, since, presumably, the root node could be stored in main memory. The problem with this strategy is that the branching factor is so high that it would take considerable processing to determine which leaf the data was in. If the time to perform this step could be reduced, then we would have a practical scheme. This is exactly the strategy used by extendible hashing.

Let us suppose, for the moment, that our data consists of several six-bit integers. Figure 5.23 shows an extendible hashing scheme for these data. The root of the "tree" contains four pointers determined by the leading two bits of the data. Each leaf has up to $M = 4$ elements. It happens that in each leaf the first two bits are identical; this is indicated by the number in parentheses. To be more formal, D will represent the number of bits used by the root, which is sometimes known as the *directory*. The number of entries in the directory is thus 2^D. d_L is the number of leading bits that all the elements of some leaf L have in common. d_L will depend on the particular leaf, and $d_L \leq D$.

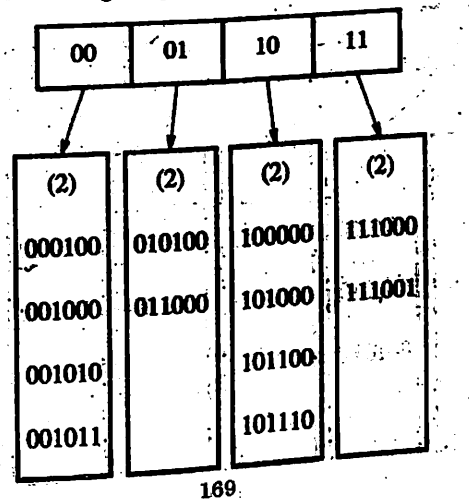
Suppose that we want to insert the key 100100. This would go into the third leaf, but as the third leaf is already full, there is no room. We thus split this leaf into two leaves, which are now determined by the first *three* bits. This requires increasing the directory size to 3. These changes are reflected in Figure 5.24.

Notice that all of the leaves not involved in the split are now pointed to by two adjacent directory entries. Thus, although an entire directory is rewritten, none of the other leaves is actually accessed.

If the key 000000 is now inserted, then the first leaf is split, generating two leaves with $d_L = 3$. Since $D = 3$, the only change required in the directory is the updating of the 000 and 001 pointers. See Figure 5.25.

This very simple strategy provides quick access times for *Insert* and *Find* operations on large databases. There are a few important details we have not considered.

Figure 5.23 Extendible hashing: original data



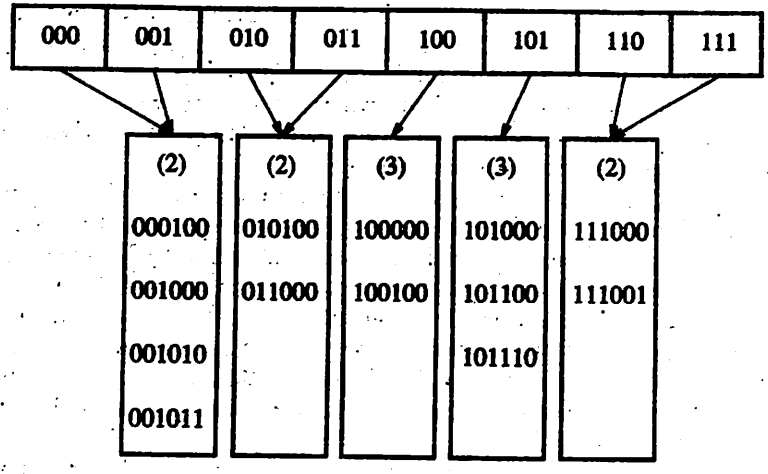
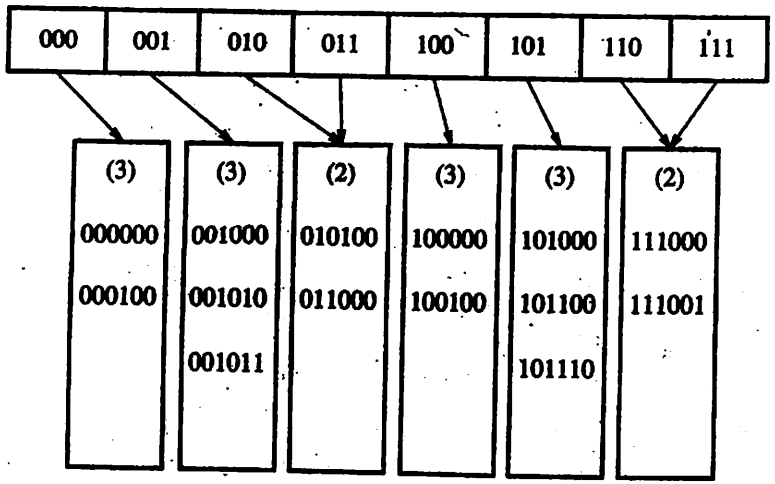


Figure 5.24 Extendible hashing: after insertion of 100100 and directory split

First, it is possible that several directory splits will be required if the elements in a leaf agree in more than $D + 1$ leading bits. For instance, starting at the original example, with $D = 2$, if 111010, 111011, and finally 111100 are inserted, the directory size must be increased to 4 to distinguish between the five keys. This is an easy detail to take care of, but must not be forgotten. Second, there is the possibility of duplicate keys; if there are more than M duplicates, then this algorithm does not work at all. In this case, some other arrangements need to be made.

These possibilities suggest that it is important for the bits to be fairly random. This can be accomplished by hashing the keys into a reasonably long integer—hence the name.

Figure 5.25 Extendible hashing: after insertion of 000000 and leaf split



We close by mentioning some of the performance properties of extendible hashing, which are derived after a very difficult analysis. These results are based on the reasonable assumption that the bit patterns are uniformly distributed.

The expected number of leaves is $(N/M)\log_2 e$. Thus the average leaf is $\ln 2 = 0.69$ full. This is the same as for B-trees, which is not entirely surprising, since for both data structures new nodes are created when the $(M + 1)$th entry is added.

The more surprising result is that the expected size of the directory (in other words, 2^D) is $O(N^{1+1/M}/M)$. If M is very small, then the directory can get unduly large. In this case, we can have the leaves contain pointers to the records instead of the actual records, thus increasing the value of M. This adds a second disk access to each *Find* operation in order to maintain a smaller directory. If the directory is too large to fit in main memory, the second disk access would be needed anyway.

Summary

Hash tables can be used to implement the *Insert* and *Find* operations in constant average time. It is especially important to pay attention to details such as load factor when using hash tables, since otherwise the time bounds are not valid. It is also important to choose the hash function carefully when the key is not a short string or integer.

For separate chaining hashing, the load factor should be close to 1, although performance does not significantly degrade unless the load factor becomes very large. For open addressing hashing, the load factor should not exceed 0.5, unless this is completely unavoidable. If linear probing is used, performance degenerates rapidly as the load factor approaches 1. Rehashing can be implemented to allow the table to grow (and shrink), thus maintaining a reasonable load factor. This is important if space is tight and it is not possible just to declare a huge hash table.

Binary search trees can also be used to implement *Insert* and *Find* operations. Although the resulting average time bounds are $O(\log N)$, binary search trees also support routines that require order and are thus more powerful. Using a hash table it is not possible to find the minimum element. It is not possible to search efficiently for a string unless the exact string is known. A binary search tree could quickly find all items in a certain range; this is not supported by hash tables. Furthermore, the $O(\log N)$ bound is not necessarily that much more than $O(1)$, especially since no multiplications or divisions are required by search trees.

On the other hand, the worst case for hashing generally results from an implementation error, whereas sorted input can make binary trees perform poorly. Balanced search trees are quite expensive to implement, so if no ordering information is required and there is any suspicion that the input might be sorted, then hashing is the data structure of choice.

Hashing applications are abundant. Compilers use hash tables to keep track of declared variables in source code. The data structure is known as a *symbol* table. Hash tables are the ideal application for this problem because only *Inserts* and *Finds* are performed. Identifiers are typically short, so the hash function can be computed quickly.

A hash table is useful for any graph theory problem where the nodes have real names instead of numbers. Here, as the input is read, vertices are assigned integers from 1 onward by order of appearance. Again, the input is likely to have large groups of alphabetized entries. For example, the vertices could be computers. Then if one particular installation lists its computers as *ibm1, ibm2, ibm3,...*, there could be a dramatic effect on efficiency if a search tree is used.

A third common use of hash tables is in programs that play games. As the program searches through different lines of play, it keeps track of positions it has seen by computing a hash function based on the position (and storing its move for that position). If the same position reoccurs, usually by a simple transposition of moves, the program can avoid expensive recomputation. This general feature of all game-playing programs is known as the *transposition table*.

Yet another use of hashing is in on-line spelling checkers. If misspelling detection (as opposed to correction) is important, an entire dictionary can be prehashed and words can be checked in constant time. Hash tables are well suited for this, because it is not important to alphabetize words; printing out misspellings in the order they occurred in the document is certainly acceptable.

We close this chapter by returning to the word puzzle problem of Chapter 1. If the second algorithm described in Chapter 1 is used, and we assume that the maximum word size is some small constant, then the time to read in the dictionary containing W words and put it in a hash table is $O(W)$. This time is likely to be dominated by the disk I/O and not the hashing routines. The rest of the algorithm would test for the presence of a word for each ordered quadruple (*row, column, orientation, number of characters*). As each lookup would be $O(1)$, and there are only a constant number of orientations (8) and characters per word, the running time of this phase would be $O(R \cdot C)$. The total running time would be $O(R \cdot C + W)$, which is a distinct improvement over the original $O(R \cdot C \cdot W)$. We could make further optimizations, which would decrease the running time in practice; these are described in the exercises.

Exercises

5.1 Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(X) = X \pmod{10}$, show the resulting:

a. Separate chaining hash table.

b. Open addressing hash table using linear probing.

c. Open addressing hash table using quadratic probing.

d. Open addressing hash table with second hash function $h_2(X) = 7 - (X \bmod 7)$.

5.2 Show the result of rehashing the hash tables in Exercise 5.1.

5.3 Write a program to compute the number of collisions required in a long random sequence of insertions using linear probing, quadratic probing, and double hashing.

5.4 A large number of deletions in a separate chaining hash table can cause the table to be fairly empty, which wastes space. In this case, we can rehash to a table half as large. Assume that we rehash to a larger table when there are twice as many elements as the table size. How empty should the table be before we rehash to a smaller table?

5.5 An alternative collision resolution strategy is to define a sequence, $F(i) = r_i$, where $r_0 = 0$ and $r_1, r_2, \ldots, r_N$ is a random permutation of the first N integers (each integer appears exactly once).

a. Prove that under this strategy, if the table is not full, then the collision can always be resolved.

b. Would this strategy be expected to eliminate clustering?

c. If the load factor of the table is λ, what is the expected time to perform an insert?

d. If the load factor of the table is λ, what is the expected time for a successful search?

e. Give an efficient algorithm (theoretically as well as practically) to generate the random sequence. Explain why the rules for choosing P are important.

5.6 What are the advantages and disadvantages of the various collision resolution strategies?

5.7 Write a program to implement the following strategy for multiplying two sparse polynomials P_1, P_2 of size M and N, respectively. Each polynomial is represented as a linked list with cells consisting of a coefficient, an exponent, and a *Next* pointer (Exercise 3.7). We multiply each term in P_1 by a term in P_2 for a total of MN operations. One method is to sort these terms and combine like terms, but this requires sorting MN records, which could be expensive, especially in small-memory environments. Alternatively, we could merge terms as they are computed and then sort the result.

a. Write a program to implement the alternative strategy.

b. If the output polynomial has about $O(M + N)$ terms, what is the running time of both methods?

5.8 A spelling checker reads an input file and prints out all words not in some on-line dictionary. Suppose the dictionary contains 30,000 words and the file is large, so that the algorithm can make only one pass through the input file. A simple strategy is to read the dictionary into a hash table and look for each input word as it is read. Assuming that an average word is seven characters and that it is possible to store words of length L in $L+1$ bytes (so space waste is not much of a consideration), and assuming an open addressing table, how much space does this require?

5.9 If memory is limited and the entire dictionary cannot be stored in a hash table, we can still get an efficient algorithm that almost always works. We declare an array *Table* of bits (initialized to zeros) from 0 to *TableSize* − 1. As we read in a word, we set *Table*[*Hash*(*Word*)] = 1. Which of the following is true?

a. If a word hashes to a location with value 0, the word is not in the dictionary.

b. If a word hashes to a location with value 1, then the word is in the dictionary.

Suppose we choose *TableSize* = 300,007.

c. How much memory does this require?

d. What is the probability of an error in this algorithm?

e. A typical document might have about three actual misspellings per page of 500 words. Is this algorithm usable?

*5.10 Describe a procedure that avoids initializing a hash table (at the expense of memory).

5.11 Suppose we want to find the first occurrence of a string $P_1P_2 \cdots P_k$ in a long input string $A_1A_2 \cdots A_N$. We can solve this problem by hashing the pattern string, obtaining a hash value H_P, and comparing this value with the hash value formed from $A_1A_2 \cdots A_k, A_2A_3 \cdots A_{k+1}, A_3A_4 \cdots A_{k+2}$, and so on until $A_{N-k+1}A_{N-k+2} \cdots A_N$. If we have a match of hash values, we compare the strings character by character to verify the match. We return the position (in A) if the strings actually do match, and we continue in the unlikely event that the match is false.

*a. Show that if the hash value of $A_iA_{i+1} \cdots A_{i+k-1}$ is known, then the hash value of $A_{i+1}A_{i+2} \cdots A_{i+k}$ can be computed in constant time.

b. Show that the running time is $O(k + N)$ plus the time spent refuting false matches.

*c. Show that the expected number of false matches is negligible.

d. Write a program to implement this algorithm.

**e. Describe an algorithm that runs in $O(k + N)$ worst-case time.

**f. Describe an algorithm that runs in $O(N/k)$ average time.

5.12 A BASIC program consists of a series of statements numbered in ascending order. Control is passed by use of a *goto* or *gosub* and a statement number. Write a program that reads in a legal BASIC program and renumbers the statements so that the first starts at number F and each statement has a number D higher than the previous statement. You may assume an upper limit of N statements, but the statement numbers in the input might be as large as a 32-bit integer. Your program must run in linear time.

5.13 a. Implement the word puzzle program using the algorithm described at the end of the chapter.

b. We can get a big speed increase by storing, in addition to each word W, all of W's prefixes. (If one of W's prefixes is another word in the dictionary, it is stored as a real word.) Although this may seem to increase the size of the hash table drastically, it does not, because many words have the same prefixes. When a scan is performed in a particular direction, if the word that is looked up is not even in the hash table as a prefix, then the scan in that direction can be terminated early. Use this idea to write an improved program to solve the word puzzle.

c. If we are willing to sacrifice the sanctity of the hash table ADT, we can speed up the program in part (b) by noting that if, for example, we have just computed the hash function for "excel," we do not need to compute the hash function for "excels" from scratch. Adjust your hash function so that it can take advantage of its previous calculation.

d. In Chapter 2, we suggested using binary search. Incorporate the idea of using prefixes into your binary search algorithm. The modification should be simple. Which algorithm is faster?

5.14 Show the result of inserting the keys 10111101, 00000010, 10011011, 10111110, 01111111, 01010001, 10010110, 00001011, 11001111, 10011110, 11011011, 00101011, 01100001, 11110000, 01101111 into an initially empty extendible hashing data structure with $M = 4$.

5.15 Write a program to implement extendible hashing. If the table is small enough to fit in main memory, how does its performance compare with separate chaining and open addressing hashing?

References

Despite the apparent simplicity of hashing, much of the analysis is quite difficult and there are still many unresolved questions. There are also many interesting theoretical issues, which generally attempt to make it unlikely that the worst-case possibilities of hashing arise.

An early paper on hashing is [17]. A wealth of information on the subject, including an analysis of hashing with linear probing, can be found in [11]. An excellent survey on the subject is [14]; [15] contains suggestions, and pitfalls, for choosing hash functions. Precise analytic and simulation results for all of the methods described in this chapter can be found in [8].

An analysis of double hashing can be found in [9] and [13]. Yet another collision resolution scheme is coalesced hashing, described in [18]. Yao [20] has shown that uniform hashing, in which no clustering exists, is optimal with respect to cost of a successful search.

If the input keys are known in advance, then perfect hash functions, which do not allow collisions, exist [2], [7]. Some more complicated hashing schemes, for which the worst case depends not on the particular input but on random numbers chosen by the algorithm, appear in [3] and [4].

Extendible hashing appears in [5], with analysis in [6] and [19].

One method of implementing Exercise 5.5 is described in [16]. Exercise 5.11 (a–d) is from [10]. Part (e) is from [12], and part (f) is from [1].

1. R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, 20 (1977), 762–772.
2. J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *Journal of Computer and System Sciences*, 18 (1979), 143–154.

3. M. Dietzfelbinger, A. R. Karlin, K. Melhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, "Dynamic Perfect Hashing: Upper and Lower Bounds," *SIAM Journal on Computing*, 23 (1994), 738–761.
4. R. J. Enbody and H. C. Du, "Dynamic Hashing Schemes," *Computing Surveys*, 20 (1988), 85–113.
5. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing—A Fast Access Method for Dynamic Files," *ACM Transactions on Database Systems*, 4 (1979), 315–344.
6. P. Flajolet, "On the Performance Evaluation of Extendible Hashing and Trie Searching," *Acta Informatica*, 20 (1983), 345–369.
7. M. L. Fredman, J. Komlos, and E. Szemeredi, "Storing a Sparse Table with O(1) Worst Case Access Time," *Journal of the ACM*, 31 (1984), 538–544.
8. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2nd ed., Addison-Wesley, Reading, Mass., 1991.
9. L. J. Guibas and E. Szemeredi, "The Analysis of Double Hashing," *Journal of Computer and System Sciences*, 16 (1978), 226–274.
10. R. M. Karp and M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *Aiken Computer Laboratory Report TR-31-81*, Harvard University, Cambridge, Mass., 1981.
11. D. E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.
12. D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, 6 (1977), 323–350.
13. G. Lueker and M. Molodowitch, "More Analysis of Double Hashing," *Proceedings of the Twentieth ACM Symposium on Theory of Computing* (1988), 354–359.
14. W. D. Maurer and T. G. Lewis, "Hash Table Methods," *Computing Surveys*, 7 (1975), 5–20.
15. B. J. McKenzie, R. Harries, and T. Bell, "Selecting a Hashing Algorithm," *Software—Practice and Experience*, 20 (1990), 209–224.
16. R. Morris, "Scatter Storage Techniques," *Communications of the ACM*, 11 (1968), 38–44.
17. W. W. Peterson, "Addressing for Random Access Storage," *IBM Journal of Research and Development*, 1 (1957), 130–146.
18. J. S. Vitter, "Implementations for Coalesced Hashing," *Communications of the ACM*, 25 (1982), 911–926.
19. A. C. Yao, "A Note on The Analysis of Extendible Hashing," *Information Processing Letters*, 11 (1980), 84–86.
20. A. C. Yao, "Uniform Hashing Is Optimal," *Journal of the ACM*, 32 (1985), 687–693.

CHAPTER 6

Priority Queues (Heaps)

Although jobs sent to a printer are generally placed on a queue, this might not always be the best thing to do. For instance, one job might be particularly important, so it might be desirable to allow that job to be run as soon as the printer is available. Conversely, if, when the printer becomes available, there are several 1-page jobs and one 100-page job, it might be reasonable to make the long job go last, even if it is not the last job submitted. (Unfortunately, most systems do not do this, which can be particularly annoying at times.)

Similarly, in a multiuser environment, the operating system scheduler must decide which of several processes to run. Generally a process is allowed to run only for a fixed period of time. One algorithm uses a queue. Jobs are initially placed at the end of the queue. The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up, and place it at the end of the queue if it does not finish. This strategy is generally not appropriate, because very short jobs will seem to take a long time because of the wait involved to run. Generally, it is important that short jobs finish as fast as possible, so these jobs should have precedence over jobs that have already been running. Furthermore, some jobs that are not short are still very important and should also have precedence.

This particular application seems to require a special kind of queue, known as a *priority queue*. In this chapter, we will discuss

- Efficient implementation of the priority queue ADT.
- Uses of priority queues.
- Advanced implementations of priority queues.

The data structures we will see are among the most elegant in computer science.

6.1. Model

A priority queue is a data structure that allows at least the following two operations: *Insert*, which does the obvious thing; and *DeleteMin*, which finds, returns, and removes the minimum element in the priority queue. The *Insert* operation is the equivalent of *Enqueue*, and *DeleteMin* is the priority queue equivalent of the queue's *Dequeue* operation. The *DeleteMin* function also alters its input. Current

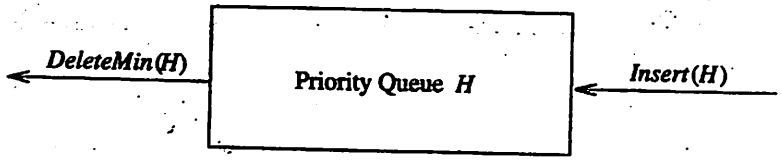


Figure 6.1 Basic model of a priority queue

thinking in the software engineering community suggests that this is no longer a good idea. However, we will continue to use this function for historical reasons; many programmers expect *DeleteMin* to operate this way.

As with most data structures, it is sometimes possible to add other operations, but these are extensions and not part of the basic model depicted in Figure 6.1.

Priority queues have many applications besides operating systems. In Chapter 7, we will see how priority queues are used for external sorting. Priority queues are also important in the implementation of *greedy algorithms*, which operate by repeatedly finding a minimum; we will see specific examples in Chapters 9 and 10. In this chapter we will see a use of priority queues in discrete event simulation.

6.2. Simple Implementations

There are several obvious ways to implement a priority queue. We could use a simple linked list, performing insertions at the front in $O(1)$ and traversing the list, which requires $O(N)$ time, to delete the minimum. Alternatively, we could insist that the list be kept always sorted; this makes insertions expensive ($O(N)$) and *DeleteMin*s cheap ($O(1)$). The former is probably the better idea of the two, based on the fact that there are never more *DeleteMin*s than insertions.

Another way of implementing priority queues would be to use a binary search tree. This gives an $O(\log N)$ average running time for both operations. This is true in spite of the fact that although the insertions are random, the deletions are not. Recall that the only element we ever delete is the minimum. Repeatedly removing a node that is in the left subtree would seem to hurt the balance of the tree by making the right subtree heavy. However, the right subtree is random. In the worst case, where the *DeleteMin*s have depleted the left subtree, the right subtree would have at most twice as many elements as it should. This adds only a small constant to its expected depth. Notice that the bound can be made into a worst-case bound by using a balanced tree; this protects one against bad insertion sequences.

Using a search tree could be overkill because it supports a host of operations that are not required. The basic data structure we will use will not require pointers and will support both operations in $O(\log N)$ worst-case time. Insertion will actually take constant time on average, and our implementation will allow building a priority queue of N items in linear time, if no deletions intervene. We will then discuss how to implement priority queues to support efficient merging. This additional operation seems to complicate matters a bit and apparently requires the use of pointers.

6.3. Binary Heap

The implementation we will use is known as a *binary heap*. Its use is so common for priority queue implementations that when the word *heap* is used without a qualifier, it is generally assumed to be referring to this implementation of the data structure. In this section, we will refer to binary heaps merely as *heaps*. Like binary search trees, heaps have two properties, namely, a structure property and a heap order property. As with AVL trees, an operation on a heap can destroy one of the properties, so a heap operation must not terminate until all heap properties are in order. This turns out to be simple to do.

6.3.1. Structure Property

A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a *complete binary tree*. Figure 6.2 shows an example.

It is easy to show that a complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes. This implies that the height of a complete binary tree is $\lfloor \log N \rfloor$, which is clearly $O(\log N)$.

An important observation is that because a complete binary tree is so regular, it can be represented in an array and no pointers are necessary. The array in Figure 6.3 corresponds to the heap in Figure 6.2.

For any element in array position i, the left child is in position $2i$, the right child is in the cell after the left child ($2i + 1$), and the parent is in position $\lfloor i/2 \rfloor$. Thus not only are pointers not required, but the operations required to traverse the tree are

Figure 6.2 A complete binary tree

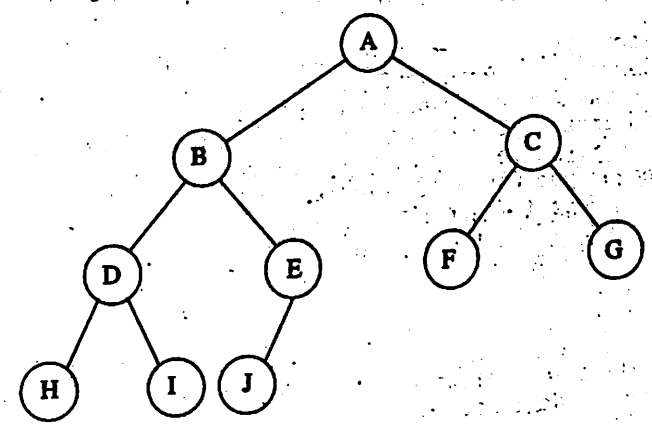


Figure 6.3 Array implementation of complete binary tree

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

extremely simple and likely to be very fast on most computers. The only problem with this implementation is that an estimate of the maximum heap size is required in advance, but typically this is not a problem. In Figure 6.3, the limit on the heap size is 13 elements. The array has a position 0; more on this later.

A heap data structure will, then, consist of an array (of whatever type the key is) and an integer representing the maximum and current heap sizes. Figure 6.4 shows a typical priority queue declaration. Notice the similarity to the stack declaration in Figure 3.47. Figure 6.4a creates an empty heap. Line 11 will be explained later.

Throughout this chapter, we shall draw the heaps as trees, with the implication that an actual implementation will use simple arrays.

6.3.2. *Heap Order Property*

The property that allows operations to be performed quickly is the *heap order* property. Since we want to be able to find the minimum quickly, it makes sense that the smallest element should be at the root. If we consider that any subtree should also be a heap, then any node should be smaller than all of its descendants.

Applying this logic, we arrive at the heap order property. In a heap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with

Figure 6.4 Declaration for priority queue

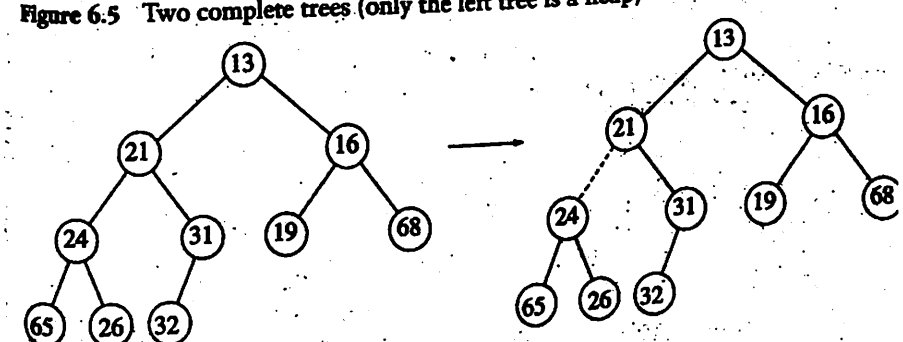
```
#ifndef _BinHeap_H

struct HeapStruct;
typedef struct HeapStruct *PriorityQueue;

PriorityQueue Initialize( int MaxElements );
void Destroy( PriorityQueue H );
void MakeEmpty( PriorityQueue H );
void Insert( ElementType X, PriorityQueue H );
ElementType DeleteMin( PriorityQueue H );
ElementType FindMin( PriorityQueue H );
int IsEmpty( PriorityQueue H );
int IsFull( PriorityQueue H );

#endif


/* Place in implementation file */
struct HeapStruct
{
    int Capacity;
    int Size;
    ElementType *Elements;
};
```


the exception of the root (which has no parent).* In Figure 6.5 the tree on the left is a heap, but the tree on the right is not (the dashed line shows the violation of heap order). As usual, we will assume that the keys are integers, although they could be arbitrarily complex.

