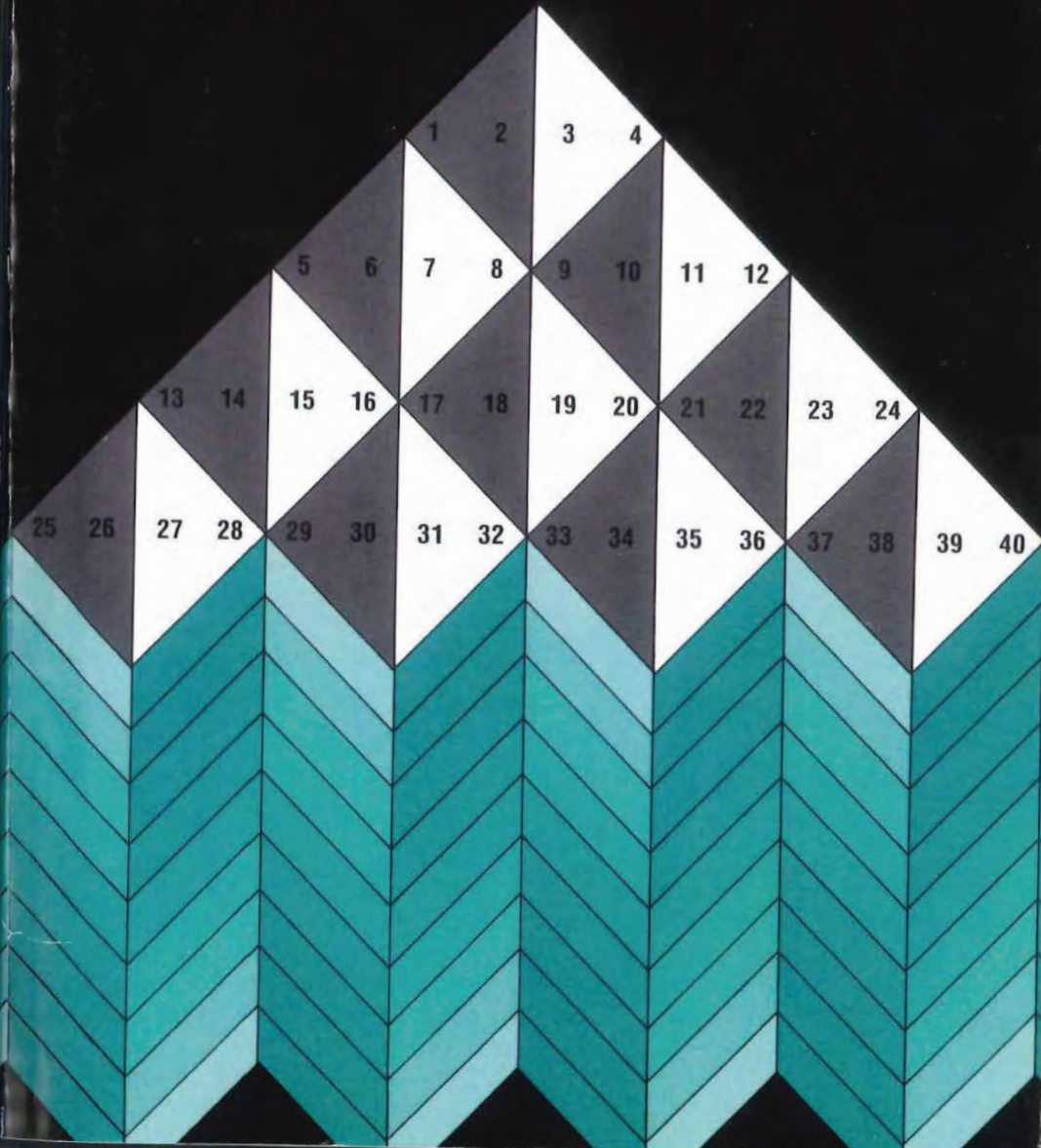# DATA STRUCTURES AND ALGORITHMS

ALFRED V. AHO
JOHN E. HOPCROFT
JEFFREY D. ULLMAN

Data Structures

and Algorithms

ALFRED V. AHO

Bell Laboratories
Murray Hill, New Jersey

JOHN E. HOPCROFT

Cornell University
Ithaca, New York

JEFFREY D. ULLMAN

Stanford University
Stanford, California

Contents

CHAPTER 3

Trees

A tree imposes a hierarchical structure on a collection of items. Familiar examples of trees are genealogies and organization charts. Trees are used to help analyze electrical circuits and to represent the structure of mathematical formulas. Trees also arise naturally in many different areas of computer science. For example, trees are used to organize information in database systems and to represent the syntactic structure of source programs in compilers. Chapter 5 describes applications of trees in the representation of data. Throughout this book, we shall use many different variants of trees. In this chapter we introduce the basic definitions and present some of the more common tree operations. We then describe some of the more frequently used data structures for trees that can be used to support these operations efficiently.

3.1 Basic Terminology

A tree is a collection of elements called *nodes*, one of which is distinguished as a *root*, along with a relation ("parenthood") that places a hierarchical structure on the nodes. A node, like an element of a list, can be of whatever type we wish. We often depict a node as a letter, a string, or a number with a circle around it. Formally, a *tree* can be defined recursively in the following manner.

1. A single node by itself is a tree. This node is also the root of the tree.
2. Suppose n is a node and $T_1, T_2, \ldots, T_k$ are trees with roots $n_1, n_2, \ldots, n_k$, respectively. We can construct a new tree by making n be the parent of nodes $n_1, n_2, \ldots, n_k$. In this tree n is the root and $T_1, T_2, \ldots, T_k$ are the *subtrees* of the root. Nodes $n_1, n_2, \ldots, n_k$ are called the *children* of node n.

Sometimes, it is convenient to include among trees the *null tree*, a "tree" with no nodes, which we shall represent by Λ.

Example 3.1. Consider the table of contents of a book, as suggested by Fig. 3.1(a). This table of contents is a tree. We can redraw it in the manner shown in Fig. 3.1(b). The parent-child relationship is depicted by a line. Trees are normally drawn top-down as in Fig. 3.1(b), with the parent above the child.

The root, the node called "Book," has three subtrees with roots corresponding to the chapters C1, C2, and C3. This relationship is represented by the lines downward from Book to C1, C2, and C3. Book is the parent of C1, C2, and C3, and these three nodes are the children of Book.

7

```
Book                                      Book
  C1                                       |
    s1.1                          ┌────────┼────────┐
    s1.2                         C1        C2        C3
  C2                            /  \      / | \
    s2.1                     s1.1  s1.2 s2.1 s2.2 s2.3
      s2.1.1                              / \
      s2.1.2                           s2.1.1 s2.1.2
    s2.2
    s2.3
  C3

      (a)                                  (b)
```
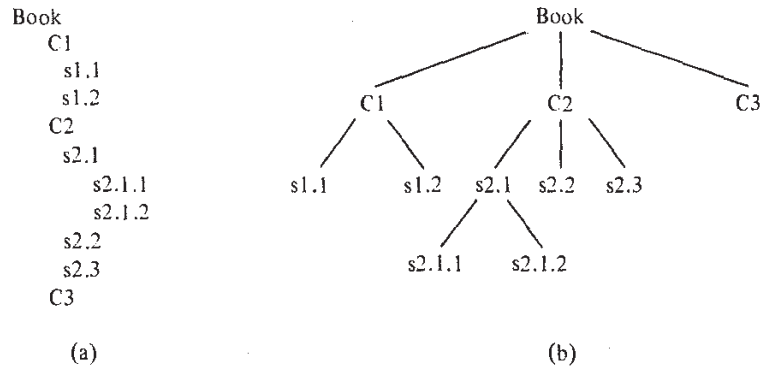
Fig. 3.1. A table of contents and its tree representation.

The third subtree, with root C3, is a tree of a single node, while the other two subtrees have a nontrivial structure. For example, the subtree with root C2 has three subtrees, corresponding to the sections s2.1, s2.2, and s2.3; the last two are one-node trees, while the first has two subtrees corresponding to the subsections s2.1.1 and s2.1.2. □

Example 3.1 is typical of one kind of data that is best represented as a tree. In this example, the parenthood relationship stands for containment; a parent node is comprised of its children, as Book is comprised of C1, C2, and C3. Throughout this book we shall encounter a variety of other relationships that can be represented by parenthood in trees.

If $n_1, n_2, \ldots, n_k$ is a sequence of nodes in a tree such that n_i is the parent of n_{i+1} for $1 \leq i < k$, then this sequence is called a *path* from node n_1 to node n_k. The *length* of a path is one less than the number of nodes in the path. Thus there is a path of length zero from every node to itself. For example, in Fig. 3.1 there is a path of length two, namely (C2, s2.1, s2.1.2) from C2 to s2.1.2.

If there is a path from node a to node b, then a is an *ancestor* of b, and b is a *descendant* of a. For example, in Fig. 3.1, the ancestors of s2.1, are itself, C2, and Book, while its descendants are itself, s2.1.1, and s2.1.2. Notice that any node is both an ancestor and a descendant of itself.

An ancestor or descendant of a node, other than the node itself, is called a *proper* ancestor or *proper* descendant, respectively. In a tree, the root is the only node with no proper ancestors. A node with no proper descendants is called a *leaf*. A subtree of a tree is a node, together with all its descendants.

The *height* of a node in a tree is the length of a longest path from the node to a leaf. In Fig. 3.1 node C1 has height 1, node C2 height 2, and node C3 height 0. The *height of a tree* is the height of the root. The *depth* of a node is the length of the unique path from the root to that node.

The Order of Nodes

The children of a node are usually ordered from left-to-right. Thus the two trees of Fig. 3.2 are different because the two children of node a appear in a different order in the two trees. If we wish explicitly to ignore the order of children, we shall refer to a tree as an *unordered* tree.
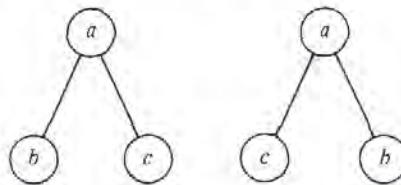
Fig. 3.2. Two distinct (ordered) trees.

The "left-to-right" ordering of *siblings* (children of the same node) can be extended to compare any two nodes that are not related by the ancestor-descendant relationship. The relevant rule is that if a and b are siblings, and a is to the left of b, then all the descendants of a are to the left of all the descendants of b.

Example 3.2. Consider the tree in Fig. 3.3. Node 8 is to the right of node 2, to the left of nodes 9, 6, 10, 4, and 7, and neither left nor right of its ancestors 1, 3, and 5.
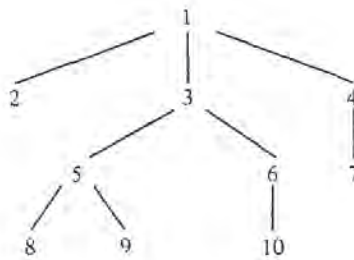
Fig. 3.3. A tree.

A simple rule, given a node n, for finding those nodes to its left and those to its right, is to draw the path from the root to n. All nodes branching off to the left of this path, and all descendants of such nodes, are to the left of n. All nodes and descendants of nodes branching off to the right are to the right of n. □

Preorder, Postorder, and Inorder

There are several useful ways in which we can systematically order all nodes of a tree. The three most important orderings are called preorder, inorder and postorder; these orderings are defined recursively as follows.

● If a tree T is null, then the empty list is the preorder, inorder and post-order listing of T.

● If T consists a single node, then that node by itself is the preorder, inorder, and postorder listing of T.

Otherwise, let T be a tree with root n and subtrees $T_1, T_2, \ldots, T_k$, as suggested in Fig. 3.4.
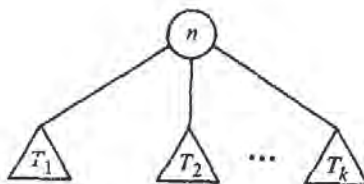
Fig. 3.4. Tree T.