Figure 6.4 Declaration for priority queue

```
        PriorityQueue
        Initialize( int MaxElements )
        {
            PriorityQueue H;

/* 1*/      if( MaxElements < MinPQSize )
/* 2*/          Error( "Priority queue size is too small" );

/* 3*/      H = malloc( sizeof( struct HeapStruct ) );
/* 4*/      if( H == NULL )
/* 5*/          FatalError( "Out of space!!!" );

            /* Allocate the array plus one extra for sentinel */
/* 6*/      H->Elements = malloc( ( MaxElements + 1 )
                                  * sizeof( ElementType ) );
/* 7*/      if( H->Elements == NULL )
/* 8*/          FatalError( "Out of space!!!" );

/* 9*/      H->Capacity = MaxElements;
/*10*/      H->Size = 0;
/*11*/      H->Elements[ 0 ] = MinData;

/*12*/      return H;
        }
```

Figure 6.5 Two complete trees (only the left tree is a heap)



*Analogously, we can declare a (*max*) heap, which enables us to efficiently find and remove the maximum element, by changing the heap order property. Thus, a priority queue can be used to find *either* a minimum or a maximum, but this needs to be decided ahead of time.

By the heap order property, the minimum element can always be found at the root. Thus, we get the extra operation, *FindMin*, in constant time.

6.3.3. Basic Heap Operations

It is easy (both conceptually and practically) to perform the two required operations. All the work involves ensuring that the heap order property is maintained.

Insert

To insert an element X into the heap, we create a hole in the next available location, since otherwise the tree will not be complete. If X can be placed in the hole without violating heap order, then we do so and are done. Otherwise we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. We continue this process until X can be placed in the hole. Figure 6.6 shows that to insert 14, we create a hole in the next available heap location. Inserting 14 in the hole would violate the heap order property, so 31 is slid down into the hole. This strategy is continued in Figure 6.7 until the correct location for 14 is found.

Figure 6.6 Attempt to insert 14: creating the hole, and bubbling the hole up

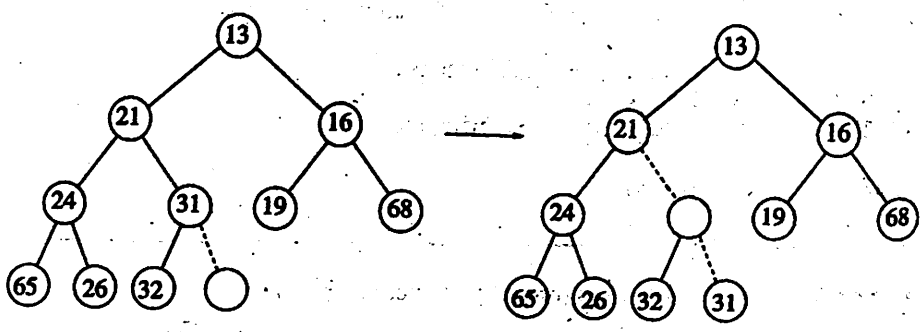
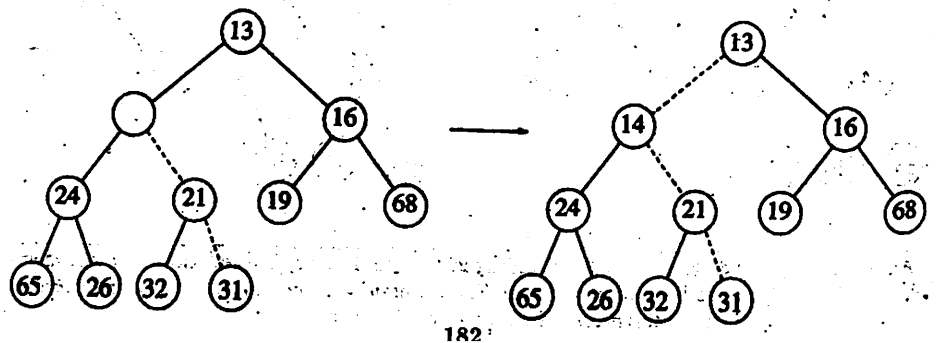


Figure 6.7 The remaining two steps to insert 14 in previous heap




```
/* H->Element[ 0 ] is a sentinel */

void
Insert( ElementType X, PriorityQueue H )
{
    int i;

    if( IsFull( H ) )
    {
        Error( "Priority queue is full" );
        return;
    }

    for( i = ++H->Size; H->Elements[ i / 2 ] > X; i /= 2 )
        H->Elements[ i ] = H->Elements[ i / 2 ];
    H->Elements[ i ] = X;
}
```

Figure 6.8 Procedure to insert into a binary key

This general strategy is known as a *percolate up*; the new element is percolated up the heap until the correct location is found. Insertion is easily implemented with the code shown in Figure 6.8.

We could have implemented the percolation in the *Insert* routine by performing repeated swaps until the correct order was established, but a swap requires three assignment statements. If an element is percolated up d levels, the number of assignments performed by the swaps would be $3d$. Our method uses $d + 1$ assignments.

If the element to be inserted is the new minimum, it will be pushed all the way to the top. At some point, i will be 1 and we will want to break out of the *while* loop. We could do this with an explicit test, but we have chosen to put a very small value in position 0 in order to make the *while* loop terminate. This value must be guaranteed to be smaller than (or equal to) any element in the heap; it is known as a *sentinel*. This idea is similar to the use of header nodes in linked lists. By adding a dummy piece of information, we avoid a test that is executed once per loop iteration, thus saving some time.

The time to do the insertion could be as much as $O(\log N)$, if the element to be inserted is the new minimum and is percolated all the way to the root. On average, the percolation terminates early; it has been shown that 2.607 comparisons are required on average to perform an insert, so the average *Insert* moves an element up 1.607 levels.

DeleteMin

DeleteMins are handled in a similar manner as insertions. Finding the minimum is easy; the hard part is removing it. When the minimum is removed, a hole is created at the root. Since the heap now becomes one smaller, it follows that the last element X in the heap must move somewhere in the heap. If X can be placed in the hole,

then we are done. This is unlikely, so we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. We repeat this step until X can be placed in the hole. Thus, our action is to place X in its correct spot along a path from the root containing *minimum* children.

In Figure 6.9 the left figure shows a heap prior to the *DeleteMin*. After 13 is removed, we must now try to place 31 in the heap. The value 31 cannot be placed in the hole, because this would violate heap order. Thus, we place the smaller child (14) in the hole, sliding the hole down one level (see Fig. 6.10). We repeat this again, placing 19 into the hole and creating a new hole one level deeper. We then place

Figure 6.9 Creation of the hole at the root

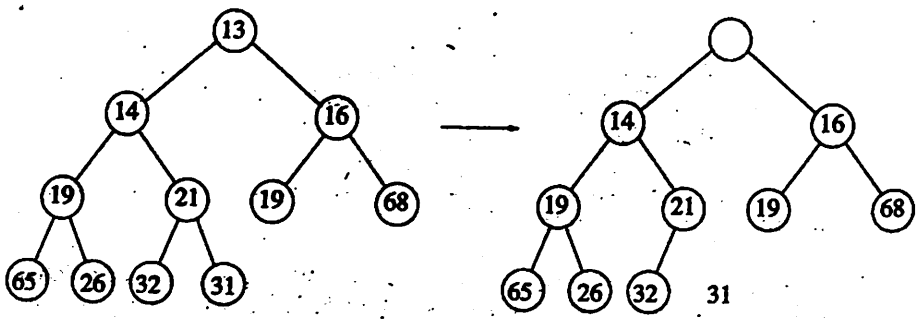


Figure 6.10 Next two steps in *DeleteMin*

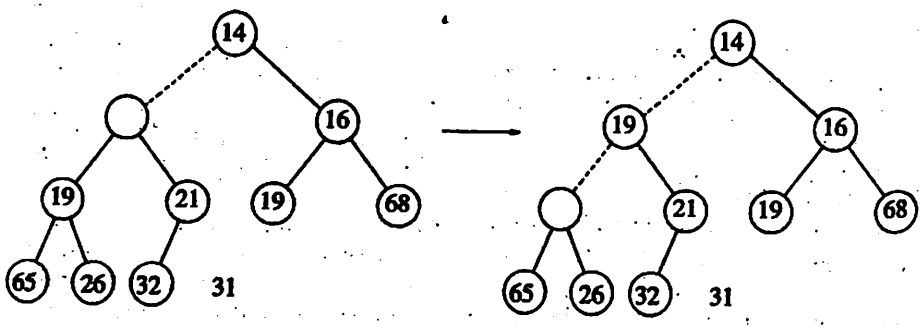
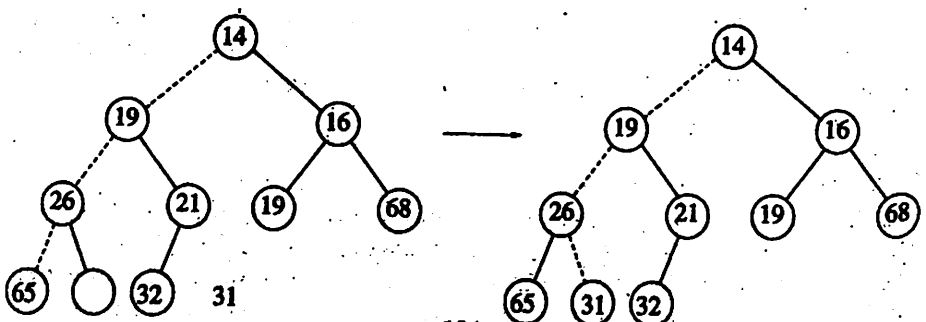


Figure 6.11 Last two steps in *DeleteMin*



26 in the hole and create a new hole on the bottom level. Finally, we are able to place 31 in the hole (Fig. 6.11). This general strategy is known as a *percolate down*. We use the same technique as in the *Insert* routine to avoid the use of swaps in this routine.

A frequent implementation error in heaps occurs when there are an even number of elements in the heap, and the one node that has only one child is encountered. You must make sure not to assume that there are always two children, so this usually involves an extra test. In the code depicted in Figure 6.12, we've done this test at line 8. One extremely tricky solution is always to ensure that your algorithm *thinks* every node has two children. Do this by placing a sentinel, of value higher than any in the heap, at the spot after the heap ends, at the start of each *percolate down* when the heap size is even. You should think very carefully before attempting this, and you must put in a prominent comment if you do use this technique. Although this

Figure 6.12 Function to perform *DeleteMin* in a binary heap

```
        ElementType
        DeleteMin( PriorityQueue H )
        {
            int i, Child;
            ElementType MinElement, LastElement;

/* 1*/      if( IsEmpty( H ) )
            {
/* 2*/          Error( "Priority queue is empty" );
/* 3*/          return H->Elements[ 0 ];
            }
/* 4*/      MinElement = H->Elements[ 1 ];
/* 5*/      LastElement = H->Elements[ H->Size-- ];

/* 6*/      for( i = 1; i * 2 <= H->Size; i = Child )
            {
                /* Find smaller child */
/* 7*/          Child = i * 2;
/* 8*/          if( Child != H->Size && H->Elements[ Child + 1 ]
/* 9*/                              < H->Elements[ Child ] )
/*10*/              Child++;

                /* Percolate one level */
/*11*/          if( LastElement > H->Elements[ Child ] )
/*12*/              H->Elements[ i ] = H->Elements[ Child ];
                else
/*13*/              break;
            }
/*14*/      H->Elements[ i ] = LastElement;
/*15*/      return MinElement;
        }
```

eliminates the need to test for the presence of a right child, you cannot eliminate the requirement that you test when you reach the bottom, because this would require a sentinel for every leaf.

The worst-case running time for this operation is $O(\log N)$. On average, the element that is placed at the root is percolated almost to the bottom of the heap (which is the level it came from), so the average running time is $O(\log N)$.

6.3.4. Other Heap Operations

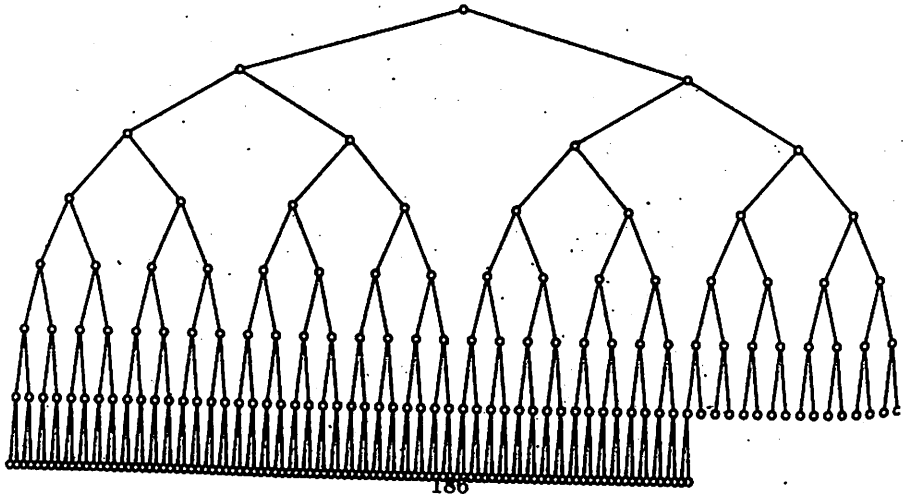
Notice that although finding the minimum can be performed in constant time, a heap designed to find the minimum element (also known as a (*min*)heap) is of no help whatsoever in finding the maximum element. In fact, a heap has very little ordering information, so there is no way to find any particular key without a linear scan through the entire heap. To see this, consider the large heap structure (the elements are not shown) in Figure 6.13, where we see that the only information known about the maximum element is that it is at one of the leaves. Half the elements, though, are contained in leaves, so this is practically useless information. For this reason, if it is important to know where elements are, some other data structure, such as a hash table, must be used in addition to the heap. (Recall that the model does not allow looking inside the heap.)

If we assume that the position of every element is known by some other method, then several other operations become cheap. The three operations below all run in logarithmic worst-case time.

DecreaseKey

The *DecreaseKey*(P, Δ, H) operation lowers the value of the key at position P by a positive amount Δ. Since this might violate the heap order, it must be fixed by a *percolate up*. This operation could be useful to system administrators: They can make their programs run with highest priority.

Figure 6.13 A very large complete binary tree



IncreaseKey

The *IncreaseKey*(P, Δ, H) operation increases the value of the key at position P by a positive amount Δ. This is done with a *percolate down*. Many schedulers automatically drop the priority of a process that is consuming excessive CPU time.

Delete

The *Delete*(P, H) operation removes the node at position P from the heap. This is done by first performing *DecreaseKey*(P, ∞, H) and then performing *DeleteMin*(H). When a process is terminated by a user (instead of finishing normally), it must be removed from the priority queue.

BuildHeap

The *BuildHeap*(H) operation takes as input N keys and places them into an empty heap. Obviously, this can be done with N successive *Inserts*. Since each *Insert* will take $O(1)$ average and $O(\log N)$ worst-case time, the total running time of this algorithm would be $O(N)$ average but $O(N \log N)$ worst-case. Since this is a special instruction and there are no other operations intervening, and we already know that the instruction can be performed in linear average time, it is reasonable to expect that with reasonable care a linear time bound can be guaranteed.

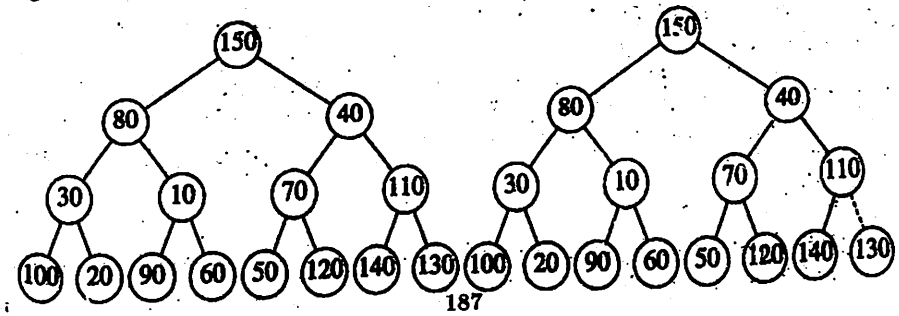
The general algorithm is to place the N keys into the tree in any order, maintaining the structure property. Then, if *PercolateDown*(i) percolates down from node i, perform the algorithm in Figure 6.14 to create a heap-ordered tree.

The first tree in Figure 6.15 is the unordered tree. The seven remaining trees in Figures 6.15 through 6.18 show the result of each of the seven *PercolateDowns*. Each dashed line corresponds to two comparisons: one to find the smaller child and one to compare the smaller child with the node. Notice that there are only 10 dashed

Figure 6.14 Sketch of *BuildHeap*

```
for( i = N / 2; i > 0; i-- )
    PercolateDown( i );
```

Figure 6.15 Left: initial heap; right: after *PercolateDown*(7)



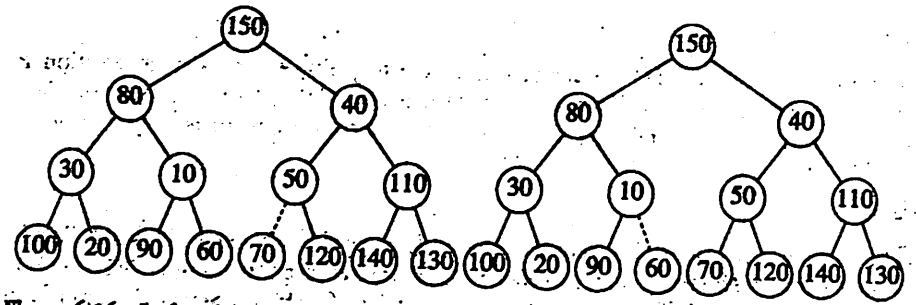


Figure 6.16 Left: after *PercolateDown*(6); right: after *PercolateDown*(5)

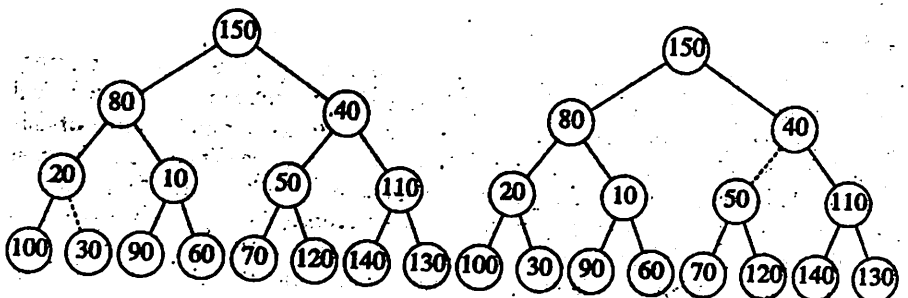


Figure 6.17 Left: after *PercolateDown*(4); right: after *PercolateDown*(3)

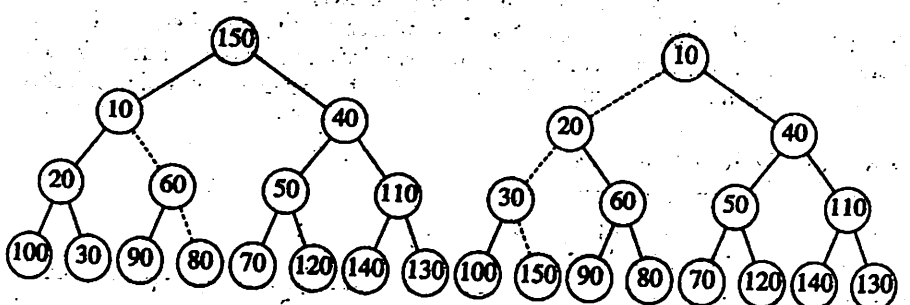


Figure 6.18 Left: after *PercolateDown*(2); right: after *PercolateDown*(1)

lines in the entire algorithm (there could have been an 11th—where?) corresponding to 20 comparisons.

To bound the running time of *BuildHeap*, we must bound the number of dashed lines. This can be done by computing the sum of the heights of all the nodes in the heap, which is the maximum number of dashed lines. What we would like to show is that this sum is $O(N)$.

THEOREM 6.1.

For the perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $2^{h+1} - 1 - (h + 1)$.

PROOF:

It is easy to see that this tree consists of 1 node at height h, 2 nodes at height $h - 1$, 2^2 nodes at height $h - 2$, and in general 2^i nodes at height $h - i$. The sum of the heights of all the nodes is then

$$S = \sum_{i=0}^{h} 2^i (h - i)$$

$$= h + 2(h - 1) + 4(h - 2) + 8(h - 3) + 16(h - 4) + \cdots + 2^{h-1}(1) \quad (6.1)$$

Multiplying by 2 gives the equation

$$2S = 2h + 4(h - 1) + 8(h - 2) + 16(h - 3) + \cdots + 2^h(1) \quad (6.2)$$

We subtract these two equations and obtain Equation (6.3). We find that certain terms almost cancel. For instance, we have $2h - 2(h - 1) = 2$, $4(h - 1) - 4(h - 2) = 4$, and so on. The last term in Equation (6.2), 2^h, does not appear in Equation (6.1); thus, it appears in Equation (6.3). The first term in Equation (6.1), h, does not appear in Equation (6.2); thus, $-h$ appears in Equation (6.3). We obtain

$$S = -h + 2 + 4 + 8 + \cdots + 2^{h-1} + 2^h = (2^{h+1} - 1) - (h + 1) \quad (6.3)$$

which proves the theorem.

A complete tree is not a perfect binary tree, but the result we have obtained is an upper bound on the the sum of the heights of the nodes in a complete tree. Since a complete tree has between 2^h and 2^{h+1} nodes, this theorem implies that this sum is $O(N)$, where N is the number of nodes.

Although the result we have obtained is sufficient to show that *BuildHeap* is linear, the bound on the sum of the heights is not as strong as possible. For a complete tree with $N = 2^h$ nodes, the bound we have obtained is roughly $2N$. The sum of the heights can be shown by induction to be $N - b(N)$, where $b(N)$ is the number of 1s in the binary representation of N.

6.4. Applications of Priority Queues

We have already mentioned how priority queues are used in operating systems design. In Chapter 9, we will see how priority queues are used to implement several graph algorithms efficiently. Here we will show how to use priority queues to obtain solutions to two problems.

6.4.1. *The Selection Problem*

The first problem we will examine is the *selection problem* from Chapter 1. Recall that the input is a list of N elements, which can be totally ordered, and an integer k. The selection problem is to find the kth largest element.

Two algorithms were given in Chapter 1, but neither is very efficient. The first algorithm, which we shall call algorithm 1A, is to read the elements into an array and sort them, returning the appropriate element. Assuming a simple sorting algorithm, the running time is $O(N^2)$. The alternative algorithm, 1B, is to read k elements into an array and sort them. The smallest of these is in the kth position. We process the remaining elements one by one. As an element arrives, it is compared with the kth element in the array. If it is larger, then the kth element is removed, and the new element is placed in the correct place among the remaining $k-1$ elements. When the algorithm ends, the element in the kth position is the answer. The running time is $O(N \cdot k)$ (why?). If $k = \lceil N/2 \rceil$, then both algorithms are $O(N^2)$. Notice that for any k, we can solve the symmetric problem of finding the $(N-k+1)$th smallest element, so $k = \lceil N/2 \rceil$ is really the hardest case for these algorithms: This also happens to be the most interesting case, since this value of k is known as the *median*.

We give two algorithms here, both of which run in $O(N \log N)$ in the extreme case of $k = \lceil N/2 \rceil$, which is a distinct improvement.

Algorithm 6A

For simplicity, we assume that we are interested in finding the kth *smallest* element. The algorithm is simple. We read the N elements into an array. We then apply the *BuildHeap* algorithm to this array. Finally, we perform k *DeleteMin* operations. The last element extracted from the heap is our answer. It should be clear that by changing the heap order property, we could solve the original problem of finding the kth *largest* element.

The correctness of the algorithm should be clear. The worst-case timing is $O(N)$ to construct the heap, if *BuildHeap* is used, and $O(\log N)$ for each *DeleteMin*. Since there are k *DeleteMins*, we obtain a total running time of $O(N + k \log N)$. If $k = O(N/\log N)$, then the running time is dominated by the *BuildHeap* operation and is $O(N)$. For larger values of k, the running time is $O(k \log N)$. If $k = \lceil N/2 \rceil$, then the running time is $\Theta(N \log N)$.

Notice that if we run this program for $k = N$ and record the values as they leave the heap, we will have essentially sorted the input file in $O(N \log N)$ time. In Chapter 7, we will refine this idea to obtain a fast sorting algorithm known as *heapsort*.

Algorithm 6B

For the second algorithm, we return to the original problem and find the kth *largest* element. We use the idea from algorithm 1B. At any point in time we will maintain a set S of the k largest elements. After the first k elements are read, when a new element is read it is compared with the kth largest element, which we denote by S_k. Notice that S_k is the smallest element in S. If the new element is larger, then it replaces S_k in S. S will then have a new smallest element, which may or may not be the newly added element. At the end of the input, we find the smallest element in S and return it as the answer.

This is essentially the same algorithm described in Chapter 1. Here, however, we will use a heap to implement S. The first k elements are placed into the heap in total time $O(k)$ with a call to *BuildHeap*. The time to process each of the remaining elements is $O(1)$, to test if the element goes into S, plus $O(\log k)$,

to delete S_k and insert the new element if this is necessary. Thus, the total time is $O(k + (N-k)\log k) = O(N \log k)$. This algorithm also gives a bound of $\Theta(N \log N)$ for finding the median.

In Chapter 7, we will see how to solve this problem in $O(N)$ average time. In Chapter 10, we will see an elegant, albeit impractical, algorithm to solve this problem in $O(N)$ worst-case time.

6.4.2. *Event Simulation*

In Section 3.4.3, we described an important queuing problem. Recall that we have a system, such as a bank, where customers arrive and wait on a line until one of k tellers is available. Customer arrival is governed by a probability distribution function, as is the service time (the amount of time to be served once a teller is available). We are interested in statistics such as how long on average a customer has to wait or how long the line might be.

With certain probability distributions and values of k, these answers can be computed exactly. However, as k gets larger, the analysis becomes considerably more difficult, so it is appealing to use a computer to simulate the operation of the bank. In this way, the bank officers can determine how many tellers are needed to ensure reasonably smooth service.

A simulation consists of processing events. The two events here are (a) a customer arriving and (b) a customer departing, thus freeing up a teller.

We can use the probability functions to generate an input stream consisting of ordered pairs of arrival time and service time for each customer, sorted by arrival time. We do not need to use the exact time of day. Rather, we can use a quantum unit, which we will refer to as a *tick*.

One way to do this simulation is to start a simulation clock at zero ticks. We then advance the clock one tick at a time, checking to see if there is an event. If there is, then we process the event(s) and compile statistics. When there are no customers left in the input stream and all the tellers are free, then the simulation is over.