1. The *preorder listing* (or *preorder traversal*) of the nodes of T is the root n of T followed by the nodes of T_1 in preorder, then the nodes of T_2 in preorder, and so on, up to the nodes of T_k in preorder.

2. The *inorder listing* of the nodes of T is the nodes of T_1 in inorder, followed by node n, followed by the nodes of $T_2, \ldots, T_k$, each group of nodes in inorder.

3. The *postorder listing* of the nodes of T is the nodes of T_1 in postorder, then the nodes of T_2 in postorder, and so on, up to T_k, all followed by node n.

Figure 3.5(a) shows a sketch of a procedure to list the nodes of a tree in preorder. To make it a postorder procedure, we simply reverse the order of steps (1) and (2). Figure 3.5(b) is a sketch of an inorder procedure. In each case, we produce the desired ordering of the tree by calling the appropriate procedure on the root of the tree.

Example 3.3. Let us list the tree of Fig. 3.3 in preorder. We first list 1 and then call PREORDER on the first subtree of 1, the subtree with root 2. This subtree is a single node, so we simply list it. Then we proceed to the second subtree of 1, the tree rooted at 3. We list 3, and then call PREORDER on the first subtree of 3. That call results in listing 5, 8, and 9, in that order.

```
        procedure PREORDER ( n: node );
            begin
(1)             list n;
(2)             for each child c of n, if any, in order from the left do
                    PREORDER(c)
            end; { PREORDER }
```

(a) PREORDER procedure.

```
        procedure INORDER ( n: node );
            begin
                if n is a leaf then
                    list n
                else begin
                    INORDER(leftmost child of n);
                    list n;
                    for each child c of n, except for the leftmost,
                        in order from the left do
                            INORDER(c)
                end
            end; { INORDER }
```

(b) INORDER procedure.

Fig. 3.5. Recursive ordering procedures.

Continuing in this manner, we obtain the complete preorder traversal of Fig. 3.3: 1, 2, 3, 5, 8, 9, 6, 10, 4, 7.

Similarly, by simulating Fig. 3.5(a) with the steps reversed, we can discover that the postorder of Fig. 3.3 is 2, 8, 9, 5, 10, 6, 3, 7, 4, 1. By simulating Fig. 3.5(b), we find that the inorder listing of Fig. 3.3 is 2, 1, 8, 5, 9, 3, 10, 6, 7, 4. □

A useful trick for producing the three node orderings is the following. Imagine we walk around the outside of the tree, starting at the root, moving counterclockwise, and staying as close to the tree as possible; the path we have in mind for Fig. 3.3 is shown in Fig. 3.6.

For preorder, we list a node the first time we pass it. For postorder, we list a node the last time we pass it, as we move up to its parent. For inorder, we list a leaf the first time we pass it, but list an interior node the second time we pass it. For example, node 1 in Fig. 3.6 is passed the first time at the beginning, and the second time while passing through the "bay" between nodes 2 and 3. Note that the order of the leaves in the three orderings is
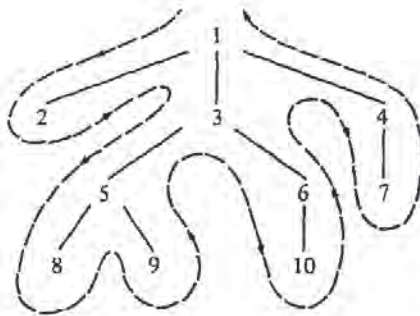
Fig. 3.6. Traversal of a tree.

always the same left-to-right ordering of the leaves. It is only the ordering of the interior nodes and their relationship to the leaves that vary among the three. □

Labeled Trees and Expression Trees

Often it is useful to associate a *label*, or value, with each node of a tree, in the same spirit with which we associated a value with a list element in the previous chapter. That is, the label of a node is not the name of the node, but a value that is "stored" at the node. In some applications we shall even change the label of a node, while the name of a node remains the same. A useful analogy is tree:list = label:element = node:position.

Example 3.4. Figure 3.7 shows a labeled tree representing the arithmetic expression $(a+b) * (a+c)$, where $n_1, \ldots, n_7$ are the names of the nodes, and the labels, by convention, are shown next to the nodes. The rules whereby a labeled tree represents an expression are as follows:

1. Every leaf is labeled by an operand and consists of that operand alone. For example, node n_4 represents the expression a.

2. Every interior node n is labeled by an operator. Suppose n is labeled by a binary operator θ, such as $+$ or $*$, and that the left child represents expression E_1 and the right child E_2. Then n represents expression $(E_1) \theta (E_2)$. We may remove the parentheses if they are not necessary.

For example, node n_2 has operator $+$, and its left and right children represent the expressions a and b, respectively. Therefore, n_2 represents $(a)+(b)$, or just $a+b$. Node n_1 represents $(a+b)*(a+c)$, since $*$ is the label

at n_1, and $a+b$ and $a+c$ are the expressions represented by n_2 and n_3, respectively. □
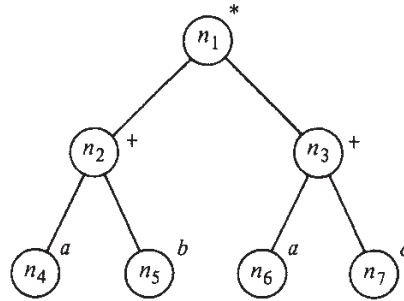
Fig. 3.7. Expression tree with labels.

Often, when we produce the preorder, inorder, or postorder listing of a tree, we prefer to list not the node names, but rather the labels. In the case of an expression tree, the preorder listing of the labels gives us what is known as the *prefix* form of an expression, where the operator precedes its left operand and its right operand. To be precise, the prefix expression for a single operand a is a itself. The prefix expression for $(E_1)\ \theta\ (E_2)$, with θ a binary operator, is $\theta P_1 P_2$, where P_1 and P_2 are the prefix expressions for E_1 and E_2. Note that no parentheses are necessary in the prefix expression, since we can scan the prefix expression $\theta P_1 P_2$ and uniquely identify P_1 as the shortest (and only) prefix of $P_1 P_2$ that is a legal prefix expression.