The problem with this simulation strategy is that its running time does not depend on the number of customers or events (there are two events per customer), but instead depends on the number of ticks, which is not really part of the input. To see why this is important, suppose we changed the clock units to milliticks and multiplied all the times in the input by 1,000. The result would be that the simulation would take 1,000 times longer!

The key to avoiding this problem is to advance the clock to the next event time at each stage. This is conceptually easy to do. At any point, the next event that can occur is either (a) the next customer in the input file arrives or (b) one of the customers at a teller leaves. Since all the times when the events will happen are available, we just need to find the event that happens nearest in the future and process that event.

If the event is a departure, processing includes gathering statistics for the departing customer and checking the line (queue) to see whether there is another customer waiting. If so, we add that customer, process whatever statistics are required, compute the time when that customer will leave, and add that departure to the set of events waiting to happen.

If the event is an arrival, we check for an available teller. If there is none, we place the arrival on the line (queue); otherwise we give the customer a teller, compute the customer's departure time, and add the departure to the set of events waiting to happen.

The waiting line for customers can be implemented as a queue. Since we need to find the event *nearest* in the future, it is appropriate that the set of departures waiting to happen be organized in a priority queue. The next event is thus the next arrival or next departure (whichever is sooner); both are easily available.

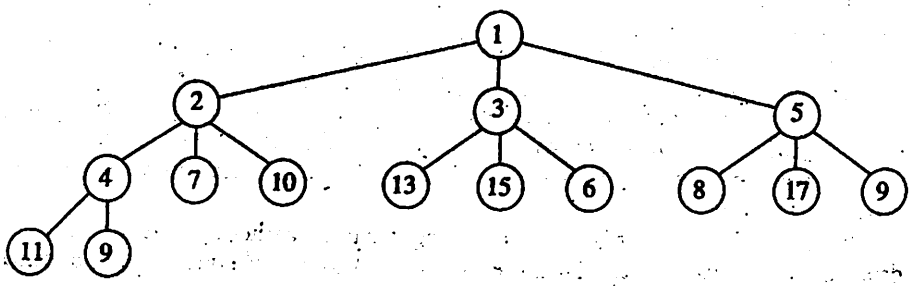
It is then straightforward, although possibly time-consuming, to write the simulation routines. If there are C customers (and thus $2C$ events) and k tellers, then the running time *of the simulation* would be $O(C \log(k+1))$* because computing and processing each event takes $O(\log H)$, where $H = k+1$ is the size of the heap.

6.5. *d*-Heaps

Binary heaps are so simple that they are almost always used when priority queues are needed. A simple generalization is a *d-heap*, which is exactly like a binary heap except that all nodes have d children (thus, a binary heap is a 2-heap).

Figure 6.19 shows a 3-heap. Notice that a d-heap is much shallower than a binary heap, improving the running time of *Inserts* to $O(\log_d N)$. However, for large d, the *DeleteMin* operation is more expensive, because even though the tree is shallower, the minimum of d children must be found, which takes $d-1$ comparisons using a standard algorithm. This raises the time for this operation to $O(d \log_d N)$. If d is a constant, both running times are, of course, $O(\log N)$. Although an array can still be used, the multiplications and divisions to find children and parents are now by d, which, unless d is a power of 2, seriously increases the running time, because we can no longer implement division by a bit shift. d-heaps are interesting in theory, because there are many algorithms where the number of insertions is much greater than the number of *DeleteMins* (and thus a theoretical speedup is possible). They are also of interest when the priority queue is too large to fit entirely in main memory. In this case, a d-heap can be advantageous in much the same way as B-trees. Finally, there is evidence suggesting that 4-heaps may outperform binary heaps in practice.

Figure 6.19 A d-heap



*We use $O(C \log(k+1))$ instead of $O(C \log k)$ to avoid confusion for the $k = 1$ case.

The most glaring weakness of the heap implementation, aside from the inability to perform *Finds*, is that combining two heaps into one is a hard operation. This extra operation is known as a *Merge*. There are quite a few ways of implementing heaps so that the running time of a *Merge* is $O(\log N)$. We will now discuss three data structures, of various complexity, that support the *Merge* operation efficiently. We will defer any complicated analysis until Chapter 11.

6.6. Leftist Heaps

It seems difficult to design a data structure that efficiently supports merging (that is, processes a *Merge* in $O(N)$ time) and uses only an array, as in a binary heap. The reason for this is that merging would seem to require copying one array into another, which would take $\Theta(N)$ time for equal-sized heaps. For this reason, all the advanced data structures that support efficient merging require the use of pointers. In practice, we can expect that this will make all the other operations slower, pointer manipulation is generally more time-consuming than multiplication and division by 2.

Like a binary heap, a *leftist heap* has both a structural property and an ordering property. Indeed, a leftist heap, like virtually all heaps used, has the same heap order property we have already seen. Furthermore, a leftist heap is also a binary tree. The only difference between a leftist heap and a binary heap is that leftist heaps are not perfectly balanced, but actually attempt to be very unbalanced.

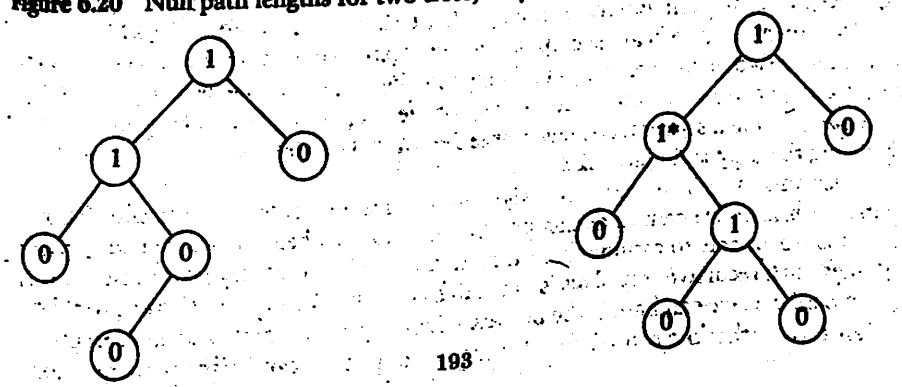
6.6.1. *Leftist Heap Property*

We define the *null path length*, $Npl(X)$, of any node X to be the length of the shortest path from X to a node without two children. Thus, the Npl of a node with zero or one child is 0, while $Npl(NULL) = -1$. In the tree in Figure 6.20, the null path lengths are indicated inside the tree nodes.

Notice that the null path length of any node is 1 more than the minimum of the null path lengths of its children. This applies to nodes with less than two children because the null path length of $NULL$ is -1.

The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child. This property is

Figure 6.20 Null path lengths for two trees; only the left tree is leftist



satisfied by only one of the trees in Figure 6.20, namely, the tree on the left. This property actually goes out of its way to ensure that the tree is unbalanced, because it clearly biases the tree to get deep toward the left. Indeed, a tree consisting of a long path of left nodes is possible (and actually preferable to facilitate merging)—hence the name *leftist heap*.

Because leftist heaps tend to have deep left paths, it follows that the right path ought to be short. Indeed, the right path down a leftist heap is as short as any in the heap. Otherwise, there would be a path that goes through some node X and takes the left child. Then X would violate the leftist property.

THEOREM 6.2:

A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.

PROOF:

The proof is by induction. If $r = 1$, there must be at least one tree node. Otherwise, suppose that the theorem is true for $1, 2, \ldots, r$. Consider a leftist tree with $r + 1$ nodes on the right path. Then the root has a right subtree with r nodes on the right path, and a left subtree with at least r nodes on the right path (otherwise it would not be leftist). Applying the inductive hypothesis to these subtrees yields a minimum of $2^r - 1$ nodes in each subtree. This plus the root gives at least $2^{r+1} - 1$ nodes in the tree, proving the theorem.

From this theorem, it follows immediately that a leftist tree of N nodes has a right path containing at most $\lfloor \log(N + 1) \rfloor$ nodes. The general idea for the leftist heap operations is to perform all the work on the right path, which is guaranteed to be short. The only tricky part is that performing *Inserts* and *Merges* on the right path could destroy the leftist heap property. It turns out to be extremely easy to restore the property.

6.6.2. Leftist Heap Operations

The fundamental operation on leftist heaps is merging. Notice that insertion is merely a special case of merging, since we may view an insertion as a *Merge* of a one-node heap with a larger heap. We will first give a simple recursive solution and then show how this might be done nonrecursively. Our input is the two leftist heaps, H_1 and H_2, in Figure 6.21. You should check that these heaps really are leftist. Notice that the smallest elements are at the roots. In addition to space for the data and left and right pointers, each cell will have an entry that indicates the null path length.

If either of the two heaps is empty, then we can return the other heap. Otherwise, to merge the two heaps, we compare their roots. First, we recursively merge the heap with the larger root with the right subheap of the heap with the smaller root. In our example, this means we recursively merge H_2 with the subheap of H_1 rooted at 8, obtaining the heap in Figure 6.22.

Since this tree is formed recursively, and we have not yet finished the description of the algorithm, we cannot at this point show how this heap was obtained. However, it is reasonable to assume that the resulting tree is a leftist heap, because it was obtained via a recursive step. This is much like the inductive hypothesis in a proof by induction. Since we can handle the base case (which occurs when one tree is empty), we can assume that the recursive step works as long as we can finish the merge; this

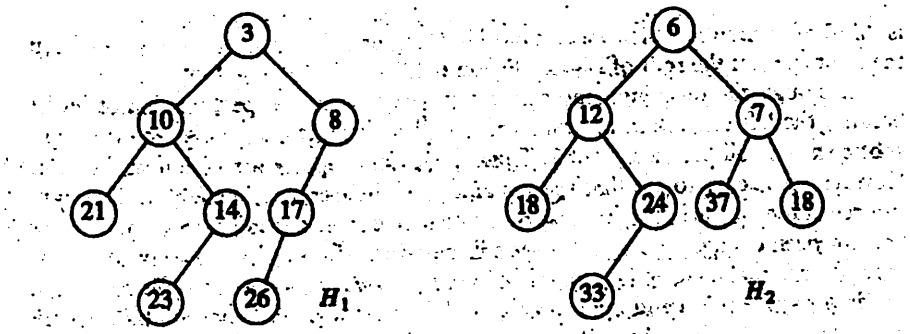


Figure 6.21 Two leftist heaps H_1 and H_2

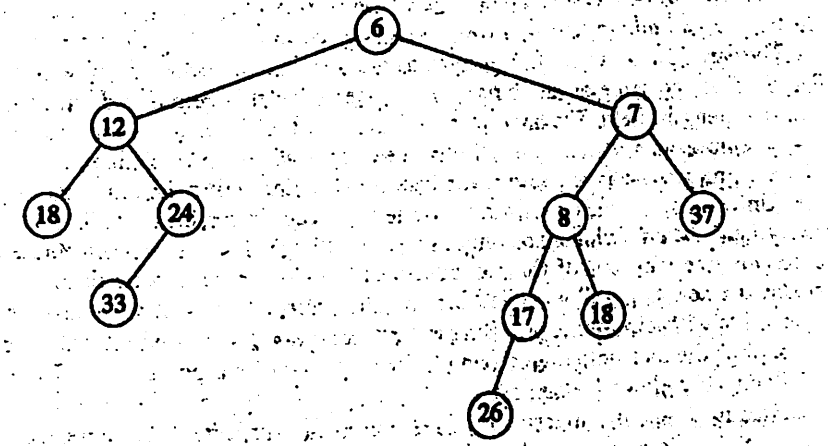


Figure 6.22 Result of merging H_2 with H_1's right subheap

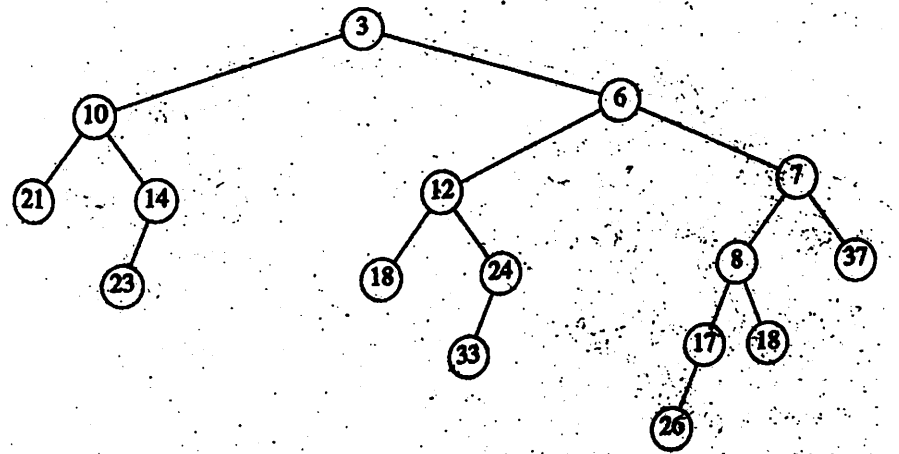


Figure 6.23 Result of attaching leftist heap of previous figure as H_1's right child

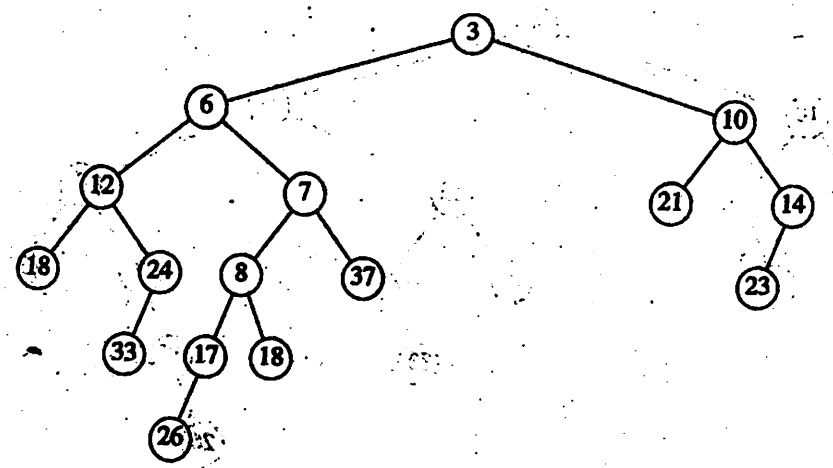
is rule 3 of recursion, which we discussed in Chapter 1. We now make this new heap the right child of the root of H_1 (see Figure 6.23).

Although the resulting heap satisfies the heap order property, it is not leftist because the left subtree of the root has a null path length of 1 whereas the right subtree has a null path length of 2. Thus, the leftist property is violated at the root. However, it is easy to see that the remainder of the tree must be leftist. The right subtree of the root is leftist, because of the recursive step. The left subtree of the root has not been changed, so it too must still be leftist. Thus, we need only to fix the root. We can make the entire tree leftist by merely swapping the root's left and right children (Figure 6.24) and updating the null path length—the new null path length is 1 plus the null path length of the new right child—completing the *Merge*. Notice that if the null path length is not updated, then all null path lengths will be 0, and the heap will not be leftist but merely random. In this case, the algorithm will work, but the time bound we will claim will no longer be valid.

The description of the algorithm translates directly into code. The type definition (Fig. 6.25) is the same as the binary tree, except that it is augmented with the *Npl* (null path length) field. We have seen in Chapter 4 that when an element is inserted into an empty binary tree, the pointer to the root will need to change. The easiest way to implement this is to have the insertion routine return a pointer to the new tree. Unfortunately, this will make the leftist heap *Insert* incompatible with the binary heap *Insert* (which does not return anything). The last line in Figure 6.25 represents one way out of this quandary. The leftist heap insertion routine which returns the new tree will be called *Insert1*; the *Insert* macro will make an insertion compatible with binary heaps. Using macros this way may not be the best or safest course, but the alternative, declaring a *PriorityQueue* as a pointer to a *TreeNode*, will flood the code with extra asterisks.*

Because *Insert* is a macro and is textually substituted by the preprocessor, any routine that calls *Insert* must be able to see the macro definition. Figure 6.25 would

Figure 6.24 Result of swapping children of H_1's root



*Another possibility is to accept the incompatible interfaces as a necessary evil.


```
#ifndef _LeftHeap_H

struct TreeNode;
typedef struct TreeNode *PriorityQueue;

/* Minimal set of priority queue operations */
/* Note that nodes will be shared among several */
/* leftist heaps after a merge; the user must */
/* make sure to not use the old leftist heaps. */

PriorityQueue Initialize( void );
ElementType FindMin( PriorityQueue H );
int IsEmpty( PriorityQueue H );
PriorityQueue Merge( PriorityQueue H1, PriorityQueue H2 );

#define Insert( X, H ) ( H = Insert1( ( X ), H ) )
/* DeleteMin macro is left as an exercise */

PriorityQueue Insert1( ElementType X, PriorityQueue H );
PriorityQueue DeleteMin1( PriorityQueue H );

#endif


/* Place in implementation file */
struct TreeNode
{
    ElementType   Element;
    PriorityQueue Left;
    PriorityQueue Right;
    int           Npl;
};
```

Figure 6.25 Leftist heap type declarations

typically be a header file, so placing the macro declaration there is the only reasonable course. As we will see later, *DeleteMin* also needs to be written as a macro.

The routine to merge (Fig. 6.26) is a driver designed to remove special cases and ensure that H_1 has the smaller root. The actual merging is performed in *Merge1* (Fig. 6.27). Note that the original leftist heaps should never be used again; changes in them will affect the merged result.

The time to perform the merge is proportional to the sum of the length of the right paths, because constant work is performed at each node visited during the recursive calls. Thus we obtain an $O(\log N)$ time bound to merge two leftist heaps. We can also perform this operation nonrecursively by essentially performing two passes. In the first pass, we create a new tree by merging the right paths of both heaps. To do this, we arrange the nodes on the right paths of H_1 and H_2 in sorted order, keeping their respective left children. In our example, the new right path is

```
        PriorityQueue
        Merge( PriorityQueue H1, PriorityQueue H2 )
        {
/* 1*/      if( H1 == NULL )
/* 2*/          return H2;
/* 3*/      if( H2 == NULL )
/* 4*/          return H1;
/* 5*/      if( H1->Element < H2->Element )
/* 6*/          return Merge1( H1, H2 );
            else
/* 7*/          return Merge1( H2, H1 );
        }
```

Figure 6.26 Driving routine for merging leftist heaps

```
        static PriorityQueue
        Merge1( PriorityQueue H1, PriorityQueue H2 )
        {
/* 1*/      if( H1->Left == NULL )  /* Single node */
/* 2*/          H1->Left = H2;      /* H1->Right is already NULL,
                                       H1->Npl is already 0 */
            else
            {
/* 3*/          H1->Right = Merge( H1->Right, H2 );
/* 4*/          if( H1->Left->Npl < H1->Right->Npl )
/* 5*/              SwapChildren( H1 );

/* 6*/          H1->Npl = H1->Right->Npl + 1;
            }
/* 7*/      return H1;
        }
```

Figure 6.27 Actual routine to merge leftist heaps

3, 6, 7, 8, 18 and the resulting tree is shown in Figure 6.28. A second pass is made up the heap, and child swaps are performed at nodes that violate the leftist heap property. In Figure 6.28, there is a swap at nodes 7 and 3, and the same tree as before is obtained. The nonrecursive version is simpler to visualize but harder to code. We leave it to the reader to show that the recursive and nonrecursive procedures do the same thing.

As mentioned above, we can carry out insertions by making the item to be inserted a one-node heap and performing a *Merge*. To perform a *DeleteMin*, we merely destroy the root, creating two heaps, which can then be merged. Thus, the time to perform a *DeleteMin* is $O(\log N)$. These two routines are coded in Figure 6.29 and Figure 6.30. *DeleteMin* can be written as a macro that calls *DeleteMin1* and *FindMin*. This is left as an exercise to the reader.

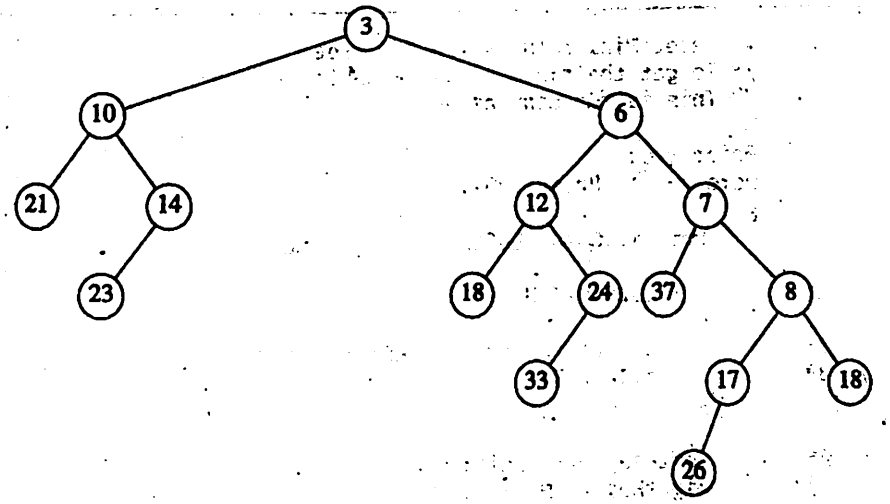


Figure 6.28 Result of merging right paths of H_1 and H_2

```
        PriorityQueue
        Insert1( ElementType X, PriorityQueue H )
        {
            PriorityQueue SingleNode;

/* 1*/      SingleNode = malloc( sizeof( struct TreeNode ) );
/* 2*/      if( SingleNode == NULL )
/* 3*/          FatalError( "Out of space!!!" );
            else
            {
/* 4*/          SingleNode->Element = X; SingleNode->Npl = 0;
/* 5*/          SingleNode->Left = SingleNode->Right = NULL;
/* 6*/          H = Merge( SingleNode, H );
            }
/* 7*/      return H;
        }
```

Figure 6.29 Insertion routine for leftist heaps

Finally, we can build a leftist heap in $O(N)$ time by building a binary heap (obviously using a pointer implementation). Although a binary heap is clearly leftist, this is not necessarily the best solution, because the heap we obtain is the worst possible leftist heap. Furthermore, traversing the tree in reverse-level order is not as easy with pointers. The *BuildHeap* effect can be obtained by recursively building the left and right subtrees and then percolating the root down. The exercises contain an alternative solution.

```
        /* DeleteMin1 returns the new tree; */
        /* To get the minimum, use FindMin */
        /* This is for convenience */

        PriorityQueue
        DeleteMin1( PriorityQueue H )
        {
            PriorityQueue LeftHeap, RightHeap;

/* 1*/      if( IsEmpty( H ) )
            {
/* 2*/          Error( "Priority queue is empty" );
/* 3*/          return H;
            }

/* 4*/      LeftHeap = H->Left;
/* 5*/      RightHeap = H->Right;
/* 6*/      free( H );
/* 7*/      return Merge( LeftHeap, RightHeap );
        }
```

Figure 6.30 *DeleteMin* routine for leftist heaps

6.7. Skew Heaps

A *skew heap* is a self-adjusting version of a leftist heap that is incredibly simple to implement. The relationship of skew heaps to leftist heaps is analogous to the relation between splay trees and AVL trees. Skew heaps are binary trees with heap order, but there is no structural constraint on these trees. Unlike leftist heaps, no information is maintained about the null path length of any node. The right path of a skew heap can be arbitrarily long at any time, so the worst-case running time of all operations is $O(N)$. However, as with splay trees, it can be shown (see Chapter 11) that for any M consecutive operations, the total worst-case running time is $O(M \log N)$. Thus, skew heaps have $O(\log N)$ amortized cost per operation.

As with leftist heaps, the fundamental operation on skew heaps is merging. The *Merge* routine is once again recursive, and we perform the exact same operations as before, with one exception. The difference is that for leftist heaps, we check to see whether the left and right children satisfy the leftist heap order property and swap them if they do not. For skew heaps, the swap is unconditional; we *always* do it, with the one exception that the largest of all the nodes on the right paths does not have its children swapped. This one exception is what happens in the natural recursive implementation, so it is not really a special case at all. Furthermore, it is not necessary to prove the bounds, but since this node is guaranteed not to have a right child, it would be silly to perform the swap and give it one. (In our example,

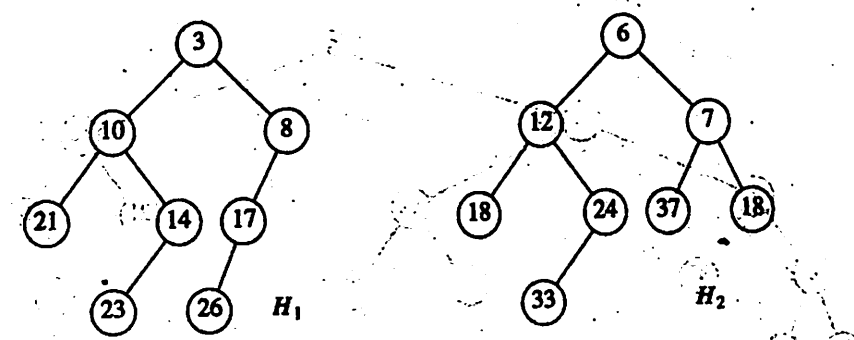


Figure 6.31 Two skew heaps H_1 and H_2

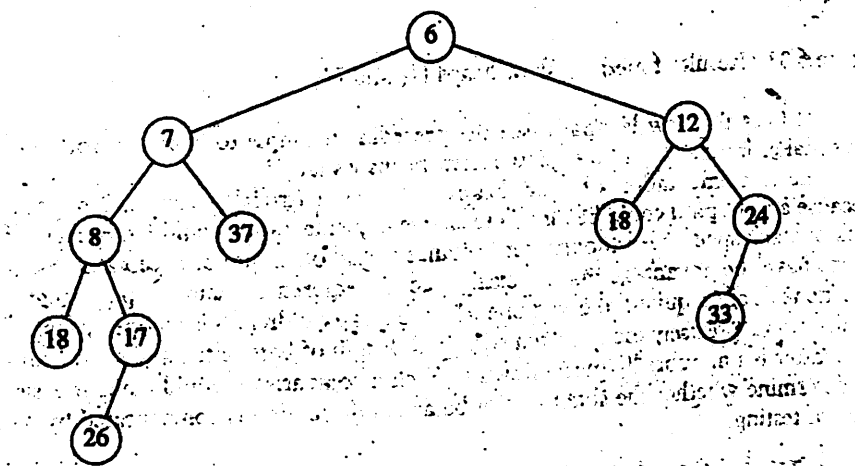


Figure 6.32 Result of merging H_2 with H_1's right subheap

there are no children of this node, so we do not worry about it.) Again, suppose our input is the same two heaps as before, Figure 6.31.

If we recursively merge H_2 with the subheap of H_1 rooted at 8, we will get the heap in Figure 6.32.

Again, this is done recursively, so by the third rule of recursion (Section 1.3) we need not worry about how it was obtained. This heap happens to be leftist, but there is no guarantee that this is always the case. We make this heap the new left child of H_1, and the old left child of H_1 becomes the new right child (see Fig. 6.33).

The entire tree is leftist, but it is easy to see that that is not always true: inserting 15 into this new heap would destroy the leftist property.

We can perform all operations nonrecursively, as with leftist heaps, by merging the right paths and swapping left and right children for every node on the right path, with the exception of the last. After a few examples, it becomes clear that since all but the last node on the right path have their children swapped, the net effect is that

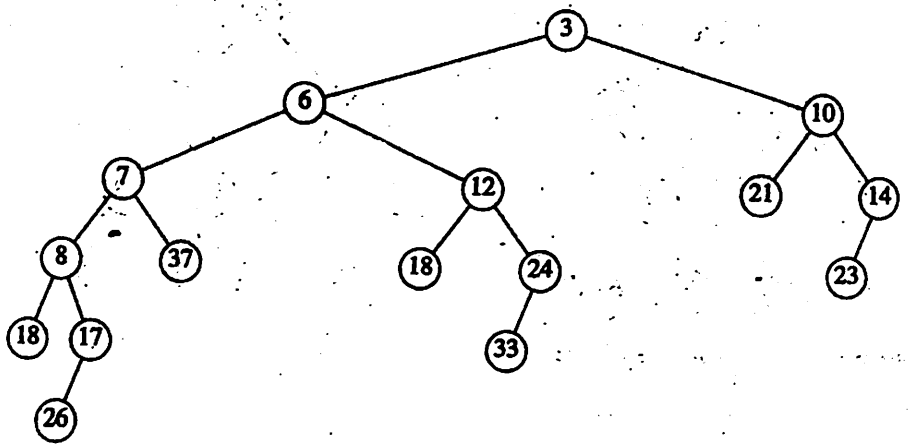


Figure 6.33 Result of merging skew heaps H_1 and H_2

this becomes the new left path (see the preceding example to convince yourself). This makes it very easy to merge two skew heaps visually.*

The implementation of skew heaps is left as a (trivial) exercise. Note that because a right path could be long, a recursive implementation could fail because of lack of stack space, even though performance would otherwise be acceptable. Skew heaps have the advantage that no extra space is required to maintain path lengths and no tests are required to determine when to swap children. It is an open problem to determine precisely the expected right path length of both leftist and skew heaps (the latter is undoubtedly more difficult). Such a comparison would make it easier to determine whether the slight loss of balance information is compensated by the lack of testing.

6.8. Binomial Queues

Although both leftist and skew heaps support merging, insertion, and *DeleteMin* all effectively in $O(\log N)$ time per operation, there is room for improvement because we know that binary heaps support insertion in *constant average* time per operation. Binomial queues support all three operations in $O(\log N)$ worst-case time per operation, but insertions take constant time on average.

6.8.1. Binomial Queue Structure

Binomial queues differ from all the priority queue implementations that we have seen in that a binomial queue is not a heap-ordered tree but rather a *collection* of heap-ordered trees, known as a *forest*. Each of the heap-ordered trees is of a

*This is not exactly the same as the recursive implementation (but yields the same time bounds). If we only swap children for nodes on the right path that are above the point where the merging of right paths terminated due to exhaustion of one heap's right path, we get the same result as the recursive version.

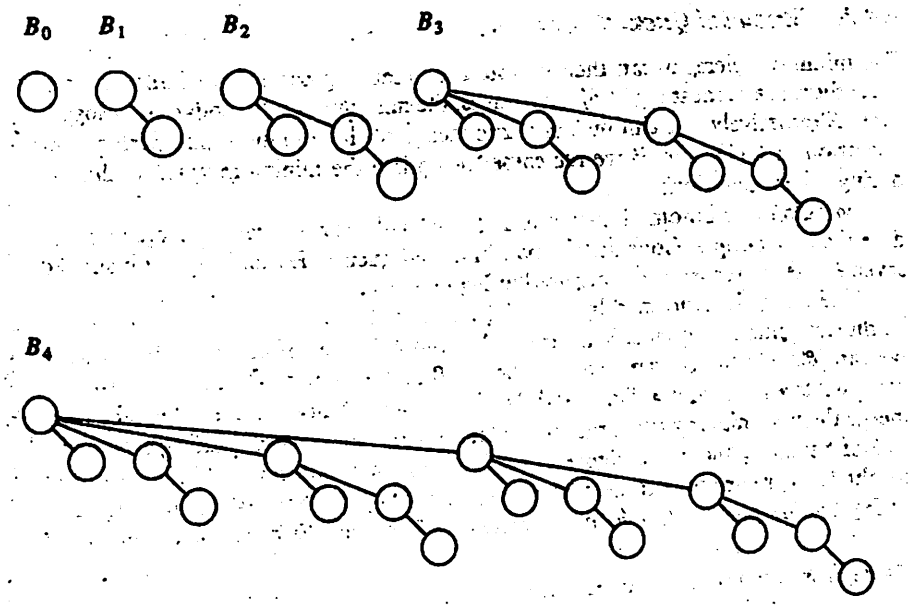


Figure 6.34 Binomial trees B_0, B_1, B_2, B_3, and B_4

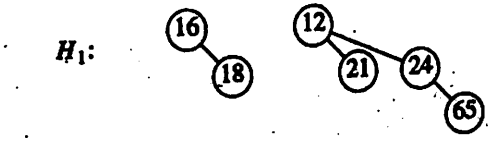


Figure 6.35 Binomial queue H_1 with six elements

constrained form known as a *binomial tree* (the reason for the name will be obvious later). There is at most one binomial tree of every height. A binomial tree of height 0 is a one-node tree; a binomial tree, B_k, of height k is formed by attaching a binomial tree, B_{k-1}, to the root of another binomial tree, B_{k-1}. Figure 6.34 shows binomial trees B_0, B_1, B_2, B_3, and B_4.

From the diagram we see that a binomial tree, B_k, consists of a root with children $B_0, B_1, \ldots, B_{k-1}$. Binomial trees of height k have exactly 2^k nodes, and the number of nodes at depth d is the binomial coefficient $\binom{k}{d}$. If we impose heap order on the binomial trees and allow at most one binomial tree of any height, we can uniquely represent a priority queue of any size by a collection of binomial trees. For instance, a priority queue of size 13 could be represented by the forest B_3, B_2, B_0. We might write this representation as 1101, which not only represents 13 in binary but also represents the fact that B_3, B_2, and B_0 are present in the representation and B_1 is not.

As an example, a priority queue of six elements could be represented as in Figure 6.35.

6.8.2. Binomial Queue Operations

The minimum element can then be found by scanning the roots of all the trees. Since there are at most $\log N$ different trees, the minimum can be found in $O(\log N)$ time. Alternatively, we can maintain knowledge of the minimum and perform the operation in $O(1)$ time, if we remember to update the minimum when it changes during other operations.

Merging two binomial queues is a conceptually easy operation, which we will describe by example. Consider the two binomial queues, H_1 and H_2, with six and seven elements, respectively, pictured in Figure 6.36.

The merge is performed by essentially adding the two queues together. Let H_3 be the new binomial queue. Since H_1 has no binomial tree of height 0 and H_2 does, we can just use the binomial tree of height 0 in H_2 as part of H_3. Next, we add binomial trees of height 1. Since both H_1 and H_2 have binomial trees of height 1, we merge them by making the larger root a subtree of the smaller, creating a binomial tree of height 2, shown in Figure 6.37. Thus, H_3 will not have a binomial tree of height 1. There are now three binomial trees of height 2, namely, the original trees of H_1 and H_2 plus the tree formed by the previous step. We keep one binomial tree

Figure 6.36 Two binomial queues H_1 and H_2

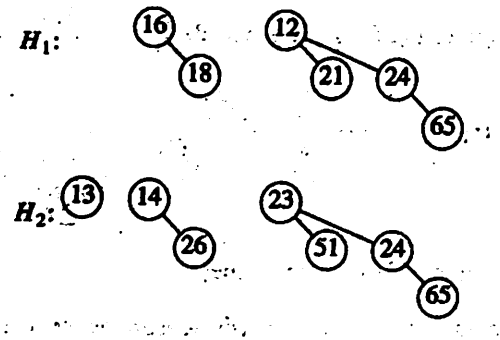


Figure 6.37 Merge of the two B_1 trees in H_1 and H_2

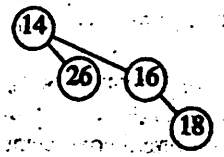
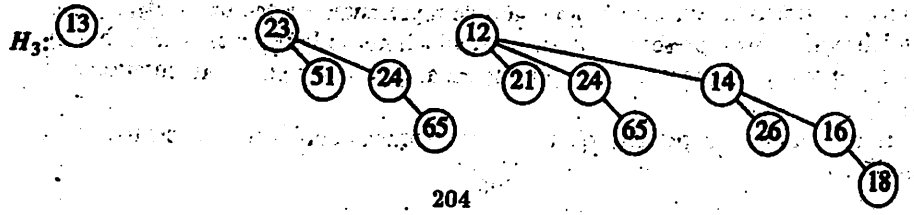


Figure 6.38 Binomial queue H_3: the result of merging H_1 and H_2



of height 2 in H_3 and merge the other two, creating a binomial tree of height 3. Since H_1 and H_2 have no trees of height 3, this tree becomes part of H_3 and we are finished. The resulting binomial queue is shown in Figure 6.38.

Since merging two binomial trees takes constant time with almost any reasonable implementation, and there are $O(\log N)$ binomial trees, the merge takes $O(\log N)$ time in the worst case. To make this operation efficient, we need to keep the trees in the binomial queue sorted by height, which is certainly a simple thing to do.

Insertion is just a special case of merging, since we merely create a one-node tree and perform a merge. The worst-case time of this operation is likewise $O(\log N)$. More precisely, if the priority queue into which the element is being inserted has the property that the smallest nonexistent binomial tree is B_i, the running time is proportional to $i + 1$. For example, H_3 (Fig. 6.38) is missing a binomial tree of height 1, so the insertion will terminate in two steps. Since each tree in a binomial queue is present with probability $\frac{1}{2}$, it follows that we expect an insertion to terminate in two steps, so the average time is constant. Furthermore, an analysis will show that performing N *Inserts* on an initially empty binomial queue will take $O(N)$ worst-case time. Indeed, it is possible to do this operation using only $N - 1$ comparisons; we leave this as an exercise.

As an example, we show in Figures 6.39 through 6.45 the binomial queues that are formed by inserting 1 through 7 in order. Inserting 4 shows off a bad case. We merge 4 with B_0, obtaining a new tree of height 1. We then merge this tree with B_1, obtaining a tree of height 2, which is the new priority queue. We count this as three steps (two tree merges plus the stopping case). The next insertion after 7 is inserted is another bad case and would require three tree merges.

A *DeleteMin* can be performed by first finding the binomial tree with the smallest root. Let this tree be B_k, and let the original priority queue be H. We remove the binomial tree B_k from the forest of trees in H, forming the new binomial queue H'. We also remove the root of B_k, creating binomial trees $B_0, B_1, \ldots, B_{k-1}$, which collectively form priority queue H''. We finish the operation by merging H' and H''.

As an example, suppose we perform a *DeleteMin* on H_3, which is shown again in Figure 6.46. The minimum root is 12, so we obtain the two priority queues H' and H'' in Figure 6.47 and Figure 6.48. The binomial queue that results from merging H' and H'' is the final answer and is shown in Figure 6.49.

For the analysis, note first that the *DeleteMin* operation breaks the original binomial queue into two. It takes $O(\log N)$ time to find the tree containing the minimum element and to create the queues H' and H''. Merging these two queues takes $O(\log N)$ time, so the entire *DeleteMin* operation takes $O(\log N)$ time.

6.8.3. Implementation of Binomial Queues

The *DeleteMin* operation requires the ability to find all the subtrees of the root quickly, so the standard representation of general trees is required: The children of each node are kept in a linked list, and each node has a pointer to its first child (if any). This operation also requires that the children be ordered by the size of their subtrees. We also need to make sure that it is easy to merge two trees. When two trees are merged, one of the trees is added as a child to the other. Since this new tree will be the largest subtree, it makes sense to maintain the subtrees in decreasing

Figure 6.39 After 1 is inserted

Figure 6.40 After 2 is inserted

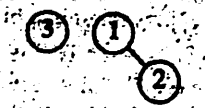


Figure 6.41 After 3 is inserted

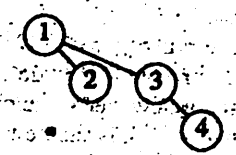


Figure 6.42 After 4 is inserted

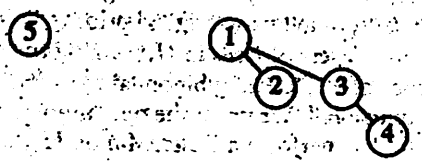


Figure 6.43 After 5 is inserted

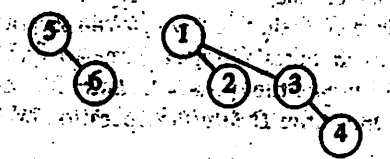


Figure 6.44 After 6 is inserted

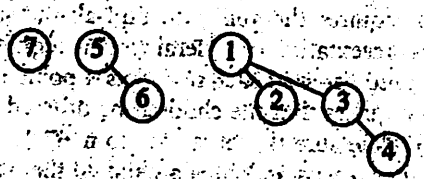


Figure 6.45 After 7 is inserted

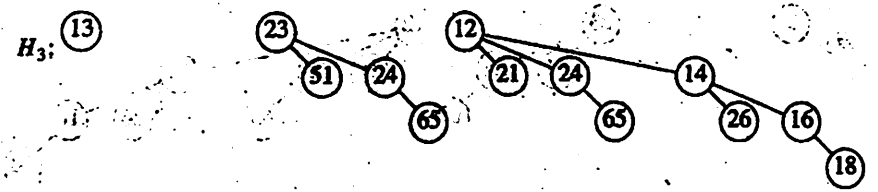


Figure 6.46 Binomial queue H_3

Figure 6.47 Binomial queue H'; containing all the binomial trees in H_3 except B_3

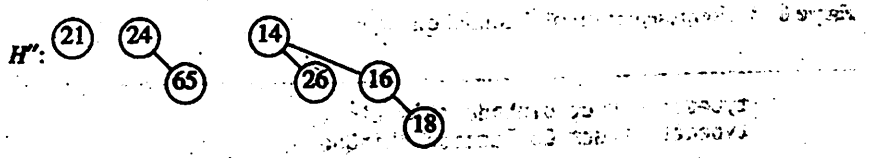


Figure 6.48 Binomial queue H'': B_3 with 12 removed

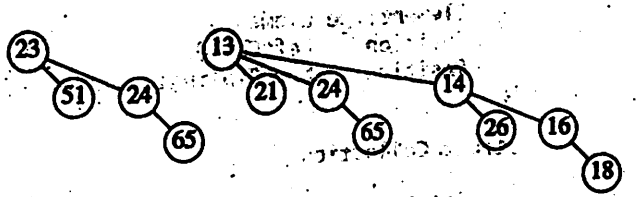


Figure 6.49 Result of *DeleteMin*(H_3)

sizes. Only then will we be able to merge two binomial trees, and thus two binomial queues, efficiently. The binomial queue will be an array of binomial trees.

To summarize, then, each node in a binomial tree will contain the data, first child, and right sibling. The children in a binomial tree are arranged in decreasing rank.

Figure 6.51 shows how the binomial queue in Figure 6.50 is represented. Figure 6.52 shows the type declarations for a node in the binomial tree.

In order to merge two binomial queues, we need a routine to merge two binomial trees of the same size. Figure 6.53 shows how the pointers change when two binomial trees are merged. The code to do this is simple and is shown in Figure 6.54.

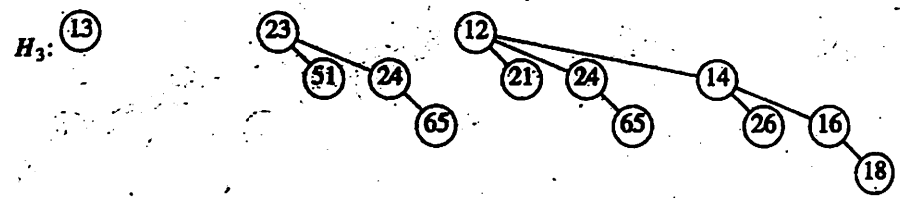


Figure 6.50 Binomial queue H_3 drawn as a forest

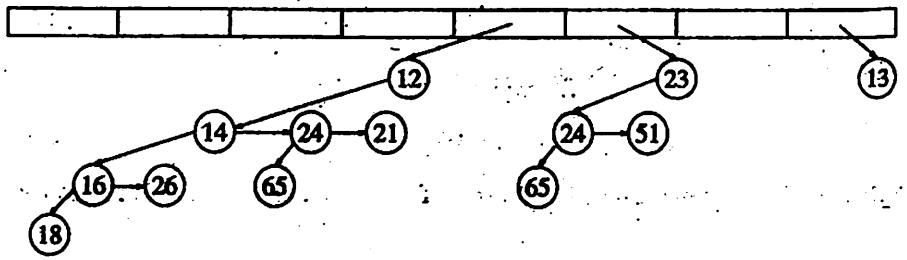


Figure 6.51 Representation of binomial queue H_3

```
typedef struct BinNode *Position;
typedef struct Collection *BinQueue;

struct BinNode
{
    ElementType Element;
    Position    LeftChild;
    Position    NextSibling;
};

struct Collection
{
    int CurrentSize;
    BinTree TheTrees[ MaxTrees ];
};
```

Figure 6.52 Binomial queue type declarations

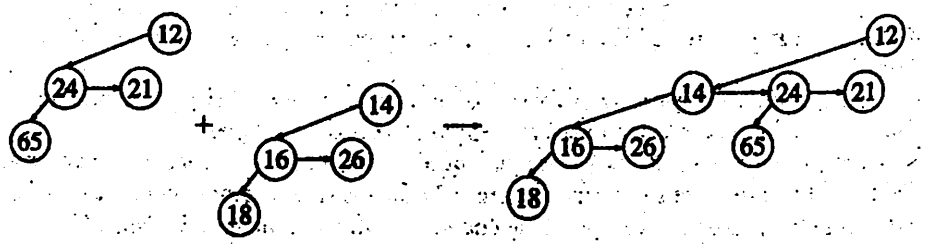


Figure 6.53 Merging two binomial trees


```
/* Return the result of merging equal-sized T1 and T2 */

BinTree
CombineTrees( BinTree T1, BinTree T2 )
{
    if( T1->Element > T2->Element )
        return CombineTrees( T2, T1 );
    T2->NextSibling = T1->LeftChild;
    T1->LeftChild = T2;
    return T1;
}
```

Figure 6.54 Routine to merge two equal-sized binomial trees

We provide a simple implementation of the *Merge* routine. The routine combines H_1 and H_2, placing the result in H_1 and making H_2 empty. At any point we are dealing with trees of rank i. T_1 and T_2 are the trees in H_1 and H_2, respectively, and *Carry* is the tree carried from a previous step (it might be *NULL*). $!!T_1$ is 1 if T_1 exists and is 0 otherwise, and the same is true for the other trees. Depending on each of the eight possible cases, the tree that results for rank i and the *Carry* tree of rank $i + 1$ is formed. This process proceeds from rank 0 to the last rank in the resulting binomial queue. The code is shown in Figure 6.55.

The *DeleteMin* routine for binary queues is given in Figure 6.56.

We can extend binomial queues to support some of the nonstandard operations that binary heaps allow, such as *DecreaseKey* and *Delete*, when the position of the affected element is known. A *DecreaseKey* is a *PercolateUp*, which can be performed in $O(\log N)$ time if we add a field to each node pointing to its parent. An arbitrary *Delete* can be performed by a combination of *DecreaseKey* and *DeleteMin* in $O(\log N)$ time.

Figure 6.55 Routine to merge two priority queues

```
/* Merge two binomial queues */
/* Not optimized for early termination */
/* H1 contains merged result */

BinQueue
Merge( BinQueue H1, BinQueue H2 )
{
    BinTree T1, T2, Carry = NULL;
    int i, j;

    if( H1->CurrentSize + H2->CurrentSize > Capacity )
        Error( "Merge would exceed capacity" );
```

Figure 6.55 (*continued*)

```
        H1->CurrentSize += H2->CurrentSize;
        for( i = 0, j = 1; j <= H1->CurrentSize; i++, j *= 2 )
        {
            T1 = H1->TheTrees[ i ]; T2 = H2->TheTrees[ i ];

            switch( !!T1 + 2 * !!T2 + 4 * !!Carry )
            {
                case 0: /* No trees */
                case 1: /* Only H1 */
                    break;
                case 2: /* Only H2 */
                    H1->TheTrees[ i ] = T2;
                    H2->TheTrees[ i ] = NULL;
                    break;
                case 4: /* Only Carry */
                    H1->TheTrees[ i ] = Carry;
                    Carry = NULL;
                    break;
                case 3: /* H1 and H2 */
                    Carry = CombineTrees( T1, T2 );
                    H1->TheTrees[ i ] = H2->TheTrees[ i ] = NULL;
                    break;
                case 5: /* H1 and Carry */
                    Carry = CombineTrees( T1, Carry );
                    H1->TheTrees[ i ] = NULL;
                    break;
                case 6: /* H2 and Carry */
                    Carry = CombineTrees( T2, Carry );
                    H2->TheTrees[ i ] = NULL;
                    break;
                case 7: /* All three */
                    H1->TheTrees[ i ] = Carry;
                    Carry = CombineTrees( T1, T2 );
                    H2->TheTrees[ i ] = NULL;
                    break;
            }
        }
        return H1;
    }
```



```
ElementType
DeleteMin( BinQueue H )
{
    int i, j;
    int MinTree;    /* The tree with the minimum item */
    BinQueue DeletedQueue;
    Position DeletedTree, OldRoot;
    ElementType MinItem;

    if( IsEmpty( H ) )
    {
        Error( "Empty binomial queue" );
        return -Infinity;
    }

    MinItem = Infinity;
    for( i = 0; i < MaxTrees; i++ )
    {
        if( H->TheTrees[ i ] &&
            H->TheTrees[ i ]->Element < MinItem )
        {
            /* Update minimum */
            MinItem = H->TheTrees[ i ]->Element;
            MinTree = i;
        }
    }

    DeletedTree = H->TheTrees[ MinTree ];
    OldRoot = DeletedTree;
    DeletedTree = DeletedTree->LeftChild;
    free( OldRoot );

    DeletedQueue = Initialize( );
    DeletedQueue->CurrentSize = ( 1 << MinTree ) - 1;
    for( j = MinTree - 1; j >= 0; j-- )
    {
        DeletedQueue->TheTrees[ j ] = DeletedTree;
        DeletedTree = DeletedTree->NextSibling;
        DeletedQueue->TheTrees[ j ]->NextSibling = NULL;
    }

    H->TheTrees[ MinTree ] = NULL;
    H->CurrentSize -= DeletedQueue->CurrentSize + 1;

    Merge( H, DeletedQueue );
    return MinItem;
}
```

Figure 6.56 *DeleteMin* for binomial queues

Summary

In this chapter we have seen various implementations and uses of the priority queue ADT. The standard binary heap implementation is elegant because of its simplicity and speed. It requires no pointers and only a constant amount of extra space, yet supports the priority queue operations efficiently.

We considered the additional *Merge* operation and developed three implementations, each of which is unique in its own way. The leftist heap is a wonderful example of the power of recursion. The skew heap represents a remarkable data structure because of the lack of balance criteria. Its analysis, which we will perform in Chapter 11, is interesting in its own right. The binomial queue shows how a simple idea can be used to achieve a good time bound.

We have also seen several uses of priority queues, ranging from operating systems scheduling to simulation. We will see their use again in Chapters 7, 9, and 10.

Exercises

6.1 Suppose that we replace the *DeleteMin* function with *FindMin*. Can both *Insert* and *FindMin* be implemented in constant time?

6.2 a. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.

b. Show the result of using the linear-time algorithm to build a binary heap using the same input.

6.3 Show the result of performing three *DeleteMin* operations in the heap of the previous exercise.

6.4 Write the routines to do a *percolate up* and a *percolate down* in a binary heap.

6.5 Write and test a program that performs the operations *Insert*, *DeleteMin*, *BuildHeap*, *FindMin*, *DecreaseKey*, *Delete*, and *IncreaseKey* in a binary heap.

6.6 How many nodes are in the large heap in Figure 6.13?

6.7 a. Prove that for binary heaps, *BuildHeap* does at most $2N - 2$ comparisons between elements.

b. Show that a heap of eight elements can be constructed in eight comparisons between heap elements.

**c. Give an algorithm to build a binary heap in $\frac{13}{8}N + O(\log N)$ element comparisons.

6.8 Show that the expected depth of the kth smallest element in a large complete heap (you may assume $N = 2^k - 1$) is bounded by $\log k$.

6.9 *a. Give an algorithm to find all nodes less than some value, X, in a binary heap. Your algorithm should run in $O(K)$, where K is the number of nodes output.

b. Does your algorithm extend to any of the other heap structures discussed in this chapter?

*c. Give an algorithm that finds an arbitrary item X in a binary heap using at most roughly $3N/4$ comparisons.

6.10 Propose an algorithm to insert M nodes into a binary heap on N elements in $O(M + \log N \log \log N)$ time. Prove your time bound.

6.11 Write a program to take N elements and do the following:

a. Insert them into a heap one by one.

b. Build a heap in linear time.

Compare the running time of both algorithms for sorted, reverse-ordered, and random inputs.

6.12 Each *DeleteMin* operation uses $2 \log N$ comparisons in the worst case.

*a. Propose a scheme so that the *DeleteMin* operation uses only $\log N + \log \log N + O(1)$ comparisons between elements. This need not imply less data movement.

**b. Extend your scheme in part (a) so that only $\log N + \log \log \log N + O(1)$ comparisons are performed.

**c. How far can you take this idea?

d. Do the savings in comparisons compensate for the increased complexity of your algorithm?

6.13 If a d-heap is stored as an array, for an entry located in position i, where are the parents and children?

6.14 Suppose we need to perform M *PercolateUps* and N *DeleteMins* on a d-heap that initially has N elements.

a. What is the total running time of all operations in terms of M, N, and d?

b. If $d = 2$, what is the running time of all heap operations?

c. If $d = \Theta(N)$, what is the total running time?

*d. What choice of d minimizes the total running time?

6.15 A *min-max heap* is a data structure that supports both *DeleteMin* and *DeleteMax* in $O(\log N)$ per operation. The structure is identical to a binary heap, but the heap order property is that for any node, X, at even depth, the key stored at X is smaller than the parent but larger than the grandparent (where this makes sense), and for any node X at odd depth, the key stored at X is larger than the parent but smaller than the grandparent. See Figure 6.57.

a. How do we find the minimum and maximum elements?

*b. Give an algorithm to insert a new node into the min-max heap.

*c. Give an algorithm to perform *DeleteMin* and *DeleteMax*.

*d. Can you build a min-max heap in linear time?

**e. Suppose we would like to support *DeleteMin*, *DeleteMax*, and *Merge*. Propose a data structure to support all operations in $O(\log N)$ time.