For example, the preorder listing of the labels of Fig. 3.7 is $*+ab+ac$. The prefix expression for n_2, which is $+ab$, is the shortest legal prefix of $+ab+ac$.

Similarly, a postorder listing of the labels of an expression tree gives us what is known as the *postfix* (or *Polish*) representation of an expression. The expression $(E_1)\ \theta\ (E_2)$ is represented by the postfix expression $P_1 P_2 \theta$, where P_1 and P_2 are the postfix representations of E_1 and E_2, respectively. Again, no parentheses are necessary in the postfix representation, as we can deduce what P_2 is by looking for the shortest suffix of $P_1 P_2$ that is a legal postfix expression. For example, the postfix expression for Fig. 3.7 is $ab+ac+*$. If we write this expression as $P_1 P_2 *$, then P_2 is $ac+$, the shortest suffix of $ab+ac+$ that is a legal postfix expression.

The inorder traversal of an expression tree gives the infix expression itself, but with no parentheses. For example, the inorder listing of the labels of Fig. 3.7 is $a+b * a+c$. The reader is invited to provide an algorithm for traversing an expression tree and producing an infix expression with all needed pairs of parentheses.

Computing Ancestral Information

The preorder and postorder traversals of a tree are useful in obtaining ancestral information. Suppose $postorder(n)$ is the position of node n in a postorder listing of the nodes of a tree. Suppose $desc(n)$ is the number of proper descendants of node n. For example, in the tree of Fig. 3.7 the postorder numbers of nodes n_2, n_4, and n_5 are 3, 1, and 2, respectively.

The postorder numbers assigned to the nodes have the useful property that the nodes in the subtree with root n are numbered consecutively from $postorder(n) - desc(n)$ to $postorder(n)$. To test if a vertex x is a descendant of vertex y, all we need do is determine whether

$$postorder(y) - desc(y) \leq postorder(x) \leq postorder(y).$$

A similar property holds for preorder.

3.2 The ADT TREE

In Chapter 2, lists, stacks, queues, and mappings were treated as abstract data types (ADT's). In this chapter trees will be treated both as ADT's and as data structures. One of our most important uses of trees occurs in the design of implementations for the various ADT's we study. For example, in Section 5.1, we shall see how a "binary search tree" can be used to implement abstract data types based on the mathematical model of a set, together with operations such as INSERT, DELETE, and MEMBER (to test whether an element is in a set). The next two chapters present a number of other tree implementations of various ADT's.

In this section, we shall present several useful operations on trees and show how tree algorithms can be designed in terms of these operations. As with lists, there are a great variety of operations that can be performed on trees. Here, we shall consider the following operations:

1. PARENT(n, T). This function returns the parent of node n in tree T. If n is the root, which has no parent, Λ is returned. In this context, Λ is a "null node," which is used as a signal that we have navigated off the tree.

2. LEFTMOST_CHILD(n, T) returns the leftmost child of node n in tree T, and it returns Λ if n is a leaf, which therefore has no children.

3. RIGHT_SIBLING(n, T) returns the right sibling of node n in tree T, defined to be that node m with the same parent p as n such that m lies immediately to the right of n in the ordering of the children of p. For example, for the tree in Fig. 3.7, LEFTMOST_CHILD(n_2) = n_4; RIGHT_SIBLING(n_4) = n_5, and RIGHT_SIBLING (n_5) = Λ.

14

4. LABEL(n, T) returns the label of node n in tree T. We do not, however, require labels to be defined for every tree.

5. CREATEi(v, T_1, T_2, . . . , T_i) is one of an infinite family of functions, one for each value of $i = 0, 1, 2, ...$ CREATEi makes a new node r with label v and gives it i children, which are the roots of trees T_1, T_2, . . . , T_i, in order from the left. The tree with root r is returned. Note that if $i = 0$, then r is both a leaf and the root.

6. ROOT(T) returns the node that is the root of tree T, or Λ if T is the null tree.

7. MAKENULL(T) makes T be the null tree.

Example 3.5. Let us write both recursive and nonrecursive procedures to take a tree and list the labels of its nodes in preorder. We assume that there are data types node and TREE already defined for us, and that the data type TREE is for trees with labels of the type labeltype. Figure 3.8 shows a recursive procedure that, given node n, lists the labels of the subtree rooted at n in preorder. We call PREORDER(ROOT(T)) to get a preorder listing of tree T.

```
procedure PREORDER ( n: node );
    { list the labels of the descendants of n in preorder }
    var
        c: node;
    begin
        print(LABEL(n, T));
        c := LEFTMOST_CHILD(n, T);
        while c <> Λ do begin
            PREORDER(c);
            c := RIGHT_SIBLING(c, T)
        end
    end; { PREORDER }
```

Fig. 3.8. A recursive preorder listing procedure.

We shall also develop a nonrecursive procedure to print a tree in preorder. To find our way around the tree, we shall use a stack S, whose type STACK is really "stack of nodes." The basic idea underlying our algorithm is that when we are at a node n, the stack will hold the path from the root to n, with the root at the bottom of the stack and node n at the top.†

---

† Recall our discussion of recursion in Section 2.6 in which we illustrated how the implementation of a recursive procedure involves a stack of activation records. If we examine Fig. 3.8, we can observe that when PREORDER(n) is called, the active procedure calls, and therefore the stack of activation records, correspond to the calls of PREORDER for all the ancestors of n. Thus our nonrecursive preorder procedure, like the example in Section 2.6, models closely the way the recursive procedure is implemented.

One way to perform a nonrecursive preorder traversal of a tree is given by the program NPREORDER shown in Fig. 3.9. This program has two modes of operation. In the first mode it descends down the leftmost unexplored path in the tree, printing and stacking the nodes along the path, until it reaches a leaf.