6.16 Merge the two leftist heaps in Figure 6.58.

6.17 Show the result of inserting keys 1 to 15 in order into an initially empty leftist heap.

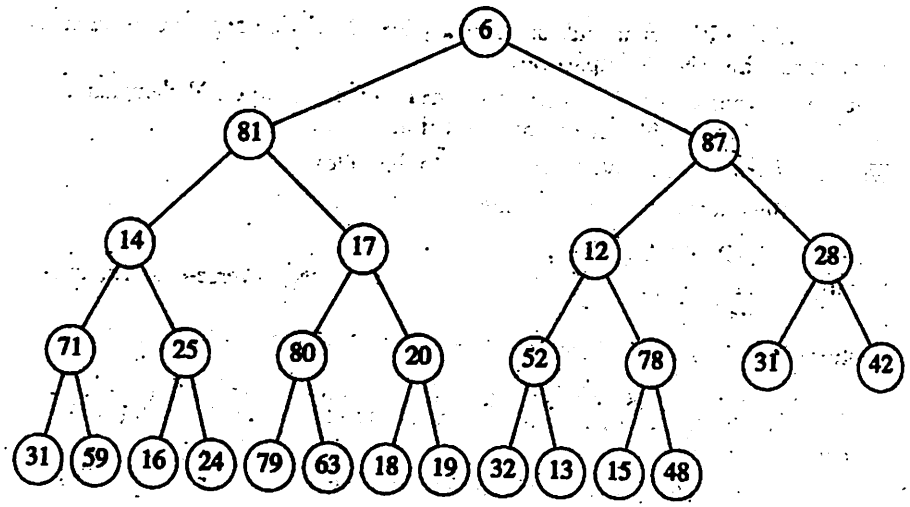


Figure 6.57 Min-max heap

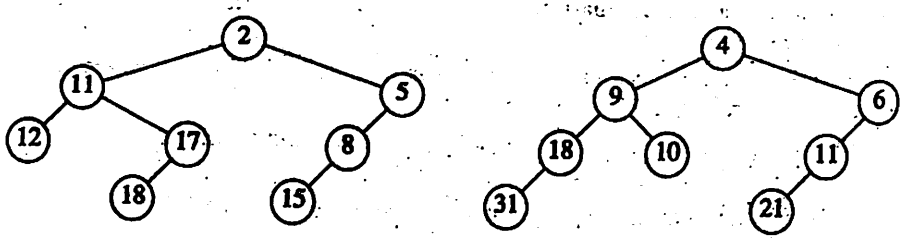


Figure 6.58

6.18 Prove or disprove: A perfectly balanced tree forms if keys 1 to $2^k - 1$ are inserted in order into an initially empty leftist heap.

6.19 Give an example of input that generates the best leftist heap.

6.20 a. Can leftist heaps efficiently support *DecreaseKey*?

b. What changes, if any (if possible), are required to do this?

6.21 One way to delete nodes from a known position in a leftist heap is to use a lazy strategy. To delete a node, merely mark it deleted. When a *FindMin* or *DeleteMin* is performed, there is a potential problem if the root is marked deleted, since then the node has to be actually deleted and the real minimum needs to be found, which may involve deleting other marked nodes. In this strategy, *Deletes* cost one unit, but the cost of a *DeleteMin* or *FindMin* depends on the number of nodes that are marked deleted. Suppose that after a *DeleteMin* or *FindMin* there are k fewer marked nodes than before the operation.

*a. Show how to perform the *DeleteMin* in $O(k \log N)$ time.

**b. Propose an implementation, with an analysis to show that the time to perform the *DeleteMin* is $O(k \log(2N/k))$.

6.22 We can perform *BuildHeap* in linear time for leftist heaps by considering each element as a one-node leftist heap, placing all these heaps on a queue, and performing the following step: Until only one heap is on the queue, dequeue two heaps, merge them, and enqueue the result.

a. Prove that this algorithm is $O(N)$ in the worst case.

b. Why might this algorithm be preferable to the algorithm described in the text?

6.23 Merge the two skew heaps in Figure 6.58.

6.24 Show the result of inserting keys 1 to 15 in order into a skew heap.

6.25 Prove or disprove: A perfectly balanced tree forms if the keys 1 to $2^k - 1$ are inserted in order into an initially empty skew heap.

6.26 A skew heap of N elements can be built using the standard binary heap algorithm. Can we use the same merging strategy described in Exercise 6.22 for skew heaps to get an $O(N)$ running time?

6.27 Prove that a binomial tree B_k has binomial trees $B_0, B_1, \ldots, B_{k-1}$ as children of the root.

6.28 Prove that a binomial tree of height k has $\binom{k}{d}$ nodes at depth d.

6.29 Merge the two binomial queues in Figure 6.59.

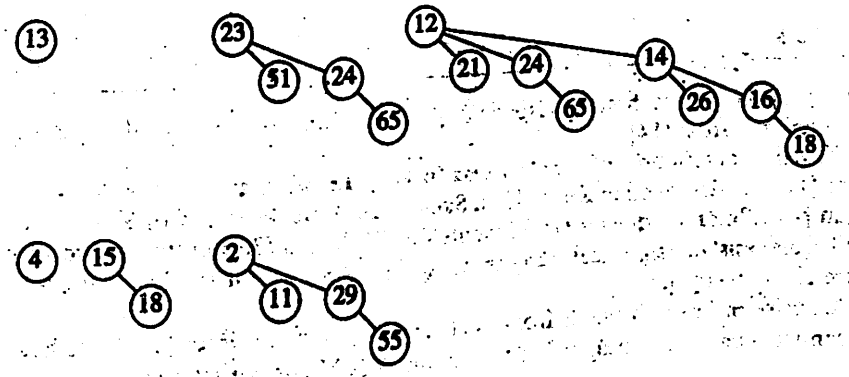
6.30 a. Show that N *Inserts* into an initially empty binomial queue takes $O(N)$ time in the worst case.

b. Give an algorithm to build a binomial queue of N elements, using at most $N - 1$ comparisons between elements.

*c. Propose an algorithm to insert M nodes into a binomial queue of N elements in $O(M + \log N)$ worst-case time. Prove your bound.

6.31 Write an efficient routine to perform *Insert* using binomial queues. Do not call *Merge*.

Figure 6.59



6.32 For the binomial queue:

a. What happens when the call *Merge*(H, H) is made? Modify the code to fix this problem.

b. Modify the *Merge* routine to terminate merging if there are no trees left in H_2 and the *Carry* tree is *NULL*.

c. Modify the *Merge* so that the smaller tree is always merged into the larger.

*6.33 Suppose we extend binomial queues to allow at most two trees of the same height per structure. Can we obtain $O(1)$ worst-case time for insertion while retaining $O(\log N)$ for the other operations?

6.34 Suppose you have a number of boxes, each of which can hold total weight C and items $i_1, i_2, i_3, \ldots, i_N$, which weigh $w_1, w_2, w_3, \ldots, w_N$, respectively. The object is to pack all the items without placing more weight in any box than its capacity and using as few boxes as possible. For instance, if $C = 5$, and the items have weights 2, 2, 3, 3, then we can solve the problem with two boxes.

In general, this problem is very hard, and no efficient solution is known. Write programs to implement efficiently the following approximation strategies:

*a. Place the weight in the first box for which it fits (creating a new box if there is no box with enough room). (This strategy and all that follow would give three boxes, which is suboptimal.)

b. Place the weight in the box with the most room for it.

*c. Place the weight in the most filled box that can accept it without overflowing.

**d. Are any of these strategies enhanced by presorting the items by weight?

6.35 Suppose we want to add the *DecreaseAllKeys*(Δ) operation to the heap repertoire. The result of this operation is that all keys in the heap have their value decreased by an amount Δ. For the heap implementation of your choice, explain the necessary modifications so that all other operations retain their running times and *DecreaseAllKeys* runs in $O(1)$.

6.36 Which of the two selection algorithms has the better time bound?

References

The binary heap was first described in [27]. The linear-time algorithm for its construction is from [14].

The first description of d-heaps was in [19]. Leftist heaps were invented by Crane [11] and described in Knuth [21]. Skew heaps were developed by Sleator and Tarjan [23]. Binomial queues were invented by Vuillemin [26]; Brown provided a detailed analysis and empirical study showing that they perform well in practice [4], if carefully implemented.

Exercise 6.7(b–c) is taken from [17]. Exercise 6.9(c) is from [6]. A method for constructing binary heaps that uses about $1.52N$ comparisons on average is

described in [22]. Lazy deletion in leftist heaps (Exercise 6.21) is from [10]. A solution to Exercise 6.33 can be found in [8].

Min-max heaps (Exercise 6.15) were originally described in [1]. A more efficient implementation of the operations is given in [18] and [24]. Alternative representations for double-ended priority queues are the *deap* and *diamond dequeue*. Details can be found in [5], [7], and [9]. Solutions to 6.15(e) are given in [12] and [20].

A theoretically interesting priority queue representation is the *Fibonacci heap* [16], which we will describe in Chapter 11. The Fibonacci heap allows all operations to be performed in $O(1)$ amortized time, except for deletions, which are $O(\log N)$. *Relaxed heaps* [13] achieve identical bounds in the worst case (with the exception of *Merge*). The procedure of [3] achieves optimal worst-case bounds for all operations. Another interesting implementation is the *pairing heap* [15], which is described in Chapter 12. Finally, priority queues that work when the data consist of small integers are described in [2] and [25].

1. M. D. Atkinson, J. R. Sack, N. Santoro, and T. Strothotte, "Min-Max Heaps and Generalized Priority Queues," *Communications of the ACM*, 29 (1986), 996–1000.
2. J. D. Bright, "Range Restricted Mergeable Priority Queues," *Information Processing Letters*, 47 (1993), 159–164.
3. G. S. Brodal, "Worst-Case Efficient Priority Queues," *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (1996), 52–58.
4. M. R. Brown, "Implementation and Analysis of Binomial Queue Algorithms," *SIAM Journal on Computing*, 7 (1978), 298–319.
5. S. Carlsson, "The Deap—A Double-Ended Heap to Implement Double-Ended Priority Queues," *Information Processing Letters*, 26 (1987), 33–36.
6. S. Carlsson and J. Chen, "The Complexity of Heaps," *Proceedings of the Third Symposium on Discrete Algorithms* (1992), 393–402.
7. S. Carlsson, J. Chen, and T. Strothotte, "A Note on the Construction of the Data Structure 'Deap'," *Information Processing Letters*, 31 (1989), 315–317.
8. S. Carlsson, J. I. Munro, and P. V. Poblete, "An Implicit Binomial Queue with Constant Insertion Time," *Proceedings of First Scandinavian Workshop on Algorithm Theory* (1988), 1–13.
9. S. C. Chang and M. W. Due, "Diamond Deque: A Simple Data Structure for Priority Deques," *Information Processing Letters*, 46 (1993), 231–237.
10. D. Cheriton and R. E. Tarjan, "Finding Minimum Spanning Trees," *SIAM Journal on Computing*, 5 (1976), 724–742.
11. C. A. Crane, "Linear Lists and Priority Queues as Balanced Binary Trees," *Technical Report STAN-CS-72-259*, Computer Science Department, Stanford University, Stanford, Calif., 1972.
12. Y. Ding and M. A. Weiss, "The Relaxed Min-Max Heap: A Mergeable Double-Ended Priority Queue," *Acta Informatica*, 30 (1993), 215–231.
13. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation," *Communications of the ACM*, 31 (1988), 1343–1354.
14. R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, 7 (1964), 701.
15. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, "The Pairing Heap: A New Form of Self-adjusting Heap," *Algorithmica*, 1 (1986), 111–129.
16. M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the ACM*, 34 (1987), 596–615.

17. G. H. Gonnet and J. I. Munro, "Heaps on Heaps," *SIAM Journal on Computing*, 15 (1986), 964–971.
18. A. Hasham and J. R. Sack, "Bounds for Min-max Heaps," *BIT*, 27 (1987), 315–323.
19. D. B. Johnson, "Priority Queues with Update and Finding Minimum Spanning Trees," *Information Processing Letters*, 4 (1975), 53–57.
20. C. M. Khoong and H. W. Leong, "Double-Ended Binomial Queues," *Proceedings of the Fourth Annual International Symposium on Algorithms and Computation* (1993).
21. D. E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.
22. C. J. H. McDiarmid and B. A. Reed, "Building Heaps Fast," *Journal of Algorithms*, 10 (1989), 352–365.
23. D. D. Sleator and R. E. Tarjan, "Self-adjusting Heaps," *SIAM Journal on Computing*, 15 (1986), 52–69.
24. T. Strothotte, P. Eriksson, and S. Vallner, "A Note on Constructing Min-max Heaps," *BIT*, 29 (1989), 251–256.
25. P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and Implementation of an Efficient Priority Queue," *Mathematical Systems Theory*, 10 (1977), 99–127.
26. J. Vuillemin, "A Data Structure for Manipulating Priority Queues," *Communications of the ACM*, 21 (1978), 309–314.
27. J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, 7 (1964), 347–348.

CHAPTER 7

Sorting

In this chapter we discuss the problem of sorting an array of elements. To simplify matters, we will assume in our examples that the array contains only integers, although, obviously, more complicated structures are possible. For most of this chapter, we will also assume that the entire sort can be done in main memory, so that the number of elements is relatively small (less than a million). Sorts that cannot be performed in main memory and must be done on disk or tape are also quite important. This type of sorting, known as external sorting, will be discussed at the end of the chapter.

Our investigation of internal sorting will show that

- There are several easy algorithms to sort in $O(N^2)$, such as insertion sort.
- There is an algorithm, Shellsort, that is very simple to code, runs in $o(N^2)$, and is efficient in practice.
- There are slightly more complicated $O(N \log N)$ sorting algorithms.
- Any general-purpose sorting algorithm requires $\Omega(N \log N)$ comparisons.

The rest of this chapter will describe and analyze the various sorting algorithms. These algorithms contain interesting and important ideas for code optimization as well as algorithm design. Sorting is also an example where the analysis can be precisely performed. Be forewarned that where appropriate, we will do as much analysis as possible.

7.1. Preliminaries

The algorithms we describe will all be interchangeable. Each will be passed an array containing the elements and an integer containing the number of elements.

We will assume that N, the number of elements passed to our sorting routines, has already been checked and is legal. In accordance with C conventions, the data will start at position 0 for all the sorts.

We will also assume the existence of the "<" and ">" operators, which can be used to place a consistent ordering on the input. Besides the assignment operator, these are the only operations allowed on the input data. Sorting under these conditions is known as *comparison-based* sorting.

7.2. Insertion Sort

7.2.1. The Algorithm

One of the simplest sorting algorithms is the *insertion sort*. Insertion sort consists of $N - 1$ *passes*. For pass $P = 1$ through $N - 1$, insertion sort ensures that the elements in positions 0 through P are in sorted order. Insertion sort makes use of the fact that elements in positions 0 through $P - 1$ are already known to be in sorted order. Figure 7.1 shows a sample array after each pass of insertion sort.

Figure 7.1 shows the general strategy. In pass P, we move the element in position P left until its correct place is found among the first $P + 1$ elements. The code in Figure 7.2 implements this strategy. Lines 2 through 5 implement that data movement without the explicit use of swaps. The element in position P is saved in *Tmp*, and all larger elements (prior to position P) are moved one spot to the right. Then *Tmp* is placed in the correct spot. This is the same technique that was used in the implementation of binary heaps.

Figure 7.1 Insertion sort after each pass

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Figure 7.2 Insertion sort routine

```
    void
    InsertionSort( ElementType A[ ], int N )
    {
        int j, P;

        ElementType Tmp;
/* 1*/  for( P = 1; P < N; P++ )
        {
/* 2*/      Tmp = A[ P ];
/* 3*/      for( j = P; j > 0 && A[ j - 1 ] > Tmp; j-- )
/* 4*/          A[ j ] = A[ j - 1 ];
/* 5*/      A[ j ] = Tmp;
        }
    }
```


7.2.2. Analysis of Insertion Sort

Because of the nested loops, each of which can take N iterations, insertion sort is $O(N^2)$. Furthermore, this bound is tight, because input in reverse order can achieve this bound. A precise calculation shows that the test at line 4 can be executed at most $P + 1$ times for each value of P. Summing over all P gives a total of

$$\sum_{i=2}^{N} i = 2 + 3 + 4 + \cdots + N = \Theta(N^2)$$

On the other hand, if the input is presorted, the running time is $O(N)$, because the test in the inner *for* loop always fails immediately. Indeed, if the input is almost sorted (this term will be more rigorously defined in the next section), insertion sort will run quickly. Because of this wide variation, it is worth analyzing the average-case behavior of this algorithm. It turns out that the average case is $\Theta(N^2)$ for insertion sort, as well as for a variety of other sorting algorithms, as the next section shows.

7.3. A Lower Bound for Simple Sorting Algorithms

An *inversion* in an array of numbers is any ordered pair (i, j) having the property that $i < j$ but $A[i] > A[j]$. In the example of the last section, the input list 34, 8, 64, 51, 32, 21 had nine inversions, namely (34, 8), (34, 32), (34, 21), (64, 51), (64, 32), (64, 21), (51, 32), (51, 21), and (32, 21). Notice that this is exactly the number of swaps that needed to be (implicitly) performed by insertion sort. This is always the case, because swapping two adjacent elements that are out of place removes exactly one inversion, and a sorted array has no inversions. Since there is $O(N)$ other work involved in the algorithm, the running time of insertion sort is $O(I + N)$, where I is the number of inversions in the original array. Thus, insertion sort runs in linear time if the number of inversions is $O(N)$.

We can compute precise bounds on the average running time of insertion sort by computing the average number of inversions in a permutation. As usual, defining *average* is a difficult proposition. We will assume that there are no duplicate elements (if we allow duplicates, it is not even clear what the average number of duplicates is). Using this assumption, we can assume that the input is some permutation of the first N integers (since only relative ordering is important) and that all are equally likely. Under these assumptions, we have the following theorem:

THEOREM 7.1.

The average number of inversions in an array of N distinct numbers is $N(N - 1)/4$.

PROOF:

For any list, L, of numbers, consider L_r, the list in reverse order. The reverse list of the example is 21, 32, 51, 64, 8, 34. Consider any pair of two numbers in the list (x, y), with $y > x$. Clearly, in exactly one of L and L_r this ordered

pair represents an inversion. The total number of these pairs in a list L and its reverse L_r is $N(N-1)/2$. Thus, an average list has half this amount, or $N(N-1)/4$ inversions.

This theorem implies that insertion sort is quadratic on average. It also provides a very strong lower bound about any algorithm that only exchanges adjacent elements.

THEOREM 7.2.

Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average.

PROOF:

The average number of inversions is initially $N(N-1)/4 = \Omega(N^2)$. Each swap removes only one inversion, so $\Omega(N^2)$ swaps are required.

This is an example of a lower-bound proof. It is valid not only for insertion sort, which performs adjacent exchanges implicitly, but also for other simple algorithms such as bubble sort and selection sort, which we will not describe here. In fact, it is valid over an entire *class* of sorting algorithms, including those undiscovered, that perform only adjacent exchanges. Because of this, this proof cannot be confirmed empirically. Although this lower-bound proof is rather simple, in general proving lower bounds is much more complicated than proving upper bounds and in some cases resembles voodoo.

This lower bound shows us that in order for a sorting algorithm to run in subquadratic, or $o(N^2)$, time, it must do comparisons and, in particular, exchanges between elements that are far apart. A sorting algorithm makes progress by eliminating inversions, and to run efficiently, it must eliminate more than just one inversion per exchange.

7.4. Shellsort

Shellsort, named after its inventor, Donald Shell, was one of the first algorithms to break the quadratic time barrier, although it was not until several years after its initial discovery that a subquadratic time bound was proven. As suggested in the previous section, it works by comparing elements that are distant; the distance between comparisons decreases as the algorithm runs until the last phase, in which adjacent elements are compared. For this reason, Shellsort is sometimes referred to as *diminishing increment* sort.

Shellsort uses a sequence, $h_1, h_2, \ldots, h_t$, called the *increment sequence*. Any increment sequence will do as long as $h_1 = 1$, but some choices are better than others (we will discuss that question later). After a *phase*, using some increment h_k, for every i, we have $A[i] \leq A[i + h_k]$ (where this makes sense); all elements spaced h_k apart are sorted. The file is then said to be *h_k-sorted*. For example, Figure 7.3 shows an array after several phases of Shellsort. An important property of Shellsort (which we state without proof) is that an h_k-sorted file that is then h_{k-1}-sorted

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

Figure 7.3 Shellsort after each pass

remains h_k-sorted. If this were not the case, the algorithm would likely be of little value, since work done by early phases would be undone by later phases.

The general strategy to h_k-sort is for each position, i, in $h_k, h_k + 1, \ldots, N - 1$, place the element in the correct spot among $i, i - h_k, i - 2h_k$, etc. Although this does not affect the implementation, a careful examination shows that the action of an h_k-sort is to perform an insertion sort on h_k independent subarrays. This observation will be important when we analyze the running time of Shellsort.

A popular (but poor) choice for increment sequence is to use the sequence suggested by Shell: $h_t = \lfloor N/2 \rfloor$, and $h_k = \lfloor h_{k+1}/2 \rfloor$. Figure 7.4 contains a program that implements Shellsort using this sequence. We shall see later that there are increment sequences that give a significant improvement in the algorithm's running time; even a minor change can drastically affect performance (Exercise 7.10).

The program in Figure 7.4 avoids the explicit use of swaps in the same manner as our implementation of insertion sort.

Figure 7.4 Shellsort routine using Shell's increments (better increments are possible)

```
        void
        Shellsort( ElementType A[ ], int N )
        {
            int i, j, Increment;
            ElementType Tmp;

/* 1*/      for( Increment = N / 2; Increment > 0; Increment /= 2 )
/* 2*/          for( i = Increment; i < N; i++ )
                {
/* 3*/              Tmp = A[ i ];
/* 4*/              for( j = i; j >= Increment; j -= Increment )
/* 5*/                  if( Tmp < A[ j - Increment ] )
/* 6*/                      A[ j ] = A[ j - Increment ];
                    else
                        break;
/* 7*/              A[ j ] = Tmp;
                }
        }
```

7.4.1. Worst-Case Analysis of Shellsort

Although Shellsort is simple to code, the analysis of its running time is quite another story. The running time of Shellsort depends on the choice of increment sequence, and the proofs can be rather involved. The average-case analysis of Shellsort is a long-standing open problem, except for the most trivial increment sequences. We will prove tight worst-case bounds for two particular increment sequences.

THEOREM 7.3.

The worst-case running time of Shellsort, using Shell's increments, is $\Theta(N^2)$.

PROOF:

The proof requires showing not only an upper bound on the worst-case running time but also showing that there exists some input that actually takes $\Omega(N^2)$ time to run. We prove the lower bound first, by constructing a bad case. First, we choose N to be a power of 2. This makes all the increments even, except for the last increment, which is 1. Now, we will give as input an array, *InputData*, with the $N/2$ largest numbers in the even positions and the $N/2$ smallest numbers in the odd positions (for this proof, the first position is position 1). As all the increments except the last are even, when we come to the last pass, the $N/2$ largest numbers are still all in even positions and the $N/2$ smallest numbers are still all in odd positions. The ith smallest number ($i \leq N/2$) is thus in position $2i - 1$ before the beginning of the last pass. Restoring the ith element to its correct place requires moving it $i - 1$ spaces in the array. Thus, to merely place the $N/2$ smallest elements in the correct place requires at least $\sum_{i=1}^{N/2} i - 1 = \Omega(N^2)$ work. As an example, Figure 7.5 shows a bad (but not the worst) input when $N = 16$. The number of inversions remaining after the 2-sort is exactly $1 + 2 + 3 + 4 + 5 + 6 + 7 = 28$; thus, the last pass will take considerable time.

To finish the proof, we show the upper bound of $O(N^2)$. As we have observed before, a pass with increment h_k consists of h_k insertion sorts of about N/h_k elements. Since insertion sort is quadratic, the total cost of a pass is $O(h_k(N/h_k)^2) = O(N^2/h_k)$. Summing over all passes gives a total bound of $O(\sum_{i=1}^{t} N^2/h_i) = O(N^2 \sum_{i=1}^{t} 1/h_i)$. Because the increments form a geometric series with common ratio 2, and the largest term in the series is $h_1 = 1$, $\sum_{i=1}^{t} 1/h_i < 2$. Thus we obtain a total bound of $O(N^2)$.

The problem with Shell's increments is that pairs of increments are not necessarily relatively prime, and thus the smaller increment can have little effect. Hibbard suggested a slightly different increment sequence, which gives better results in practice (and theoretically). His increments are of the form $1, 3, 7, \ldots, 2^k - 1$. Although these increments are almost identical, the key difference is that consecutive increments have no common factors. We now analyze the worst-case running time of Shellsort for this increment sequence. The proof is rather complicated.

Figure 7.5 Bad case for Shellsort with Shell's increments (positions are numbered 1 to 16)

Start	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 8-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 4-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 2-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 1-sort	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

THEOREM 7.4.

The worst-case running time of Shellsort using Hibbard's increments is $\Theta(N^{3/2})$.

PROOF:

We will prove only the upper bound and leave the proof of the lower bound as an exercise. The proof requires some well-known results from additive number theory. References to these results are provided at the end of the chapter.

For the upper bound, as before, we bound the running time of each pass and sum over all passes. For increments $h_k > N^{1/2}$, we will use the bound $O(N^2/h_k)$ from the previous theorem. Although this bound holds for the other increments, it is too large to be useful. Intuitively, we must take advantage of the fact that *this* increment sequence is *special*. What we need to show is that for any element A_P in position P, when it is time to perform an h_k-sort, there are only a few elements to the left of position P that are larger than A_P.

When we come to h_k-sort the input array, we know that it has already been h_{k+1}- and h_{k+2}-sorted. Prior to the h_k-sort, consider elements in positions P and $P - i$, $i \leq P$. If i is a multiple of h_{k+1} or h_{k+2}, then clearly $A[P - i] < A[P]$. We can say more, however. If i is expressible as a linear combination (in nonnegative integers) of h_{k+1} and h_{k+2}, then $A[P - i] < A[P]$. As an example, when we come to 3-sort, the file is already 7- and 15-sorted. 52 is expressible as a linear combination of 7 and 15, because $52 = 1 * 7 + 3 * 15$. Thus, $A[100]$ cannot be larger than $A[152]$ because $A[100] \leq A[107] \leq A[122] \leq A[137] \leq A[152]$.

Now, $h_{k+2} = 2h_{k+1} + 1$, so h_{k+1} and h_{k+2} cannot share a common factor. In this case, it is possible to show that all integers that are at least as large as $(h_{k+1} - 1)(h_{k+2} - 1) = 8h_k^2 + 4h_k$ can be expressed as a linear combination of h_{k+1} and h_{k+2} (see the reference at the end of the chapter).

This tells us that the body of the *for* loop at line 4 can be executed at most $8h_k + 4 = O(h_k)$ times for each of the $N - h_k$ positions. This gives a bound of $O(Nh_k)$ per pass.

Using the fact that about half the increments satisfy $h_k < \sqrt{N}$, and assuming that t is even, the total running time is then

$$O\left(\sum_{k=1}^{t/2} Nh_k + \sum_{k=t/2+1}^{t} N^2/h_k\right) = O\left(N\sum_{k=1}^{t/2} h_k + N^2 \sum_{k=t/2+1}^{t} 1/h_k\right)$$

Because both sums are geometric series, and since $h_{t/2} = \Theta(\sqrt{N})$, this simplifies to

$$= O(Nh_{t/2}) + O\left(\frac{N^2}{h_{t/2}}\right) = O(N^{3/2})$$

The average-case running time of Shellsort, using Hibbard's increments, is thought to be $O(N^{5/4})$, based on simulations, but nobody has been able to prove this. Pratt has shown that the $\Theta(N^{3/2})$ bound applies to a wide range of increment sequences.

Sedgewick has proposed several increment sequences that give an $O(N^{4/3})$ worst-case running time (also achievable). The average running time is conjectured to be $O(N^{7/6})$ for these increment sequences. Empirical studies show that these sequences perform significantly better in practice than Hibbard's. The best of these is the sequence $\{1, 5, 19, 41, 109, \ldots\}$, in which the terms are either of the form $9 \cdot 4^i - 9 \cdot 2^i + 1$ or $4^i - 3 \cdot 2^i + 1$. This is most easily implemented by placing these values in an array. This increment sequence is the best known in practice, although there is a lingering possibility that some increment sequence might exist that could give a significant improvement in the running time of Shellsort.

There are several other results on Shellsort that (generally) require difficult theorems from number theory and combinatorics and are mainly of theoretical interest. Shellsort is a fine example of a very simple algorithm with an extremely complex analysis.

The performance of Shellsort is quite acceptable in practice, even for N in the tens of thousands. The simplicity of the code makes it the algorithm of choice for sorting up to moderately large input.

7.5. Heapsort

As mentioned in Chapter 6, priority queues can be used to sort in $O(N \log N)$ time. The algorithm based on this idea is known as *heapsort* and gives the best Big-Oh running time we have seen so far. In practice however, it is slower than a version of Shellsort that uses Sedgewick's increment sequence.

Recall, from Chapter 6, that the basic strategy is to build a binary heap of N elements. This stage takes $O(N)$ time. We then perform N *DeleteMin* operations. The elements leave the heap smallest first, in sorted order. By recording these elements in a second array and then copying the array back, we sort N elements. Since each *DeleteMin* takes $O(\log N)$ time, the total running time is $O(N \log N)$.

The main problem with this algorithm is that it uses an extra array. Thus, the memory requirement is doubled. This could be a problem in some instances. Notice that the extra time spent copying the second array back to the first is only $O(N)$, so that this is not likely to affect the running *time* significantly. The problem is space.

A clever way to avoid using a second array makes use of the fact that after each *DeleteMin*, the heap shrinks by 1. Thus the cell that was last in the heap can be used to store the element that was just deleted. As an example, suppose we have a heap with six elements. The first *DeleteMin* produces A_1. Now the heap has only five elements, so we can place A_1 in position 6. The next *DeleteMin* produces A_2. Since the heap will now only have four elements, we can place A_2 in position 5.

Using this strategy, after the last *DeleteMin* the array will contain the elements in *decreasing* sorted order. If we want the elements in the more typical *increasing* sorted order, we can change the ordering property so that the parent has a larger key than the child. Thus we have a (*max*)heap.

In our implementation, we will use a (*max*)heap, but avoid the actual ADT for the purposes of speed. As usual, everything is done in an array. The first step builds the heap in linear time. We then perform $N - 1$ *DeleteMax*es by swapping the last element in the heap with the first, decrementing the heap size, and percolating down. When the algorithm terminates, the array contains the elements in sorted order. For instance, consider the input sequence 31, 41, 59, 26, 53, 58, 97. The resulting heap is shown in Figure 7.6.

Figure 7.7 shows the heap that results after the first *DeleteMax*. As the figures imply, the last element in the heap is 31; 97 has been placed in a part of the heap array that is technically no longer part of the heap. After 5 more *DeleteMax*

Figure 7.6 (*Max*) heap after *BuildHeap* phase

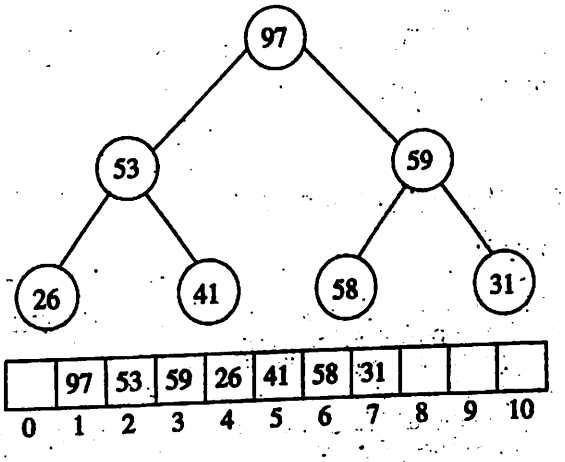
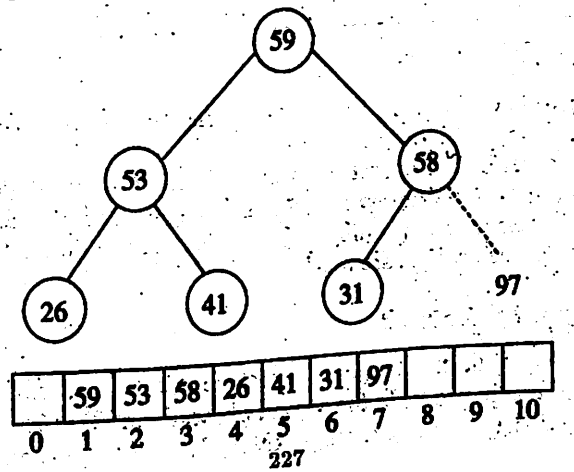


Figure 7.7 Heap after first *DeleteMax*



```
        #define LeftChild( i ) ( 2 * ( i ) + 1 )

        void
        PercDown( ElementType A[ ], int i, int N )
        {
            int Child;
            ElementType Tmp;

/* 1*/      for( Tmp = A[ i ]; LeftChild( i ) < N; i = Child )
            {
/* 2*/          Child = LeftChild( i );
/* 3*/          if( Child != N - 1 && A[ Child + 1 ] > A[ Child ] )
/* 4*/              Child++;
/* 5*/          if( Tmp < A[ Child ] )
/* 6*/              A[ i ] = A[ Child ];
                else
/* 7*/              break;
            }
/* 8*/      A[ i ] = Tmp;
        }

        void
        Heapsort( ElementType A[ ], int N )
        {
            int i;

/* 1*/      for( i = N / 2; i >= 0; i-- ) /* BuildHeap */
/* 2*/          PercDown( A, i, N );
/* 3*/      for( i = N - 1; i > 0; i-- )
            {
/* 4*/          Swap( &A[ 0 ], &A[ i ] ); /* DeleteMax */
/* 5*/          PercDown( A, 0, i );
            }
        }
```

Figure 7.8 Heapsort

operations, the heap will actually have only one element, but the elements left in the heap array will be in sorted order.

The code to perform heapsort is given in Figure 7.8. The slight complication is that, unlike the binary heap, where the data begin at array index 1, the array for heapsort contains data in position 0. Thus the code is a little different from the binary heap code. The changes are minor.

7.5.1. Analysis of Heapsort

As we saw in Chapter 6, the first phase, which constitutes the building of the heap, uses at most $2N$ comparisons. In the second phase, the ith *DeleteMax* uses at most $2\lfloor \log i \rfloor$ comparisons, for a total of at most $2N \log N - O(N)$ comparisons (assuming

$N \geq 2$). Consequently, in the worst case, at most $2N \log N - O(N)$ comparisons are used by heapsort. Exercise 7.12(b) asks you to show that it is possible for all of the *DeleteMax* operations to achieve their worst case simultaneously.

Experiments have shown that heapsort is an extremely stable algorithm: On average it uses only slightly fewer comparisons than the worst-case bound suggests. Until recently, however, nobody had been able to show nontrivial bounds on heapsort's average running time. The problem, it seems, is that successive *DeleteMax* operations destroy the heap's randomness, making the probability arguments very complex. Recently another approach proved successful.

THEOREM 7.5.

The average number of comparisons used to heapsort a random permutation of N distinct items is $2N \log N - O(N \log \log N)$.

PROOF:

The heap construction phase uses $\Theta(N)$ comparisons on average, and so we only need to prove the bound for the second phase. We assume a permutation of $\{1, 2, \ldots, N\}$.

Suppose the ith *DeleteMax* pushes the root element down d_i levels. Then it uses $2d_i$ comparisons. For heapsort on any input, there is a cost sequence $D : d_1, d_2, \ldots, d_N$ that defines the cost of phase 2. That cost is given by $M_D = \sum_{i=1}^{N} d_i$; the number of comparisons used is thus $2M_D$.

Let $f(N)$ be the number of heaps of N items. One can show (Exercise 7.42) that $f(N) > (N/(4e))^N$ (where $e = 2.71828\ldots$). We will show that only an exponentially small fraction of these heaps (in particular $(N/16)^N$) have a cost smaller than $M = N(\log N - \log \log N - 4)$. When this is shown, it follows that the average value of M_D is at least M minus a term that is $o(1)$, and thus the average number of comparisons is at least $2M$. Consequently, our basic goal is to show that there are very few heaps that have small cost sequences.

Because level d_i has at most 2^{d_i} nodes, there are 2^{d_i} possible places that the root element can go for any d_i. Consequently, for any sequence D, the number of distinct corresponding *DeleteMax* sequences is at most

$$S_D = 2^{d_1} 2^{d_2} \cdots 2^{d_N}$$

A simple algebraic manipulation shows that *for a given sequence D*

$$S_D = 2^{M_D}$$

Because each d_i can assume any value between 1 and $\lfloor \log N \rfloor$, there are at most $(\log N)^N$ possible sequences D. It follows that the number of distinct *DeleteMax* sequences that require cost exactly M is at most the number of cost sequences of total cost M times the number of *DeleteMax* sequences for each of these cost sequences. A bound of $(\log N)^N 2^M$ follows immediately.

The total number of heaps with cost sequence less than M is at most

$$\sum_{i=1}^{M-1} (\log N)^N 2^i < (\log N)^N 2^M$$

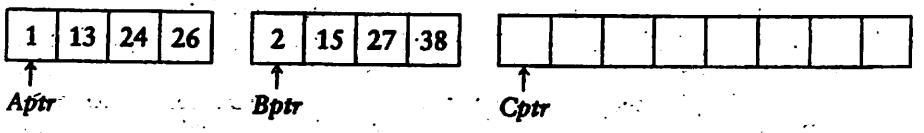
If we choose $M = N(\log N - \log\log N - 4)$, then the number of heaps that have cost sequence less than M is at most $(N/16)^N$, and the theorem follows from our earlier comments.

Using a more complex argument, it can be shown that heapsort always uses at least $N \log N - O(N)$ comparisons, and that there are inputs that can achieve this bound. It seems that the average case should also be $2N \log N - O(N)$ comparisons (rather than the more linear second term in Theorem 7.5); whether this is provable (or even true) is open.

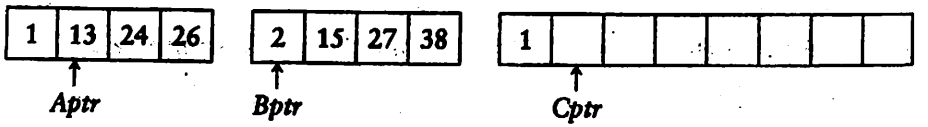
7.6. Mergesort

We now turn our attention to *mergesort*. Mergesort runs in $O(N \log N)$ worst-case running time, and the number of comparisons used is nearly optimal. It is a fine example of a recursive algorithm.

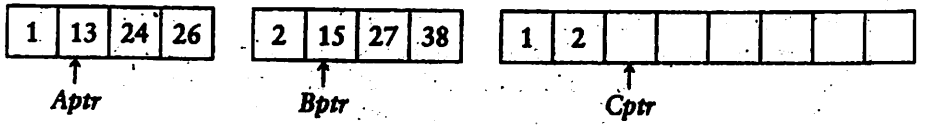
The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list. The basic merging algorithm takes two input arrays A and B, an output array C, and three counters, *Aptr*, *Bptr*, and *Cptr*, which are initially set to the beginning of their respective arrays. The smaller of $A[Aptr]$ and $B[Bptr]$ is copied to the next entry in C, and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to C. An example of how the merge routine works is provided for the following input.



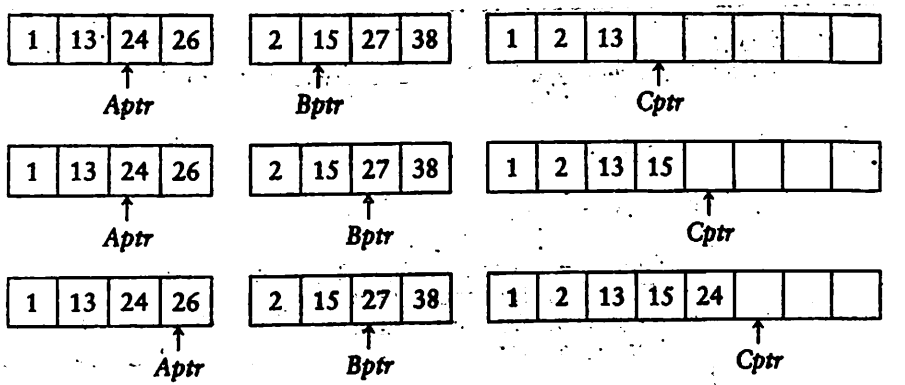
If the array A contains 1, 13, 24, 26, and B contains 2, 15, 27, 38, then the algorithm proceeds as follows: First, a comparison is done between 1 and 2. 1 is added to C, and then 13 and 2 are compared.



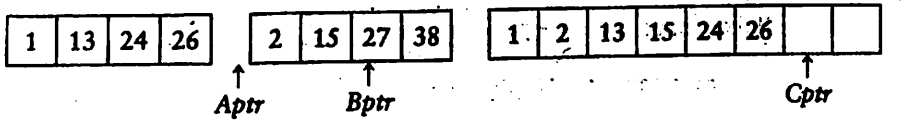
2 is added to C, and then 13 and 15 are compared.



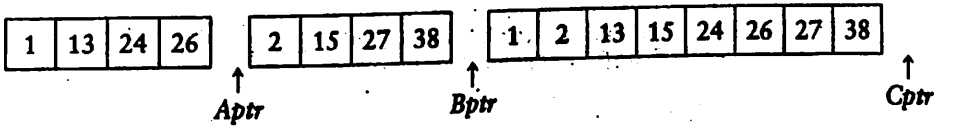
13 is added to C, and then 24 and 15 are compared. This proceeds until 26 and 27 are compared.



26 is added to C, and the A array is exhausted.



The remainder of the B array is then copied to C.



The time to merge two sorted lists is clearly linear, because at most $N - 1$ comparisons are made, where N is the total number of elements. To see this, note that every comparison adds an element to C, except the last comparison, which adds at least two.

The mergesort algorithm is therefore easy to describe. If $N = 1$, there is only one element to sort, and the answer is at hand. Otherwise, recursively mergesort the first half and the second half. This gives two sorted halves, which can then be merged together using the merging algorithm described above. For instance, to sort the eight-element array 24, 13, 26, 1, 2, 27, 38, 15, we recursively sort the first four and last four elements, obtaining 1, 13, 24, 26, 2, 15, 27, 38. Then we merge the two halves as above, obtaining the final list 1, 2, 13, 15, 24, 26, 27, 38. This algorithm is a classic divide-and-conquer strategy. The problem is *divided* into smaller problems and solved recursively. The *conquering* phase consists of patching together the answers. Divide-and-conquer is a very powerful use of recursion that we will see many times.

An implementation of mergesort is provided in Figure 7.9. The procedure called *Mergesort* is just a driver for the recursive routine *MSort*.

The *Merge* routine is subtle. If a temporary array is declared locally for each recursive call of *Merge*, then there could be $\log N$ temporary arrays active at any point. This could be fatal on a machine with small memory. On the other hand, if the

```
void
MSort( ElementType A[ ], ElementType TmpArray[ ],
                int Left, int Right )
{
    int Center;

    if( Left < Right )
    {
        Center = ( Left + Right ) / 2;
        MSort( A, TmpArray, Left, Center );
        MSort( A, TmpArray, Center + 1, Right );
        Merge( A, TmpArray, Left, Center + 1, Right );
    }
}

void
Mergesort( ElementType A[ ], int N )
{
    ElementType *TmpArray;

    TmpArray = malloc( N * sizeof( ElementType ) );
    if( TmpArray != NULL )
    {
        MSort( A, TmpArray, 0, N - 1 );
        free( TmpArray );
    }
    else
        FatalError( "No space for tmp array!!!" );
}
```

Figure 7.9 Mergesort routine

merge routine dynamically allocates and frees the minimum amount of temporary memory, considerable time will be used by *malloc*. A close examination shows that since *Merge* is the last line of *MSort*, there only needs to be one temporary array active at any point. Further, we can use any part of the temporary array; we will use the same portion as the input array A. This allows the improvement described at the end of this section. Figure 7.10 implements the *Merge* routine.

7.6.1. *Analysis of Mergesort*

Mergesort is a classic example of the techniques used to analyze recursive routines: we have to write a recurrence relation for the running time. We will assume that N is a power of 2, so that we always split into even halves. For $N = 1$, the time to mergesort is constant, which we will denote by 1. Otherwise, the time to mergesort N numbers is equal to the time to do two recursive mergesorts of size $N/2$, plus the


```
/* Lpos = start of left half, Rpos = start of right half */

void
Merge( ElementType A[ ], ElementType TmpArray[ ],
                int Lpos, int Rpos, int RightEnd )
{
    int i, LeftEnd, NumElements, TmpPos;

    LeftEnd = Rpos - 1;
    TmpPos = Lpos;
    NumElements = RightEnd - Lpos + 1;

    /* main loop */
    while( Lpos <= LeftEnd && Rpos <= RightEnd )
        if( A[ Lpos ] <= A[ Rpos ] )
            TmpArray[ TmpPos++ ] = A[ Lpos++ ];
        else
            TmpArray[ TmpPos++ ] = A[ Rpos++ ];

    while( Lpos <= LeftEnd )  /* Copy rest of first half */
        TmpArray[ TmpPos++ ] = A[ Lpos++ ];
    while( Rpos <= RightEnd ) /* Copy rest of second half */
        TmpArray[ TmpPos++ ] = A[ Rpos++ ];

    /* Copy TmpArray back */
    for( i = 0; i < NumElements; i++, RightEnd-- )
        A[ RightEnd ] = TmpArray[ RightEnd ];
}
```

Figure 7.10 *Merge* routine

time to merge, which is linear. The following equations say this exactly:

$$T(1) = 1$$

$$T(N) = 2T(N/2) + N$$

This is a standard recurrence relation, which can be solved several ways. We will show two methods. The first idea is to divide the recurrence relation through by N. The reason for doing this will become apparent soon. This yields

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

This equation is valid for any N that is a power of 2, so we may also write

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

and

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + 1$$

$$\vdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Now add up all the equations. This means that we add all of the terms on the left-hand side and set the result equal to the sum of all of the terms on the right-hand side. Observe that the term $T(N/2)/(N/2)$ appears on both sides and thus cancels. In fact, virtually all the terms appear on both sides and cancel. This is called *telescoping* a sum. After everything is added, the final result is

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \log N$$

because all of the other terms cancel and there are $\log N$ equations, and so all the 1s at the end of these equations add up to $\log N$. Multiplying through by N gives the final answer.

$$T(N) = N \log N + N = O(N \log N)$$

Notice that if we did not divide through by N at the start of the solutions, the sum would not telescope. This is why it was necessary to divide through by N.

An alternative method is to substitute the recurrence relation continually on the right-hand side. We have

$$T(N) = 2T(N/2) + N$$

Since we can substitute $N/2$ into the main equation,

$$2T(N/2) = 2(2(T(N/4)) + N/2) = 4T(N/4) + N$$

we have

$$T(N) = 4T(N/4) + 2N$$

Again, by substituting $N/4$ into the main equation, we see that

$$4T(N/4) = 4(2T(N/8)) + N/4) = 8T(N/8) + N$$

So we have

$$T(N) = 8T(N/8) + 3N$$

Continuing in this manner, we obtain

$$T(N) = 2^k T(N/2^k) + k \cdot N$$

Using $k = \log N$, we obtain

$$T(N) = NT(1) + N \log N = N \log N + N$$

The choice of which method to use is a matter of taste. The first method tends to produce scrap work that fits better on a standard, $8\frac{1}{2} \times 11$ sheet of paper, leading to fewer mathematical errors, but it requires a certain amount of experience to apply. The second method is more of a brute force approach.

Recall that we have assumed $N = 2^k$. The analysis can be refined to handle cases when N is not a power of 2. The answer turns out to be almost identical (this is usually the case).

Although mergesort's running time is $O(N \log N)$, it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory, and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. This copying can be avoided by judiciously switching the roles of A and *TmpArray* at alternate levels of the recursion. A variant of mergesort can also be implemented nonrecursively (Exercise 7.14), but even so, for serious internal sorting applications, the algorithm of choice is quicksort, which is described in the next section. Nevertheless, as we will see later in this chapter, the merging routine is the cornerstone of most external sorting algorithms.

7.7. Quicksort

As its name implies, *quicksort* is the fastest known sorting algorithm in practice. Its average running time is $O(N \log N)$. It is very fast, mainly due to a very tight and highly optimized inner loop. It has $O(N^2)$ worst-case performance, but this can be made exponentially unlikely with a little effort. The quicksort algorithm is simple to understand and prove correct, although for many years it had the reputation of being an algorithm that could in theory be highly optimized but in practice was impossible to code correctly. Like mergesort, quicksort is a divide-and-conquer recursive algorithm. The basic algorithm to sort an array S consists of the following four easy steps:

1. If the number of elements in S is 0 or 1, then return.
2. Pick any element v in S. This is called the *pivot*.
3. *Partition* $S - \{v\}$ (the remaining elements in S) into two disjoint groups: $S_1 = \{x \in S - \{v\} | x \leq v\}$, and $S_2 = \{x \in S - \{v\} | x \geq v\}$.
4. Return {quicksort(S_1) followed by v followed by quicksort(S_2)}.

Since the partition step ambiguously describes what to do with elements equal to the pivot, this becomes a design decision. Part of a good implementation is handling this case as efficiently as possible. Intuitively, we would hope that about half the keys that are equal to the pivot go into S_1 and the other half into S_2, much as we like binary search trees to be balanced.

Figure 7.11 shows the action of quicksort on a set of numbers. The pivot is chosen (by chance) to be 65. The remaining elements in the set are partitioned into two smaller sets. Recursively sorting the set of smaller numbers yields 0, 13, 26, 31, 43, 57 (by rule 3 of recursion). The set of large numbers is similarly sorted. The sorted arrangement of the entire set is then trivially obtained.

It should be clear that this algorithm works, but it is not clear why it is any faster than mergesort. Like mergesort, it recursively solves two subproblems and requires linear additional work (step 3), but, unlike mergesort, the subproblems are not guaranteed to be of equal size, which is potentially bad. The reason that quicksort is faster is that the partitioning step can actually be performed in place

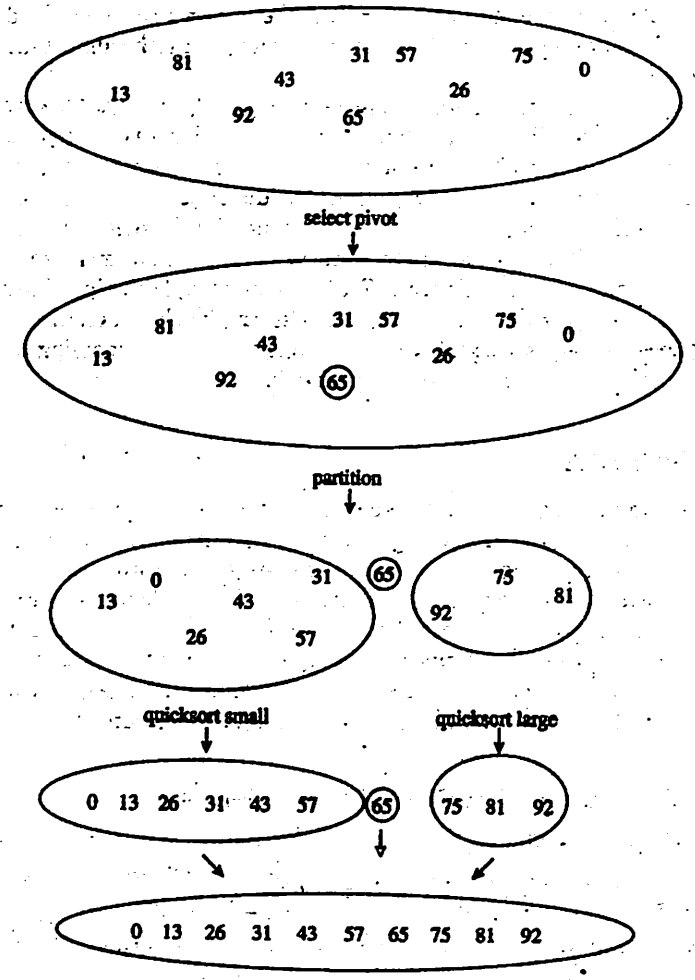


Figure 7.11 The steps of quicksort illustrated by example

and very efficiently. This efficiency more than makes up for the lack of equal-sized recursive calls.

The algorithm as described so far lacks quite a few details, which we now fill in. There are many ways to implement steps 2 and 3; the method presented here is the result of extensive analysis and empirical study and represents a very efficient way to implement quicksort. Even the slightest deviations from this method can cause surprisingly bad results.

7.7.1. *Picking the Pivot*

Although the algorithm as described works no matter which element is chosen as pivot, some choices are obviously better than others.

A Wrong Way

The popular, uninformed choice is to use the first element as the pivot. This is acceptable if the input is random, but if the input is presorted or in reverse order, then the pivot provides a poor partition, because either all the elements go into S_1 or they go into S_2. Worse, this happens consistently throughout the recursive calls. The practical effect is that if the first element is used as the pivot and the input is presorted, then quicksort will take quadratic time to do essentially nothing at all, which is quite embarrassing. Moreover, presorted input (or input with a large presorted section) is quite frequent, so using the first element as pivot *is an absolutely horrible idea* and should be discarded immediately. An alternative is choosing the larger of the first two distinct keys as pivot, but this has the same bad properties as merely choosing the first key. Do not use that pivoting strategy either.

A Safe Maneuver

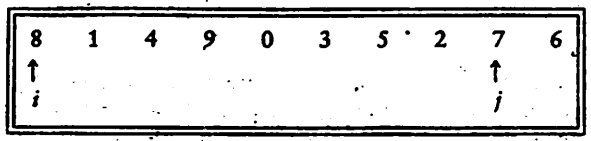
A safe course is merely to choose the pivot randomly. This strategy is generally perfectly safe, unless the random number generator has a flaw (which is not as uncommon as you might think), since it is very unlikely that a random pivot would consistently provide a poor partition. On the other hand, random number generation is generally an expensive commodity and does not reduce the average running time of the rest of the algorithm at all.

Median-of-Three Partitioning

The median of a group of N numbers is the $\lceil N/2 \rceil$ th largest number. The best choice of pivot would be the median of the array. Unfortunately, this is hard to calculate and would slow down quicksort considerably. A good estimate can be obtained by picking three elements randomly and using the median of these three as pivot. The randomness turns out not to help much, so the common course is to use as pivot the median of the left, right, and center elements. For instance, with input 8, 1, 4, 9, 6, 3, 5, 2, 7, 0 as before, the left element is 8, the right element is 0 and the center (in position $\lfloor (Left + Right)/2 \rfloor$) element is 6. Thus, the pivot would be $v = 6$. Using median-of-three partitioning clearly eliminates the bad case for sorted input (the partitions become equal in this case) and actually reduces the running time of quicksort by about 5 percent.