The program then enters the second mode of operation in which it retreats back up the stacked path, popping the nodes of the path off the stack, until it encounters a node on the path with a right sibling. The program then reverts back to the first mode of operation, starting the descent from that unexplored right sibling.

The program begins in mode one at the root and terminates when the stack becomes empty. The complete program is shown in Fig. 3.9.

3.3 Implementations of Trees

In this section we shall present several basic implementations for trees and discuss their capabilities for supporting the various tree operations introduced in Section 3.2.

An Array Representation of Trees

Let T be a tree in which the nodes are named $1, 2, \ldots, n$. Perhaps the simplest representation of T that supports the PARENT operation is a linear array A in which entry $A[i]$ is a pointer or a cursor to the parent of node i. The root of T can be distinguished by giving it a null pointer or a pointer to itself as parent. In Pascal, pointers to array elements are not feasible, so we shall have to use a cursor scheme where $A[i] = j$ if node j is the parent of node i, and $A[i] = 0$ if node i is the root.

This representation uses the property of trees that each node has a unique parent. With this representation the parent of a node can be found in constant time. A path going up the tree, that is, from node to parent to parent, and so on, can be traversed in time proportional to the number of nodes on the path. We can also support the LABEL operator by adding another array L, such that $L[i]$ is the label of node i, or by making the elements of array A be records consisting of an integer (cursor) and a label.

Example 3.6. The tree of Fig. 3.10(a) has the parent representation given by the array A shown in Fig. 3.10(b). □

The parent pointer representation does not facilitate operations that require child-of information. Given a node n, it is expensive to determine the children of n, or the height of n. In addition, the parent pointer representation does not specify the order of the children of a node. Thus, operations like LEFTMOST_CHILD and RIGHT_SIBLING are not well defined. We could impose an artificial order, for example, by numbering the children of each node after numbering the parent, and numbering the children in

```
procedure NPREORDER ( T: TREE );
    { nonrecursive preorder traversal of tree T }

    var
        m: node;  { a temporary }
        S: STACK;  { stack of nodes holding path from the root
            to the parent TOP(S) of the "current" node m }

    begin
        { initialize }
        MAKENULL(S);
        m := ROOT(T);

        while true do
            if m <> Λ then begin
                print(LABEL(m, T));
                PUSH(m, S);
                { explore leftmost child of m }
                m := LEFTMOST_CHILD(m, T)
            end
            else begin
                { exploration of path on stack
                    is now complete }
                if EMPTY(S) then
                    return;
                { explore right sibling of node
                    on top of stack }
                m := RIGHT_SIBLING(TOP(S), T);
                POP(S)
            end
    end;  { NPREORDER }
```

Fig. 3.9. A nonrecursive preorder procedure.

increasing order from left to right. On that assumption, we have written the function RIGHT_SIBLING in Fig. 3.11, for types node and TREE that are defined as follows:

```
type
    node = integer;
    TREE = array [1..maxnodes] of node;
```

For this implementation we assume the null node Λ is represented by 0.
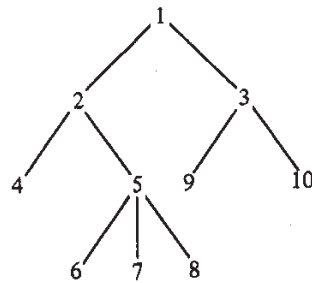
(a) a tree

(b) parent representation.

Fig. 3.10. A tree and its parent pointer representation.

```
function RIGHT_SIBLING ( n: node; T: TREE ) : node;
    { return the right sibling of node n in tree T }
  var
      i, parent: node;
  begin
      parent := T[n];
      for i := n + 1 to maxnodes do
              { search for node after n with same parent }
              if T[i] = parent then
                  return (i);
              return (0)  { null node will be returned
                  if no right sibling is ever found }
  end;  { RIGHT_SIBLING }
```

Fig. 3.11. Right sibling operation using array representation.

Representation of Trees by Lists of Children

An important and useful way of representing trees is to form for each node a list of its children. The lists can be represented by any of the methods suggested in Chapter 2, but because the number of children each node may have can be variable, the linked-list representations are often more appropriate.

Figure 3.12 suggests how the tree of Fig. 3.10(a) might be represented. There is an array of header cells, indexed by nodes, which we assume to be numbered 1, 2, . . . , 10. Each header points to a linked list of "elements," which are nodes. The elements on the list headed by $header[i]$ are the children of node i; for example, 9 and 10 are the children of 3.

header
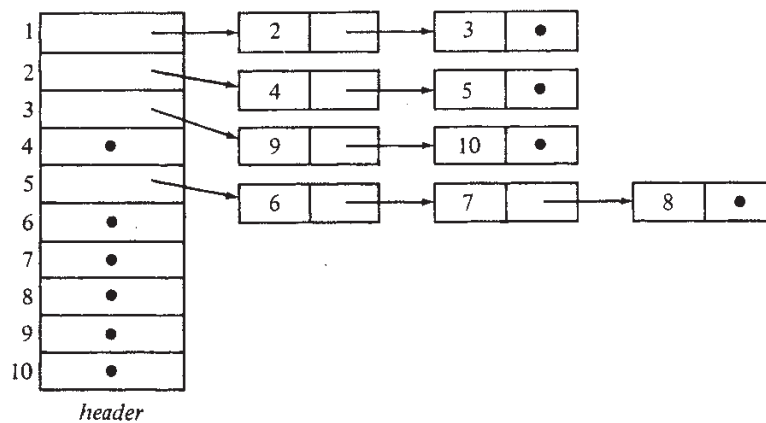
Fig. 3.12. A linked-list representation of a tree.