7.7.2. *Partitioning Strategy*

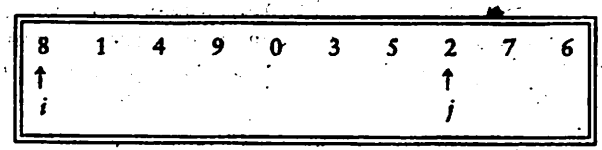
There are several partitioning strategies used in practice, but the one described here is known to give good results. It is very easy, as we shall see, to do this wrong or inefficiently, but it is safe to use a known method. The first step is to get the pivot element out of the way by swapping it with the last element. i starts at the first element and j starts at the next-to-last element. If the original input was the same as before, the following figure shows the current situation.



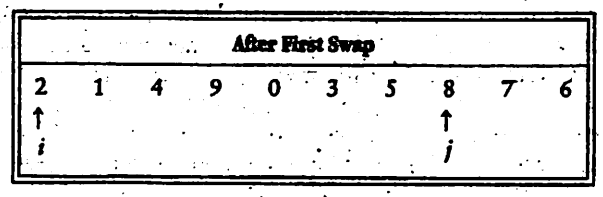
For now we will assume that all the elements are distinct. Later on we will worry about what to do in the presence of duplicates. As a limiting case, our algorithm must do the proper thing if *all* of the elements are identical. It is surprising how easy it is to do the *wrong* thing.

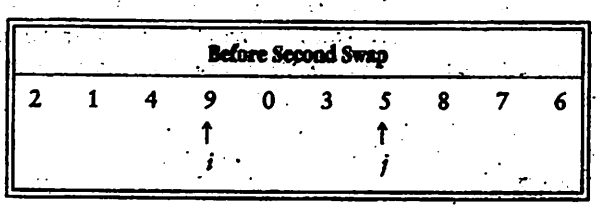
What our partitioning stage wants to do is to move all the small elements to the left part of the array and all the large elements to the right part. "Small" and "large" are, of course, relative to the pivot.

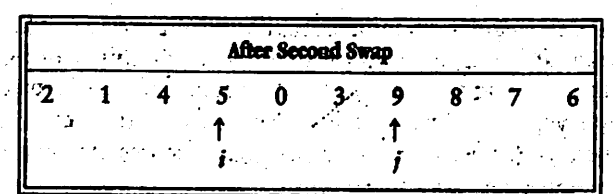
While i is to the left of j, we move i right, skipping over elements that are smaller than the pivot. We move j left, skipping over elements that are larger than the pivot. When i and j have stopped, i is pointing at a large element and j is pointing at a small element. If i is to the left of j, those elements are swapped. The effect is to push a large element to the right and a small element to the left. In the example above, i would not move and j would slide over one place. The situation is as follows.

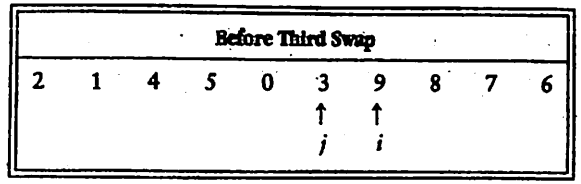


We then swap the elements pointed to by i and j and repeat the process until i and j cross.

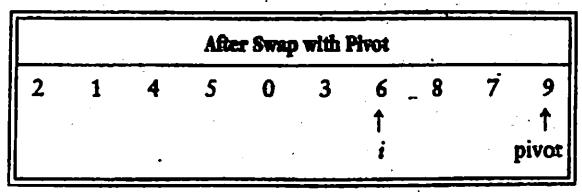








At this stage, i and j have crossed, so no swap is performed. The final part of the partitioning is to swap the pivot element with the element pointed to by i.



When the pivot is swapped with i in the last step, we know that every element in a position $P < i$ must be small. This is because either position P contained a small element to start with, or the large element originally in position P was replaced during a swap. A similar argument shows that elements in positions $P > i$ must be large.

One important detail we must consider is how to handle keys that are equal to the pivot. The questions are whether or not i should stop when it sees a key equal to the pivot and whether or not j should stop when it sees a key equal to the pivot. Intuitively, i and j ought to do the same thing, since otherwise the partitioning step is biased. For instance, if i stops and j does not, then all keys that are equal to the pivot will wind up in S_2.

To get an idea of what might be good, we consider the case where all the keys in the array are identical. If both i and j stop, there will be many swaps between identical elements. Although this seems useless, the positive effect is that i and j will cross in the middle, so when the pivot is replaced, the partition creates two nearly equal subarrays. The mergesort analysis tells us that the total running time would then be $O(N \log N)$.

If neither i nor j stops, and code is present to prevent them from running off the end of the array, no swaps will be performed. Although this seems good, a correct implementation would then swap the pivot into the last spot that i touched, which would be the next-to-last position (or last, depending on the exact implementation). This would create very uneven subarrays. If all the keys are identical, the running time is $O(N^2)$. The effect is the same as using the first element as a pivot for presorted input. It takes quadratic time to do nothing!

Thus, we find that it is better to do the unnecessary swaps and create even subarrays than to risk wildly uneven subarrays. Therefore, we will have both i and j stop if they encounter a key equal to the pivot. This turns out to be the only one of the four possibilities that does not take quadratic time for this input.

At first glance it may seem that worrying about an array of identical elements is silly. After all, why would anyone want to sort 5,000 identical elements? However,

recall that quicksort is recursive. Suppose there are 100,000 elements, of which 5,000 are identical. Eventually, quicksort will make the recursive call on only these 5,000 elements. Then it really will be important to make sure that 5,000 identical elements can be sorted efficiently.

7.7.3. Small Arrays

For very small arrays ($N \leq 20$), quicksort does not perform as well as insertion sort. Furthermore, because quicksort is recursive, these cases will occur frequently. A common solution is not to use quicksort recursively for small arrays, but instead use a sorting algorithm that is efficient for small arrays, such as insertion sort. Using this strategy can actually save about 15 percent in the running time (over doing no cutoff at all). A good cutoff range is $N = 10$, although any cutoff between 5 and 20 is likely to produce similar results. This also saves nasty degenerate cases, such as taking the median of three elements when there are only one or two.

7.7.4. Actual Quicksort Routines

The driver for quicksort is shown in Figure 7.12.

The general form of the routines will be to pass the array and the range of the array (*Left* and *Right*) to be sorted. The first routine to deal with is pivot selection. The easiest way to do this is to sort *A*[*Left*], *A*[*Right*], and *A*[*Center*] in place. This has the extra advantage that the smallest of the three winds up in *A*[*Left*], which is where the partitioning step would put it anyway. The largest winds up in *A*[*Right*], which is also the correct place, since it is larger than the pivot. Therefore, we can place the pivot in *A*[*Right* − 1] and initialize *i* and *j* to *Left* + 1 and *Right* − 2 in the partition phase. Yet another benefit is that because *A*[*Left*] is smaller than the pivot, it will act as a sentinel for *j*. Thus, we do not need to worry about *j* running past the end. Since *i* will stop on keys equal to the pivot, storing the pivot in *A*[*Right* − 1] provides a sentinel for *i*. The code in Figure 7.13 does the median-of-three partitioning with all the side effects described. It may seem that it is only slightly inefficient to compute the pivot by a method that does not actually sort *A*[*Left*], *A*[*Center*], and *A*[*Right*], but, surprisingly, this produces bad results (see Exercise 7.38).

The real heart of the quicksort routine is in Figure 7.14. It includes the partitioning and recursive calls. There are several things worth noting in this

Figure 7.12 Driver for quicksort

```
void
Quicksort( ElementType A[ ], int N )
{
    Qsort( A, 0, N - 1 );
}
```



```
/* Return median of Left, Center, and Right */
/* Order these and hide the pivot */

ElementType
Median3( ElementType A[ ], int Left, int Right )
{
    int Center = ( Left + Right ) / 2;

    if( A[ Left ] > A[ Center ] )
        Swap( &A[ Left ], &A[ Center ] );
    if( A[ Left ] > A[ Right ] )
        Swap( &A[ Left ], &A[ Right ] );
    if( A[ Center ] > A[ Right ] )
        Swap( &A[ Center ], &A[ Right ] );

    /* Invariant: A[ Left ] <= A[ Center ] <= A[ Right ] */

    Swap( &A[ Center ], &A[ Right - 1 ] );  /* Hide pivot */
    return A[ Right - 1 ];                  /* Return pivot */
}
```

Figure 7.13 Code to perform median-of-three partitioning

implementation. Line 3 initializes *i* and *j* to 1 past their correct values, so that there are no special cases to consider. This initialization depends on the fact that median-of-three partitioning has some side effects; this program will not work if you try to use it without change with a simple pivoting strategy, because *i* and *j* start in the wrong place and there is no longer a sentinel for *j*.

The *Swap* at line 8 is sometimes written explicitly, for speed purposes. For the algorithm to be fast, it is necessary to force the compiler to compile this code in-line. Many compilers will do this automatically, if asked to, but for those that do not the difference can be significant.

Finally, lines 5 and 6 show why quicksort is so fast. The inner loop of the algorithm consists of an increment/decrement (by 1, which is fast), a test, and a jump. There is no extra juggling as there is in mergesort. This code is still surprisingly tricky. It is tempting to replace lines 3 through 9 with the statements in Figure 7.15. This does not work, because there would be an infinite loop if $A[i] = A[j] = Pivot$.

7.7.5. Analysis of Quicksort

Like mergesort, quicksort is recursive, and hence, its analysis requires solving a recurrence formula. We will do the analysis for a quicksort, assuming a random pivot (no median-of-three partitioning) and no cutoff for small files. We will take $T(0) = T(1) = 1$, as in mergesort. The running time of quicksort is equal to the running time of the two recursive calls plus the linear time spent in the partition (the

```
        #define Cutoff ( 3 )

        void
        Qsort( ElementType A[ ], int Left, int Right )
        {
            int i, j;
            ElementType Pivot;

/* 1*/      if( Left + Cutoff <= Right )
            {
/* 2*/          Pivot = Median3( A, Left, Right );
/* 3*/          i = Left; j = Right - 1;
/* 4*/          for( ; ; )
                {
/* 5*/              while( A[ ++i ] < Pivot ){ }
/* 6*/              while( A[ --j ] > Pivot ){ }
/* 7*/              if( i < j )
/* 8*/                  Swap( &A[ i ], &A[ j ] );
                    else
/* 9*/                  break;
                }
/*10*/          Swap( &A[ i ], &A[ Right - 1 ] ); /* Restore pivot

/*11*/          Qsort( A, Left, i - 1 );
/*12*/          Qsort( A, i + 1, Right );
            }
            else /* Do an insertion sort on the subarray */
/*13*/          InsertionSort( A + Left, Right - Left + 1 );
        }
```

Figure 7.14 Main quicksort routine

```
/* 3*/  i = Left + 1; j = Right - 2;
/* 4*/  for( ; ; )
        {
/* 5*/      while( A[ i ] < Pivot ) i++;
/* 6*/      while( A[ j ] > Pivot ) j--;
/* 7*/      if( i < j )
/* 8*/          Swap( &A[ i ], &A[ j ] );
            else
/* 9*/          break;
        }
```

Figure 7.15 A small change to quicksort, which breaks the algorithm

pivot selection takes only constant time). This gives the basic quicksort relation

$$T(N) = T(i) + T(N - i - 1) + cN \tag{7.1}$$

where $i = |S_1|$ is the number of elements in S_1. We will look at three cases.

Worst-Case Analysis

The pivot is the smallest element, all the time. Then $i = 0$ and if we ignore $T(0) = 1$, which is insignificant, the recurrence is

$$T(N) = T(N - 1) + cN, \quad N > 1 \tag{7.2}$$

We telescope, using Equation (7.2) repeatedly. Thus

$$T(N - 1) = T(N - 2) + c(N - 1) \tag{7.3}$$

$$T(N - 2) = T(N - 3) + c(N - 2) \tag{7.4}$$

$$\vdots$$

$$T(2) = T(1) + c(2) \tag{7.5}$$

Adding up all these equations yields

$$T(N) = T(1) + c\sum_{i=2}^{N} i = O(N^2) \tag{7.6}$$

as claimed earlier.

Best-Case Analysis

In the best case, the pivot is in the middle. To simplify the math, we assume that the two subarrays are each exactly half the size of the original, and although this gives a slight overestimate, this is acceptable because we are only interested in a Big-Oh answer.

$$T(N) = 2T(N/2) + cN \tag{7.7}$$

Divide both sides of Equation (7.7) by N.

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + c \tag{7.8}$$

We will telescope using this equation.

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + c \tag{7.9}$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + c \tag{7.10}$$

$$\vdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c \tag{7.11}$$

We add all the equations from (7.7) to (7.11) and note that there are $\log N$ of them:

$$\frac{T(N)}{N} = \frac{T(1)}{1} + c \log N \tag{7.12}$$

which yields

$$T(N) = cN \log N + N = O(N \log N) \tag{7.13}$$

Notice that this is the exact same analysis as mergesort, hence we get the same answer.

Average-Case Analysis

This is the most difficult part. For the average case, we assume that each of the file sizes for S_1 is equally likely, and hence has probability $1/N$. This assumption is actually valid for our pivoting and partitioning strategy, but it is not valid for some others. Partitioning strategies that do not preserve the randomness of the subfiles cannot use this analysis. Interestingly, these strategies seem to result in programs that take longer to run in practice.

With this assumption, the average value of $T(i)$, and hence $T(N-i-1)$, is $(1/N)\sum_{j=0}^{N-1} T(j)$. Equation (7.1) then becomes

$$T(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} T(j)\right] + cN \tag{7.14}$$

If Equation (7.14) is multiplied by N, it becomes

$$NT(N) = 2\left[\sum_{j=0}^{N-1} T(j)\right] + cN^2 \tag{7.15}$$

We need to remove the summation sign to simplify matters. We note that we can telescope with one more equation.

$$(N-1)T(N-1) = 2\left[\sum_{j=0}^{N-2} T(j)\right] + c(N-1)^2 \tag{7.16}$$

If we subtract (7.16) from (7.15), we obtain

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c \tag{7.17}$$

We rearrange terms and drop the insignificant $-c$ on the right, obtaining

$$NT(N) = (N+1)T(N-1) + 2cN \tag{7.18}$$

We now have a formula for $T(N)$ in terms of $T(N-1)$ only. Again the idea is to telescope, but Equation (7.18) is in the wrong form. Divide Equation (7.18) by $N(N+1)$:

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1} \tag{7.19}$$

Now we can telescope.

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2c}{N} \tag{7.20}$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2c}{N-1} \tag{7.21}$$

$$\vdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3} \tag{7.22}$$

Adding Equations (7.19) through (7.22) yields

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{N+1}\frac{1}{i} \tag{7.23}$$

The sum is about $\log_e(N+1) + \gamma - \frac{3}{2}$, where $\gamma \approx 0.577$ is known as Euler's constant, so

$$\frac{T(N)}{N+1} = O(\log N) \tag{7.24}$$

And so

$$T(N) = O(N \log N) \tag{7.25}$$

Although this analysis seems complicated, it really is not—the steps are natural once you have seen some recurrence relations. The analysis can actually be taken further. The highly optimized version that was described above has also been analyzed, and this result gets extremely difficult, involving complicated recurrences and advanced mathematics. The effect of equal keys has also been analyzed in detail, and it turns out that the code presented does the right thing.

7.7.6. *A Linear-Expected-Time Algorithm for Selection*

Quicksort can be modified to solve the *selection problem*, which we have seen in Chapters 1 and 6. Recall that by using a priority queue, we can find the kth largest (or smallest) element in $O(N + k \log N)$. For the special case of finding the median, this gives an $O(N \log N)$ algorithm.

Since we can sort the array in $O(N \log N)$ time, one might expect to obtain a better time bound for selection. The algorithm we present to find the kth smallest element in a set S is almost identical to quicksort. In fact, the first three steps are the same. We will call this algorithm *quickselect*. Let $|S_i|$ denote the number of elements in S_i. The steps of quickselect are

1. If $|S| = 1$, then $k = 1$ and return the element in S as the answer. If a cutoff for small arrays is being used and $|S| \leq CUTOFF$, then sort S and return the kth smallest element.
2. Pick a pivot element, $v \in S$.
3. Partition $S - \{v\}$ into S_1 and S_2, as was done with quicksort.

4. If $k \leq |S_1|$, then the kth smallest element must be in S_1. In this case, return quickselect (S_1, k). If $k = 1 + |S_1|$, then the pivot is the kth smallest element and we can return it as the answer. Otherwise, the kth smallest element lies in S_2, and it is the $(k - |S_1| - 1)$st smallest element in S_2. We make a recursive call and return quickselect $(S_2, k - |S_1| - 1)$.

In contrast to quicksort, quickselect makes only one recursive call instead of two. The worst case of quickselect is identical to that of quicksort and is $O(N^2)$. Intuitively, this is because quicksort's worst case is when one of S_1 and S_2 is empty; thus, quickselect is not really saving a recursive call. The average running time, however, is $O(N)$. The analysis is similar to quicksort's and is left as an exercise.

The implementation of quickselect is even simpler than the abstract description might imply. The code to do this is shown in Figure 7.16. When the algorithm

Figure 7.16 Main quickselect routine

```
        /* Places the kth smallest element in the kth position */
        /* Because arrays start at 0, this will be index k-1 */
        void
        Qselect( ElementType A[ ], int k, int Left, int Right )
        {
            int i, j;
            ElementType Pivot;

/* 1*/      if( Left + Cutoff <= Right )
            {
/* 2*/          Pivot = Median3( A, Left, Right );
/* 3*/          i = Left; j = Right - 1;
/* 4*/          for( ; ; )
                {
/* 5*/              while( A[ ++i ] < Pivot ){ }
/* 6*/              while( A[ --j ] > Pivot ){ }
/* 7*/              if( i < j )
/* 8*/                  Swap( &A[ i ], &A[ j ] );
                    else
/* 9*/                  break;
                }
/*10*/          Swap( &A[ i ], &A[ Right - 1 ] ); /* Restore pivot */

/*11*/          if( k <= i )
/*12*/              Qselect( A, k, Left, i - 1 );
/*13*/          else if( k > i + 1 )
/*14*/              Qselect( A, k, i + 1, Right );
            }
            else  /* Do an insertion sort on the subarray */
/*15*/          InsertionSort( A + Left, Right - Left + 1 );
        }
```


terminates, the kth smallest element is in position k. This destroys the original ordering; if this is not desirable, then a copy must be made.

Using a median-of-three pivoting strategy makes the chance of the worst case occurring almost negligible. By carefully choosing the pivot, however, we can eliminate the quadratic worst case and ensure an $O(N)$ algorithm. The overhead involved in doing this is considerable, so the resulting algorithm is mostly of theoretical interest. In Chapter 10, we will examine the linear-time worst-case algorithm for selection, and we shall also see an interesting technique of choosing the pivot that results in a somewhat faster selection algorithm in practice.

7.8. Sorting Large Structures

Throughout our discussion of sorting, we have assumed that the elements to be sorted are simply integers. Frequently, we need to sort large structures by a certain key. For instance, we might have payroll records, with each record consisting of a name, address, phone number, financial information such as salary, and tax information. We might want to sort this information by one particular field, such as the name. For all of our algorithms, the fundamental operation is the swap, but here swapping two structures can be a very expensive operation, because the structures are potentially large. If this is the case, a practical solution is to have the input array contain pointers to the structures. We sort by comparing the keys the pointers point to, swapping pointers when necessary. This means that all the data movement is essentially the same as if we were sorting integers. This is known as *indirect sorting*; we can use this technique for most of the data structures we have described. This justifies our assumption that complex structures can be handled without tremendous loss of efficiency.

7.9. A General Lower Bound for Sorting

Although we have $O(N \log N)$ algorithms for sorting, it is not clear that this is as good as we can do. In this section, we prove that any algorithm for sorting that uses only comparisons requires $\Omega(N \log N)$ comparisons (and hence time) in the worst case, so that mergesort and heapsort are optimal to within a constant factor. The proof can be extended to show that $\Omega(N \log N)$ comparisons are required, even on average, for any sorting algorithm that uses only comparisons, which means that quicksort is optimal on average to within a constant factor.

Specifically, we will prove the following result: Any sorting algorithm that uses only comparisons requires $\lceil \log(N!) \rceil$ comparisons in the worst case and $\log(N!)$ comparisons on average. We will assume that all N elements are distinct, since any sorting algorithm must work for this case.

7.9.1. *Decision Trees*

A *decision tree* is an abstraction used to prove lower bounds. In our context, a decision tree is a binary tree. Each node represents a set of possible orderings,

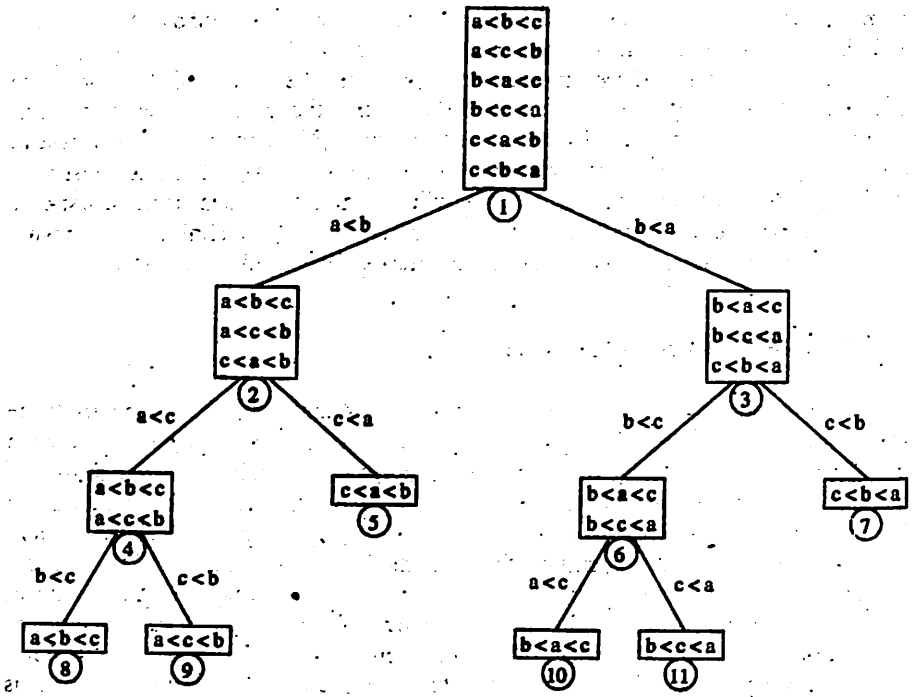


Figure 7.17 A decision tree for three-element insertion sort

consistent with comparisons that have been made, among the elements. The results of the comparisons are the tree edges.

The decision tree in Figure 7.17 represents an algorithm that sorts the three elements a, b, and c. The initial state of the algorithm is at the root. (We will use the terms *state* and *node* interchangeably.) No comparisons have been done, so all orderings are legal. The first comparison that *this particular* algorithm performs compares a and b. The two results lead to two possible states. If $a < b$, then only three possibilities remain. If the algorithm reaches node 2, then it will compare a and c. Other algorithms might do different things; a different algorithm would have a different decision tree. If $a > c$, the algorithm enters state 5. Since there is only one ordering that is consistent, the algorithm can terminate and report that it has completed the sort. If $a < c$, the algorithm cannot do this, because there are two possible orderings and it cannot possibly be sure which is correct. In this case, the algorithm will require one more comparison.

Every algorithm that sorts by using only comparisons can be represented by a decision tree. Of course, it is only feasible to draw the tree for extremely small input sizes. The number of comparisons used by the sorting algorithm is equal to the depth of the deepest leaf. In our case, this algorithm uses three comparisons in the worst case. The average number of comparisons used is equal to the average depth of the leaves. Since a decision tree is large, it follows that there must be some long

paths. To prove the lower bounds, all that needs to be shown are some basic tree properties.

LEMMA 7.1.
Let T be a binary tree of depth d. Then T has at most 2^d leaves.

PROOF:
The proof is by induction. If $d = 0$, then there is at most one leaf, so the basis is true. Otherwise, we have a root, which cannot be a leaf, and a left and right subtree, each of depth at most $d - 1$. By the induction hypothesis, they can each have at most 2^{d-1} leaves, giving a total of at most 2^d leaves. This proves the lemma.

LEMMA 7.2.
A binary tree with L leaves must have depth at least $\lceil \log L \rceil$.

PROOF:
Immediate from the preceding lemma.

THEOREM 7.6.
Any sorting algorithm that uses only comparisons between elements requires at least $\lceil \log(N!) \rceil$ comparisons in the worst case.

PROOF:
A decision tree to sort N elements must have $N!$ leaves. The theorem follows from the preceding lemma.

THEOREM 7.7.
Any sorting algorithm that uses only comparisons between elements requires $\Omega(N \log N)$ comparisons.

PROOF:
From the previous theorem, $\log(N!)$ comparisons are required.

$$\begin{aligned}
\log(N!) &= \log(N(N-1)(N-2)\cdots(2)(1)) \\
&= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\
&\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log N/2 \\
&\geq \frac{N}{2} \log \frac{N}{2} \\
&\geq \frac{N}{2} \log N - \frac{N}{2} \\
&= \Omega(N \log N)
\end{aligned}$$

This type of lower-bound argument, when used to prove a worst-case result, is sometimes known as an *information-theoretic* lower bound. The general theorem says that if there are P different possible cases to distinguish, and the questions are of the form YES/NO, then $\lceil \log P \rceil$ questions are always required in some case by any algorithm to solve the problem. It is possible to prove a similar result for the average-case running time of any comparison-based sorting algorithm. This result is implied by the following lemma, which is left as an exercise: Any binary tree with L leaves has an average depth of at least $\log L$.

7.10. Bucket Sort

Although we proved in the previous section that any general sorting algorithm that uses only comparisons requires $\Omega(N \log N)$ time in the worst case, recall that it is still possible to sort in linear time in some special cases.

A simple example is bucket sort. For bucket sort to work, extra information must be available. The input $A_1, A_2, \ldots, A_N$ must consist of only positive integers smaller than M. (Obviously extensions to this are possible:) If this is the case, then the algorithm is simple: Keep an array called *Count*, of size M, which is initialized to all 0s. Thus, *Count* has M cells, or buckets, which are initially empty. When A_i is read, increment *Count*$[A_i]$ by 1. After all the input is read, scan the *Count* array, printing out a representation of the sorted list. This algorithm takes $O(M + N)$; the proof is left as an exercise. If M is $O(N)$, then the total is $O(N)$.

Although this algorithm seems to violate the lower bound, it turns out that it does not because it uses a more powerful operation than simple comparisons. By incrementing the appropriate bucket, the algorithm essentially performs an M-way comparison in unit time. This is similar to the strategy used in extendible hashing (Section 5.6). This is clearly not in the model for which the lower bound was proven.

This algorithm does, however, question the validity of the model used in proving the lower bound. The model actually is a strong model, because a *general-purpose* sorting algorithm cannot make assumptions about the type of input it can expect to see, but must make decisions based on ordering information only. Naturally, if there is extra information available, we should expect to find a more efficient algorithm, since otherwise the extra information would be wasted.

Although bucket sort seems like much too trivial an algorithm to be useful, it turns out that there are many cases where the input is only small integers, so that using a method like quicksort is really overkill.

7.11. External Sorting

So far, all the algorithms we have examined require that the input fit into main memory. There are, however, applications where the input is much too large to fit into memory. This section will discuss *external sorting* algorithms, which are designed to handle very large inputs.

7.11.1. Why We Need New Algorithms

Most of the internal sorting algorithms take advantage of the fact that memory is directly addressable. Shellsort compares elements $A[i]$ and $A[i - h_k]$ in one time unit. Heapsort compares elements $A[i]$ and $A[i * 2 + 1]$ in one time unit. Quicksort, with median-of-three partitioning, requires comparing $A[Left]$, $A[Center]$, and $A[Right]$ in a constant number of time units. If the input is on a tape, then all these operations lose their efficiency, since elements on a tape can only be accessed sequentially. Even if the data is on a disk, there is still a practical loss of efficiency because of the delay required to spin the disk and move the disk head.

To see how slow external accesses really are, create a random file that is large, but not too big to fit in main memory. Read the file in and sort it using an efficient algorithm. The time it takes to sort the input is certain to be insignificant compared to the time to read the input, even though sorting is an $O(N \log N)$ operation and reading the input is only $O(N)$.

7.11.2. Model for External Sorting

The wide variety of mass storage devices makes external sorting much more device-dependent than internal sorting. The algorithms that we will consider work on tapes, which are probably the most restrictive storage medium. Since access to an element on tape is done by winding the tape to the correct location, tapes can be efficiently accessed only in sequential order (in either direction).

We will assume that we have at least three tape drives to perform the sorting. We need two drives to do an efficient sort; the third drive simplifies matters. If only one tape drive is present, then we are in trouble: any algorithm will require $\Omega(N^2)$ tape accesses.

7.11.3. The Simple Algorithm

The basic external sorting algorithm uses the *Merge* routine from mergesort. Suppose we have four tapes, T_{a1}, T_{a2}, T_{b1}, T_{b2}, which are two input and two output tapes. Depending on the point in the algorithm, the a and b tapes are either input tapes or output tapes. Suppose the data is initially on T_{a1}. Suppose further that the internal memory can hold (and sort) M records at a time. A natural first step is to read M records at a time from the input tape, sort the records internally, and then write the sorted records alternately to T_{b1} and T_{b2}. We will call each set of sorted records a *run*. When this is done, we rewind all the tapes. Suppose we have the same input as our example for Shellsort.

T_{a1}	81	94	11	96	12	35	17	99	28	58	41	75	15
T_{a2}													
T_{b1}													
T_{b2}													

If $M = 3$, then after the runs are constructed, the tapes will contain the data indicated in the following figure.

T_{a1}								
T_{a2}								
T_{b1}	11	81	94	17	28	99	15	
T_{b2}	12	35	96	41	58	75		

Now T_{b1} and T_{b2} contain a group of runs. We take the first run from each tape and merge them, writing the result, which is a run twice as long, onto T_{a1}. Then we take the next run from each tape, merge these, and write the result to T_{a2}. We continue this process, alternating between T_{a1} and T_{a2}, until either T_{b1} or T_{b2} is empty. At this point either both are empty or there is one run left. In the latter case, we copy this run to the appropriate tape. We rewind all four tapes, and repeat the same steps, this time using the a tapes as input and the b tapes as output. This will give runs of $4M$. We continue the process until we get one run of length N.

This algorithm will require $\lceil \log(N/M) \rceil$ passes, plus the initial run-constructing pass. For instance, if we have 10 million records of 128 bytes each, and four megabytes of internal memory, then the first pass will create 320 runs. We would then need nine more passes to complete the sort. Our example requires $\lceil \log 13/3 \rceil = 3$ more passes, which are shown in the following figure.

T_{a1}	11	12	35	81	94	96	15						
T_{a2}	17	28	41	58	75	99							
T_{b1}													
T_{b2}													

T_{a1}													
T_{a2}													
T_{b1}	11	12	17	28	35	51	58	75	81	94	96	99	
T_{b2}	15												

T_{a1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{a2}													
T_{b1}													
T_{b2}													

7.11.4. *Multiway Merge*

If we have extra tapes, then we can expect to reduce the number of passes required to sort our input. We do this by extending the basic (two-way) merge to a k-way merge.

Merging two runs is done by winding each input tape to the beginning of each run. Then the smaller element is found, placed on an output tape, and the appropriate input tape is advanced. If there are k input tapes, this strategy works the same way, the only difference being that it is slightly more complicated to find the smallest of the k elements. We can find the smallest of these elements by using a priority queue. To obtain the next element to write on the output tape, we perform a *DeleteMin* operation. The appropriate input tape is advanced, and if the run on the input tape is not yet completed, we *Insert* the new element into the priority queue. Using the same example as before, we distribute the input onto the three tapes.

T_{a1}								
T_{a2}								
T_{a3}								
T_{b1}	11	81	94	41	58	75		
T_{b2}	12	35	96	15				
T_{b3}	17	28	99					

We then need two more passes of three-way merging to complete the sort.

T_{a1}	11	12	17	28	35	81	94	96	99
T_{a2}	15	41	58	75					
T_{a3}									
T_{b1}									
T_{b2}									
T_{b3}									

T_{a1}													
T_{a2}													
T_{a3}													
T_{b1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{b2}													
T_{b3}													

After the initial run construction phase, the number of passes required using k-way merging is $\lceil \log_k(N/M) \rceil$, because the runs get k times as large in each pass. For the example above, the formula is verified, since $\lceil \log_3(13/3) \rceil = 2$. If we have 10 tapes, then $k = 5$, and our large example from the previous section would require $\lceil \log_5 320 \rceil = 4$ passes.

7.11.5. Polyphase Merge

The k-way merging strategy developed in the last section requires the use of $2k$ tapes. This could be prohibitive for some applications. It is possible to get by with only $k + 1$ tapes. As an example, we will show how to perform two-way merging using only three tapes.

Suppose we have three tapes, T_1, T_2, and T_3, and an input file on T_1 that will produce 34 runs. One option is to put 17 runs on each of T_2 and T_3. We could then merge this result onto T_1, obtaining one tape with 17 runs. The problem is that since all the runs are on one tape, we must now put some of these runs on T_2 to perform another merge. The logical way to do this is to copy the first eight runs from T_1 onto T_2 and then perform the merge. This has the effect of adding an extra half pass for every pass we do.

An alternative method is to split the original 34 runs unevenly. Suppose we put 21 runs on T_2 and 13 runs on T_3. We would then merge 13 runs onto T_1 before T_3 was empty. At this point, we could rewind T_1 and T_3, and merge T_1, with 13 runs, and T_2, which has 8 runs, onto T_3. We could then merge 8 runs until T_2 was empty, which would leave 5 runs left on T_1 and 8 runs on T_3. We could then merge T_1 and T_3, and so on. The following table shows the number of runs on each tape after each pass.

	Run Const.	After $T_3 + T_2$	After $T_1 + T_2$	After $T_1 + T_3$	After $T_2 + T_3$	After $T_1 + T_2$	After $T_1 + T_3$	After $T_2 + T_3$
T_1	0	13	5	0	3	1	0	1
T_2	21	8	0	5	2	0	1	0
T_3	13	0	8	3	0	2	1	0

The original distribution of runs makes a great deal of difference. For instance, if 22 runs are placed on T_2, with 12 on T_3, then after the first merge, we obtain 12 runs on T_1 and 10 runs on T_2. After another merge, there are 10 runs on T_1 and 2 runs on T_3. At this point the going gets slow, because we can only merge two sets of runs before T_3 is exhausted. Then T_1 has 8 runs and T_2 has 2 runs. Again, we can only merge two sets of runs, obtaining T_1 with 6 runs and T_3 with 2 runs. After three more passes, T_2 has two runs and the other tapes are empty. We must copy one run to another tape, and then we can finish the merge.

It turns out that the first distribution we gave is optimal. If the number of runs is a Fibonacci number F_N, then the best way to distribute them is to split them into two Fibonacci numbers F_{N-1} and F_{N-2}. Otherwise, it is necessary to pad the tape with dummy runs in order to get the number of runs up to a Fibonacci number. We leave the details of how to place the initial set of runs on the tapes as an exercise.

We can extend this to a k-way merge, in which case we need kth order Fibonacci numbers for the distribution, where the kth order Fibonacci number is defined as $F^{(k)}(N) = F^{(k)}(N-1) + F^{(k)}(N-2) + \cdots + F^{(k)}(N-k)$, with the appropriate initial conditions $F^{(k)}(N) = 0, 0 \le N \le k-2, F^{(k)}(k-1) = 1$.

7.11.6. Replacement Selection

The last item we will consider is construction of the runs. The strategy we have used so far is the simplest possible: We read as many records as possible and sort them, writing the result to some tape. This seems like the best approach possible, until one realizes that as soon as the first record is written to an output tape, the memory it used becomes available for another record. If the next record on the input tape is larger than the record we have just output, then it can be included in the run.

Using this observation, we can give an algorithm for producing runs. This technique is commonly referred to as *replacement selection*. Initially, M records are read into memory and placed in a priority queue. We perform a *DeleteMin*, writing the smallest record to the output tape. We read the next record from the input tape. If it is larger than the record we have just written, we can add it to the priority queue. Otherwise, it cannot go into the current run. Since the priority queue is smaller by one element, we can store this new element in the dead space of the priority queue until the run is completed and use the element for the next run. Storing an element in the dead space is similar to what is done in heapsort. We continue doing this until the size of the priority queue is zero, at which point the run is over. We start a new run by building a new priority queue, using all the elements in the dead space. Figure 7.18 shows the run construction for the small example we have been using, with $M = 3$. Dead elements are indicated by an asterisk.

In this example, replacement selection produces only three runs, compared with the five runs obtained by sorting. Because of this, a three-way merge finishes in one pass instead of two. If the input is randomly distributed, replacement selection can be shown to produce runs of average length $2M$. For our large example, we would

Figure 7.18 Example of run construction

	3 Elements In Heap Array H[0]	H[1]	H[2]	Output	Next Element Read
Run 1	11	94	81	11	96
	81	94	96	81	12*
	94	96	12*	94	35*
	96	35*	12*	96	17*
	17*	35*	12*	End of Run.	Rebuild Heap
Run 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	75*
	58	99	75*	58	end of tape
	99		75*	99	
			75*	End of Run.	Rebuild Heap
Run 3	75			75	

expect 160 runs instead of 320 runs, so a five-way merge would require four passes. In this case, we have not saved a pass, although we might if we get lucky and have 125 runs or less. Since external sorts take so long, every pass saved can make a significant difference in the running time.

As we have seen, it is possible for replacement selection to do no better than the standard algorithm. However, the input is frequently sorted or nearly sorted to start with, in which case replacement selection produces only a few very long runs. This kind of input is common for external sorts and makes replacement selection extremely valuable.

Summary

For most general internal sorting applications, either insertion sort, Shellsort, or quicksort will be the method of choice, and the decision of which to use will depend mostly on the size of the input. Figure 7.19 shows the running time obtained for each algorithm on various file sizes (on a relatively slow computer).

The data was chosen to be random permutations of N integers, and the times given include only the actual time to sort. The code given in Figure 7.2 was used for insertion sort. Shellsort used the code in Section 7.4 modified to run with Sedgewick's increments. Based on literally millions of sorts, ranging in size from 100 to 25 million, the expected running time of Shellsort with these increments is conjectured to be $O(N^{7/6})$. The heapsort routine is the same as in Section 7.5. Two versions of quicksort are given. The first uses a simple pivoting strategy and does not do a cutoff. Fortunately, the input files were random. The second uses median-of-three partitioning and a cutoff of ten. Further optimizations were possible. We could have coded the median-of-three routine in-line instead of using a function, and we could have written quicksort nonrecursively. There are some other optimizations to the code that are fairly tricky to implement, and of course we could have used an assembly language. We have made an honest attempt to code all routines efficiently, but of course the performance can vary somewhat from machine to machine.

Figure 7.19 Comparison of different sorting algorithms (all times are in seconds)

N	Insertion Sort $O(N^2)$	Shellsort $O(N^{7/6})$(?)	Heapsort $O(N \log N)$	Quicksort $O(N \log N)$	Quicksort (opt.) $O(N \log N)$
10	0.00044	0.00041	0.00057	0.00052	.00046
100	0.00675	0.00171	0.00420	0.00284	.00244
1000	0.59564	0.02927	0.05565	0.03153	.02587
10000	58.864	0.42998	0.71650	0.36765	.31532
100000	NA	5.7298	8.8591	4.2298	3.5882
1000000	NA	71.164	104.68	47.065	41.282

The highly optimized version of quicksort is as fast as Shellsort even for very small input sizes. The improved version of quicksort still has an $O(N^2)$ worst case (one exercise asks you to construct a small example), but the chances of this worst case appearing are so negligible as to not be a factor. If you need to sort large files, quicksort is the method of choice. But never, ever, take the easy way out and use the first element as pivot. It is just not safe to assume that the input will be random. If you do not want to worry about this, use Shellsort. Shellsort will give a small performance penalty but could also be acceptable, especially if simplicity is required. Its worst case is only $O(N^{4/3})$; the chance of that worst case occurring is likewise negligible.

Heapsort, although an $O(N \log N)$ algorithm with an apparently tight inner loop, is slower than Shellsort. A close examination of the algorithm reveals that in order to move data, heapsort does two comparisons. An improvement suggested by Floyd moves data with essentially only one comparison, but implementing this improvement makes the code somewhat longer. We leave it to the reader to decide whether the extra coding effort is worth the increased speed (Exercise 7.40).

Insertion sort is useful only for small or very nearly sorted inputs. We have not included mergesort, because its performance is not as good as quicksort for main memory sorts and it is not any simpler to code. We have seen, however, that merging is the central idea of external sorts.

Exercises

7.1 Sort the sequence 3, 1, 4, 1, 5, 9, 2, 6, 5 using insertion sort.

7.2 What is the running time of insertion sort if all keys are equal?

7.3 Suppose we exchange elements $A[i]$ and $A[i + k]$, which were originally out of order. Prove that at least 1 and at most $2k - 1$ inversions are removed.

7.4 Show the result of running Shellsort on the input 9, 8, 7, 6, 5, 4, 3, 2, 1 using the increments {1, 3, 7}.

7.5 a. What is the running time of Shellsort using the two-increment sequence {1, 2}?

b. Show that for any N, there exists a three-increment sequence such that Shellsort runs in $O(N^{5/3})$ time.

c. Show that for any N, there exists a six-increment sequence such that Shellsort runs in $O(N^{3/2})$ time.

7.6 *a. Prove that the running time of Shellsort is $\Omega(N^2)$ using increments of the form $1, c, c^2, \ldots, c^i$ for any integer c.

**b. Prove that for these increments, the average running time is $\Theta(N^{3/2})$.

*7.7 Prove that if a k-sorted file is then h-sorted, it remains k-sorted.

**7.8 Prove that the running time of Shellsort, using the increment sequence suggested by Hibbard, is $\Omega(N^{3/2})$ in the worst case. *Hint:* You can prove the bound by considering the special case of what Shellsort does when all elements are either

0 or 1. Set *InputData*[*i*] = 1 if *i* is expressible as a linear combination of h_t, $h_{t-1}, \ldots, h_{\lfloor t/2 \rfloor + 1}$ and 0 otherwise.

7.9 Determine the running time of Shellsort for

a. sorted input

*b. reverse-ordered input

7.10 Do either of the following modifications to the Shellsort routine coded in Fig. 7.4 affect the worst case running time?

a. Before line 2, subtract one from *Increment* if it is even.

b. Before line 2, add one to *Increment* if it is even.

7.11 Show how heapsort processes the input 142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102.

7.12 a. What is the running time of heapsort for presorted input?

*b. Prove that the worst case bound for heapsort is achievable.

7.13 Sort 3, 1, 4, 1, 5, 9, 2, 6 using mergesort.

7.14 How would you implement mergesort without using recursion?

7.15 Determine the running time of mergesort for

a. sorted input

b. reverse-ordered input

c. random input

7.16 In the analysis of mergesort, constants have been disregarded. Prove that the number of comparisons used in the worst case by mergesort is $N\lceil \log N \rceil - 2^{\lceil \log N \rceil} + 1$.

7.17 Sort 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 using quicksort with median-of-three partitioning and a cutoff of 3.

7.18 Using the quicksort implementation in this chapter, determine the running time of quicksort for

a. sorted input

b. reverse-ordered input

c. random input

7.19 Repeat Exercise 7.18 when the pivot is chosen as

a. the first element

b. the largest of the first two nondistinct keys

c. a random element

*d. the average of all keys in the set

7.20 a. For the quicksort implementation in this chapter, what is the running time when all keys are equal?

b. Suppose we change the partitioning strategy so that neither *i* nor *j* stops when an element with the same key as the pivot is found. What fixes need to be made in the code to guarantee that quicksort works, and what is the running time, when all keys are equal?

c. Suppose we change the partitioning strategy so that *i* stops at an element with the same key as the pivot, but *j* does not stop in a similar case. What fixes need to be made in the code to guarantee that quicksort works, and when all keys are equal, what is the running time of quicksort?

7.21 Suppose we choose the middle key as pivot. Does this make it unlikely that quicksort will require quadratic time?

7.22 Construct a permutation of 20 elements that is as bad as possible for quicksort using median-of-three partitioning and a cutoff of 3.

7.23 Write a program to implement the selection algorithm.

7.24 Solve the following recurrence: $T(N) = (1/N)[\sum_{i=0}^{N-1} T(i)] + cN, T(0) = 0$

7.25 A sorting algorithm is *stable* if elements with equal keys are left in the same order as they occur in the input. Which of the sorting algorithms in this chapter are stable and which are not? Why?

7.26 Suppose you are given a sorted list of *N* elements followed by $f(N)$ randomly ordered elements. How would you sort the entire list if

a. $f(N) = O(1)$?

b. $f(N) = O(\log N)$?

c. $f(N) = O(\sqrt{N})$?

*d. How large can $f(N)$ be for the entire list still to be sortable in $O(N)$ time

7.27 Prove that any algorithm that finds an element *X* in a sorted list of *N* elements requires $\Omega(\log N)$ comparisons.

7.28 Using Stirling's formula, $N! \approx (N/e)^N \sqrt{2\pi N}$, give a precise estimate for $\log(N!)$.

7.29 *a. In how many ways can two sorted arrays of *N* elements be merged?

*b. Give a nontrivial lower bound on the number of comparisons required to merge two sorted lists of *N* elements.

7.30 Prove that sorting *N* elements with integer keys in the range $1 \le Key \le M$ takes $O(M + N)$ time using bucket sort.

7.31 Suppose you have an array of *N* elements containing only two distinct keys, *true* and *false*. Give an $O(N)$ algorithm to rearrange the list so that all *false* elements precede the *true* elements. You may use only constant extra space.

7.32 Suppose you have an array of *N* elements, containing three distinct keys, *true*, *false*, and *maybe*. Give an $O(N)$ algorithm to rearrange the list so that all *false* elements precede *maybe* elements, which in turn precede *true* elements. You may use only constant extra space.

7.33 a. Prove that any comparison-based algorithm to sort 4 elements requires 5 comparisons.

b. Give an algorithm to sort 4 elements in 5 comparisons.

7.34 a. Prove that 7 comparisons are required to sort 5 elements using any comparison-based algorithm.

*b. Give an algorithm to sort 5 elements with 7 comparisons.

7.35 Write an efficient version of Shellsort and compare performance when the following increment sequences are used:

a. Shell's original sequence

b. Hibbard's increments

c. Knuth's increments: $h_i = \frac{1}{2}(3^i + 1)$

d. Gonnet's increments: $h_t = \lfloor \frac{N}{2.2} \rfloor$ and $h_k = \lfloor \frac{h_{k+1}}{2.2} \rfloor$ (with $h_1 = 1$ if $h_2 = 2$)

e. Sedgewick's increments.

7.36 Implement an optimized version of quicksort and experiment with combinations of the following:

a. Pivot: first element, middle element, random element, median of three, median of five.

b. Cutoff values from 0 to 20.

7.37 Write a routine that reads in two alphabetized files and merges them together, forming a third, alphabetized, file.

7.38 Suppose we implement the median of three routine as follows: Find the median of *A*[*Left*], *A*[*Center*], *A*[*Right*], and swap it with *A*[*Right*]. Proceed with the normal partitioning step starting *i* at *Left* and *j* at *Right* − 1 (instead of *Left* + 1 and *Right* − 2).

a. Suppose the input is 2, 3, 4, ..., $N - 1$, N, 1. For this input, what is the running time of this version of quicksort?

b. Suppose the input is in reverse order. For this input, what is the running time of this version of quicksort?

7.39 Prove that any comparison-based sorting algorithm requires $\Omega(N \log N)$ comparisons on average.

7.40 Consider the following strategy for *PercolateDown*. We have a hole at node *X*. The normal routine is to compare *X*'s children and then move the child up to *X* if it is larger (in the case of a (*max*)heap) than the element we are trying to place, thereby pushing the hole down; we stop when it is safe to place the new element in the hole. The alternate strategy is to move elements up and the hole down as far as possible, without testing whether the new cell can be inserted. This would place the new cell in a leaf and probably violate the heap order; to fix the heap order, percolate the new cell up in the normal manner. Write a routine to include this idea, and compare the running time with a standard implementation of heapsort.

7.41 Propose an algorithm to sort a large file using only two tapes.

7.42 a. Show that a lower bound of $N!/2^{2N}$ on the number of heaps is implied by the fact that build-heap uses at most $2N$ comparisons.

b. Use Stirling's formula to expand this bound.

7.43 ANSI C requires the routine *qsort* to be present in C libraries. *qsort* is typically implemented by quicksort (but this is not required). Experiment with various inputs to see if *qsort* can be driven to quadratic behavior. Try random 0s and 1s.

References

Knuth's book [13] is a comprehensive, though somewhat dated, reference for sorting. Gonnet and Baeza-Yates [5] has some more recent results, as well as a huge bibliography.

The original paper detailing Shellsort is [22]. The paper by Hibbard [6] suggested the use of the increments $2^k - 1$ and tightened the code by avoiding swaps. Theorem 7.4 is from [13]. Pratt's lower bound, which uses a more complex method than that suggested in the text, can be found in [15]. Improved increment sequences and upper bounds appear in [10], [21], and [24]; matching lower bounds have been shown in [25]. A recent result shows that no increment sequence gives an $O(N \log N)$ worst-case running time [14]. The average-case running time for Shellsort is still unresolved. Yao [27] has performed an extremely complex analysis for the three-increment case. The result has yet to be extended to more increments. Experiments with various increment sequences appear in [23].

Heapsort was invented by Williams [26]; Floyd [2] provided the linear-time algorithm for heap construction. Theorem 7.5 is from [16].

An exact average-case analysis of mergesort has been claimed in [4]; the paper detailing the results is forthcoming. An algorithm to perform merging in linear time without extra space is described in [9].

Quicksort is from Hoare [7]. This paper analyzes the basic algorithm, describes most of the improvements, and includes the selection algorithm. A detailed analysis and empirical study was the subject of Sedgewick's dissertation [20]. Many of the important results appear in the three papers [17], [18], and [19]. [1] provides a detailed C implementation with some additional improvements, and points out that most implementations of the *qsort* library routine are easily driven to quadratic behavior.

Decision trees and sorting optimality are discussed in Ford and Johnson [3]. This paper also provides an algorithm that almost meets the lower bound in terms of number of comparisons (but not other operations). This algorithm was eventually shown to be slightly suboptimal by Manacher [12].

External sorting is covered in detail in [11]. Stable sorting, described in Exercise 7.25, has been addressed by Horvath [8].

1. J. L. Bentley and M. D. McElroy, "Engineering a Sort Function," *Software—Practice and Experience*, 23 (1993), 1249–1265.
2. R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, 7 (1964), 701.
3. L. R. Ford and S. M. Johnson, "A Tournament Problem," *American Mathematics Monthly*, 66 (1959), 387–389.
4. M. Golin and R. Sedgewick, "Exact Analysis of Mergesort," *Fourth SIAM Conference on Discrete Mathematics*, 1988.
5. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2nd ed., Addison-Wesley, Reading, Mass., 1991.
6. T. H. Hibbard, "An Empirical Study of Minimal Storage Sorting," *Communications of the ACM*, 6 (1963), 206–213.
7. C. A. R. Hoare, "Quicksort," *Computer Journal*, 5 (1962), 10–15.

8. E. C. Horvath, "Stable Sorting in Asymptotically Optimal Time and Extra Space," *Journal of the ACM*, 25 (1978), 177–199.
9. B. Huang and M. Langston, "Practical In-place Merging," *Communications of the ACM*, 31 (1988), 348–352.
10. J. Incerpi and R. Sedgewick, "Improved Upper Bounds on Shellsort," *Journal of Computer and System Sciences*, 31 (1985), 210–224.
11. D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.
12. G. K. Manacher, "The Ford-Johnson Sorting Algorithm Is Not Optimal," *Journal of the ACM*, 26 (1979), 441–456.
13. A. A. Papernov and G. V. Stasevich, "A Method of Information Sorting in Computer Memories," *Problems of Information Transmission*, 1 (1965), 63–75.
14. C. G. Plaxton, B. Pooren, and T. Suel, "Improved Lower Bounds for Shellsort," Proceedings of the Thirty-third Annual Symposium on the Foundations of Computer Science (1992), 226–235.
15. V. R. Pratt, *Shellsort and Sorting Networks*, Garland Publishing, New York, 1979. (Originally presented as the author's Ph.D. thesis, Stanford University, 1971.)
16. R. Schaffer and R. Sedgewick, "The Analysis of Heapsort," *Journal of Algorithms*, 14 (1993), 76–100.
17. R. Sedgewick, "Quicksort with Equal Keys," *SIAM Journal on Computing*, 6 (1977), 240–267.
18. R. Sedgewick, "The Analysis of Quicksort Programs," *Acta Informatica*, 7 (1977), 327–355.
19. R. Sedgewick, "Implementing Quicksort Programs," *Communications of the ACM*, 21 (1978), 847–857.
20. R. Sedgewick, *Quicksort*, Garland Publishing, New York, 1978. (Originally presented as the author's Ph.D. thesis, Stanford University, 1975.)
21. R. Sedgewick, "A New Upper Bound for Shellsort," *Journal of Algorithms*, 7 (1986), 159–173.
22. D. L. Shell, "A High-Speed Sorting Procedure," *Communications of the ACM*, 2 (1959), 30–32.
23. M. A. Weiss, "Empirical Results on the Running Time of Shellsort," *Computer Journal*, 34 (1991), 88–91.
24. M. A. Weiss and R. Sedgewick, "More On Shellsort Increment Sequences," *Information Processing Letters*, 34 (1990), 267–270.
25. M. A. Weiss and R. Sedgewick, "Tight Lower Bounds For Shellsort," *Journal of Algorithms*, 11 (1990), 242–251.
26. J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, 7 (1964), 347–348.
27. A. C. Yao, "An Analysis of $(h, k, 1)$ Shellsort," *Journal of Algorithms*, 1 (1980), 14–50.

Table of contents

Fundamental Structures of Computer Science
(Course Materials)

Part 2
Data Structures and Algorithm Analysis

Тираж 100 экз. Объем 268 стр.
Формат 60x84/16. Бумага офсетная №1.
Плотность 80 г/м^2. Печать RISO.

ТОО «Эверо». Полиграфические услуги.
Тел.: 39-32-69, тел./факс: 32-38-43
e-mail: evero@nursat.kz