Let us first develop the data structures we need in terms of an abstract data type LIST (of nodes), and then give a particular implementation of lists and see how the abstractions fit together. Later, we shall see some of the simplifications we can make. We begin with the following type declarations:

```
type
    node = integer;
    LIST = { appropriate definition for list of nodes };
    position = { appropriate definition for positions in lists };
    TREE = record
        header: array [1..maxnodes] of LIST;
        labels: array [1..maxnodes] of labeltype;
        root: node
    end;
```

We assume that the root of each tree is stored explicitly in the *root* field. Also, 0 is used to represent the null node.

Figure 3.13 shows the code for the LEFTMOST_CHILD operation. The reader should write the code for the other operations as exercises.

```
function LEFTMOST_CHILD ( n : node; T : TREE ) : node;
    { returns the leftmost child of node n of tree T }
    var
        L : LIST;  { shorthand for the list of n 's children }
    begin
        L := T.header [n];
        if EMPTY(L) then  { n is a leaf }
            return (0)
        else
            return (RETRIEVE(FIRST(L), L))
    end;  { LEFTMOST_CHILD }
```

Fig. 3.13. Function to find leftmost child.

Now let us choose a particular implementation of lists, in which both LIST and position are integers, used as cursors into an array *cellspace* of records:

```
var
    cellspace : array [1..maxnodes ] of record
        node : integer;
        next : integer
    end;
```

To simplify, we shall not insist that lists of children have header cells. Rather, we shall let *T.header* [n] point directly to the first cell of the list, as is suggested by Fig. 3.12. Figure 3.14(a) shows the function LEFTMOST_CHILD of Fig. 3.13 rewritten for this specific implementation. Figure 3.14(b) shows the operator PARENT, which is more difficult to write using this representation of lists, since a search of all lists is required to determine on which list a given node appears.

The Leftmost-Child, Right-Sibling Representation

The data structure described above has, among other shortcomings, the inability to create large trees from smaller ones, using the CREATE*i* operators. The reason is that, while all trees share *cellspace* for linked lists of children, each has its own array of headers for its nodes. For example, to implement CREATE2(v, T_1, T_2) we would have to copy T_1 and T_2 into a third tree and add a new node with label v and two children — the roots of T_1 and T_2.

If we wish to build trees from smaller ones, it is best that the representation of nodes from all trees share one area. The logical extension of Fig. 3.12 is to replace the header array by an array *nodespace* consisting of records with

```
function LEFTMOST_CHILD ( n: node; T: TREE ) : node;
    { returns the leftmost child of node n on tree T }
    var
        L: integer;  { a cursor to the beginning of the list of n's children }
    begin
        L := T.header[n];
        if L = 0 then  { n is a leaf }
            return (0)
        else
            return (cellspace[L].node)
    end;  { LEFTMOST_CHILD }
```

(a) The function LEFTMOST_CHILD.

```
function PARENT ( n: node; T: TREE ) : node;
    { returns the parent of node n in tree T }
    var
        p: node;  { runs through possible parents of n }
        i: position;  { runs down list of p's children }
    begin
        for p := 1 to maxnodes do begin
            i := T.header[p];
            while i <> 0 do  { see if n is among children of p }
                if cellspace[i].node = n then
                    return (p)
                else
                    i := cellspace[i].next
        end;
        return (0)  { return null node if parent not found }
    end;  { PARENT }
```

(b) The function PARENT.

Fig. 3.14. Two functions using linked-list representation of trees.

two fields *label* and *header*. This array will hold headers for all nodes of all trees. Thus, we declare

```
var
    nodespace: array [1..maxnodes] of record
        label: labeltype;
        header: integer  { cursor to cellspace }
    end;
```

Then, since nodes are no longer named 1, 2, . . . ,n, but are represented by

21

arbitrary indices in *nodespace*, it is no longer feasible for the field *node* of *cellspace* to represent the "number" of a node; rather, *node* is now a cursor into *nodespace*, indicating the position of that node. The type TREE is simply a cursor into *nodespace*, indicating the position of the root.

Example 3.7. Figure 3.15(a) shows a tree, and Fig. 3.15(b) shows the data structure where we have placed the nodes labeled *A*, *B*, *C*, and *D* arbitrarily in positions 10, 5, 11, and 2 of *nodespace*. We have also made arbitrary choices for the cells of *cellspace* used for lists of children. □

(a) Tree *T*
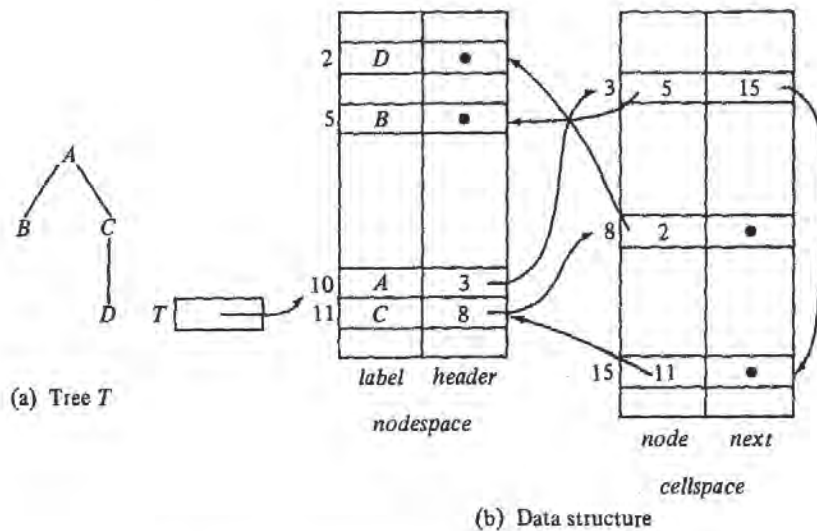
(b) Data structure

Fig. 3.15. Another linked-list structure for trees.

The structure of Fig. 3.15(b) is adequate to merge trees by the CREATE*i* operations. This data structure can be significantly simplified, however. First, observe that the chains of *next* pointers in *cellspace* are really right-sibling pointers.

Using these pointers, we can obtain leftmost children as follows. Suppose *cellspace*[*i*].*node* = *n*. (Recall that the "name" of a node, as opposed to its label, is in effect its index in *nodespace*, which is what *cellspace*[*i*].*node* gives us.) Then *nodespace*[*n*].*header* indicates the cell for the leftmost child of *n* in *cellspace*, in the sense that the *node* field of that cell is the name of that node in *nodespace*.

We can simplify matters if we identify a node not with its index in

22

nodespace, but with the index of the cell in *cellspace* that represents it as a child. Then, the *next* pointers of *cellspace* truly point to right siblings, and the information contained in the *nodespace* array can be held by introducing a field *leftmost_child* in *cellspace*. The datatype TREE becomes an integer used as a cursor to *cellspace* indicating the root of the tree. We declare *cellspace* to have the following structure.

```
var
    cellspace: array [1..maxnodes] of record
        label: labeltype;
        leftmost_child: integer;
        right_sibling: integer
    end;
```

Example 3.8. The tree of Fig. 3.15(a) is represented in our new data struc-
ture in Fig. 3.16. The same arbitrary indices as in Fig. 3.15(b) have been
used for the nodes. □
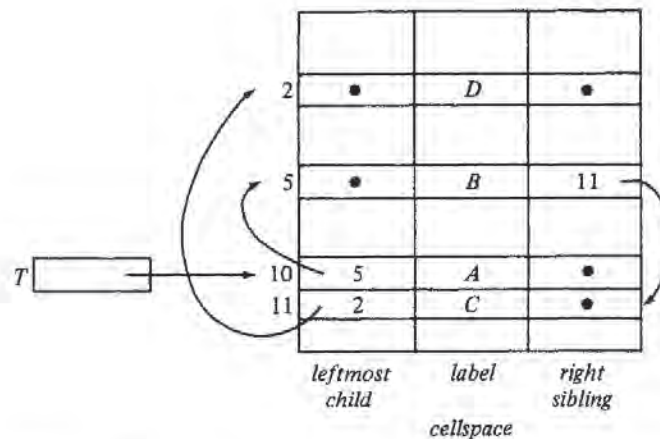
Fig. 3.16. Leftmost-child, right-sibling representation of a tree.

All operations but PARENT are straightforward to implement in the leftmost-child, right-sibling representation. PARENT requires searching the entire *cellspace*. If we need to perform the PARENT operation efficiently, we can add a fourth field to *cellspace* to indicate the parent of a node directly.

As an example of a tree operation written to use the leftmost-child, right-

sibling structure as in Fig. 3.16, we give the function CREATE2 in Fig. 3.17. We assume that unused cells are linked in an available space list, headed by *avail*, and that available cells are linked by their right-sibling fields. Figure 3.18 shows the old (solid) and the new (dashed) pointers.

```
function CREATE2 ( v: labeltype; T1, T2: integer ) : integer;
      { returns new tree with root v, having T1 and T2 as subtrees }
   var
      temp: integer;    { holds index of first available cell
            for root of new tree }
   begin
      temp := avail;
      avail := cellspace[avail].right_sibling;
      cellspace[temp].leftmost_child := T1;
      cellspace[temp].label := v;
      cellspace[temp].right_sibling := 0;
      cellspace[T1].right_sibling := T2;
      cellspace[T2].right_sibling := 0; { not necessary;
          that field should be 0 as the cell was formerly a root }
      return (temp)
   end;  { CREATE2 }
```

Fig. 3.17. The function CREATE2.
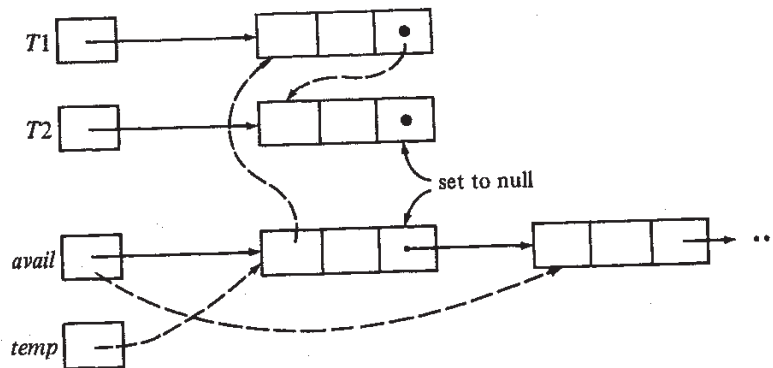
Fig. 3.18. Pointer changes produced by CREATE2.

Alternatively, we can use less space but more time if we put in the right-

sibling field of the rightmost child a pointer to the parent, in place of the null pointer that would otherwise be there. To avoid confusion, we need a bit in every cell indicating whether the right-sibling field holds a pointer to the right sibling or to the parent.

Given a node, we find its parent by following right-sibling pointers until we find one that is a parent pointer. Since all siblings have the same parent, we thereby find our way to the parent of the node we started from. The time required to find a node's parent in this representation depends on the number of siblings a node has.

3.4 Binary Trees

The tree we defined in Section 3.1 is sometimes called an *ordered, oriented* tree because the children of each node are ordered from left-to-right, and because there is an oriented path (path in a particular direction) from every node to its descendants. Another useful, and quite different, notion of "tree" is the *binary tree*, which is either an empty tree, or a tree in which every node has either no children, a *left child*, a *right child*, or both a left and a right child. The fact that each child in a binary tree is designated as a left child or as a right child makes a binary tree different from the ordered, oriented tree of Section 3.1.

Example 3.9. If we adopt the convention that left children are drawn extending to the left, and right children to the right, then Fig. 3.19 (a) and (b) represent two different binary trees, even though both "look like" the ordinary (ordered, oriented) tree of Fig. 3.20. However, let us emphasize that Fig. 3.19(a) and (b) are not the same binary tree, nor are either in any sense equal to Fig. 3.20, for the simple reason that binary trees are not directly comparable with ordinary trees. For example, in Fig. 3.19(a), 2 is the left child of 1, and 1 has no right child, while in Fig. 3.19(b), 1 has no left child but has 2 as a right child. In either binary tree, 3 is the left child of 2, and 4 is 2's right child. □

The preorder and postorder listings of a binary tree are similar to those of an ordinary tree given on p. 78. The inorder listing of the nodes of a binary tree with root n, left subtree T_1 and right subtree T_2 is the inorder listing of T_1 followed by n followed by the inorder listing of T_2. For example, 35241 is the inorder listing of the nodes of Fig. 3.19(a).

Representing Binary Trees

A convenient data structure for representing a binary tree is to name the nodes $1, 2, \ldots, n$, and to use an array of records declared

```
var
    cellspace: array [1..maxnodes] of record
        leftchild: integer;
        rightchild: integer
    end;
```
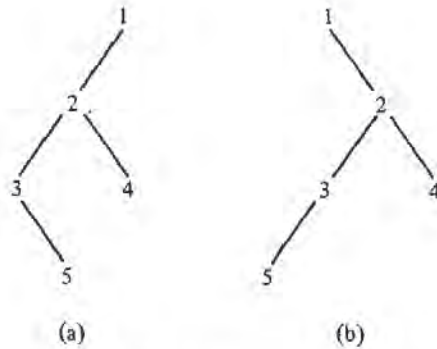
Fig. 3.19. Two binary trees.
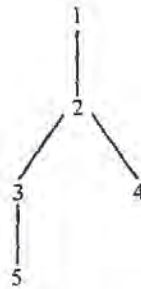
Fig. 3.20. An "ordinary" tree.

The intention is that *cellspace*[i].*leftchild* is the left child of node *i*, and *rightchild* is analogous. A value of 0 in either field indicates the absence of a child.

Example 3.10. The binary tree of Fig. 3.19(a) can be represented as shown in Fig. 3.21. □

An Example: Huffman Codes

Let us give an example of how binary trees can be used as a data structure. The particular problem we shall consider is the construction of "Huffman codes." Suppose we have messages consisting of sequences of characters. In each message, the characters are independent and appear with a known

	leftchild	rightchild
1	2	0
2	3	4
3	0	5
4	0	0
5	0	0

Fig. 3.21. Representation of a binary tree.

probability in any given position; the probabilities are the same for all positions. As an example, suppose we have a message made from the five characters a, b, c, d, e, which appear with probabilities .12, .4, .15, .08, .25, respectively.

We wish to encode each character into a sequence of 0's and 1's so that no code for a character is the prefix of the code for any other character. This *prefix property* allows us to decode a string of 0's and 1's by repeatedly deleting prefixes of the string that are codes for characters.

Example 3.11. Figure 3.22 shows two possible codes for our five symbol alphabet. Clearly Code 1 has the prefix property, since no sequence of three bits can be the prefix of another sequence of three bits. The decoding algorithm for Code 1 is simple. Just "grab" three bits at a time and translate each group of three into a character. Of course, sequences 101, 110, and 111 are impossible, if the string of bits really codes characters according to Code 1. For example, if we receive 001010011 we know the original message was *bcd*.

Symbol	Probability	Code 1	Code 2
a	.12	000	000
b	.40	001	11
c	.15	010	01
d	.08	011	001
e	.25	100	10

Fig. 3.22. Two binary codes.

It is easy to check that Code 2 also has the prefix property. We can decode a string of bits by repeatedly "grabbing" prefixes that are codes for characters and removing them, just as we did for Code 1. The only difference is that here, we cannot slice up the entire sequence of bits at once, because whether we take two or three bits for a character depends on the bits. For example, if a string begins 1101001, we can again be sure that the characters coded were *bcd*. The first two bits, 11, must have come from *b*, so we can

remove them and worry about 01001. We then deduce that the bits 01 came from c, and so on. □

The problem we face is: given a set of characters and their probabilities, find a code with the prefix property such that the average length of a code for a character is a minimum. The reason we want to minimize the average code length is to compress the length of an average message. The shorter the average code for a character is, the shorter the length of the encoded message. For example, Code 1 has an average code length of 3. This is obtained by multiplying the length of the code for each symbol by the probability of occurrence of that symbol. Code 2 has an average length of 2.2, since symbols a and d, which together appear 20% of the time, have codes of length three, and the other symbols have codes of length two.

Can we do better than Code 2? A complete answer to this question is to exhibit a code with the prefix property having an average length of 2.15. This is the best possible code for these probabilities of symbol occurrences. One technique for finding optimal prefix codes is called *Huffman's algorithm*. It works by selecting two characters a and b having the lowest probabilities and replacing them with a single (imaginary) character, say x, whose probability of occurrence is the sum of the probabilities for a and b. We then find an optimal prefix code for this smaller set of characters, using this procedure recursively. The code for the original character set is obtained by using the code for x with a 0 appended as the code for a and with a 1 appended as a code for b.

We can think of prefix codes as paths in binary trees. Think of following a path from a node to its left child as appending a 0 to a code, and proceeding from a node to its right child as appending a 1. If we label the leaves of a binary tree by the characters represented, we can represent any prefix code as a binary tree. The prefix property guarantees no character can have a code that is an interior node, and conversely, labeling the leaves of any binary tree with characters gives us a code with the prefix property for these characters.

Example 3.12. The binary trees for Code 1 and Code 2 of Fig. 3.22 are shown in Fig. 3.23(a) and (b), respectively. □

We shall implement Huffman's algorithm using a *forest* (collection of trees), each of which has its leaves labeled by characters whose codes we desire to select and whose roots are labeled by the sum of the probabilities of all the leaf labels. We call this sum the *weight* of the tree. Initially, each character is in a one-node tree by itself, and when the algorithm ends, there will be only one tree, with all the characters at its leaves. In this tree, the path from the root to any leaf represents the code for the label of that leaf, according to the left = 0, right = 1 scheme of Fig. 3.23.

The essential step of the algorithm is to select the two trees in the forest that have the smallest weights (break ties arbitrarily). Combine these two trees into one, whose weight is the sum of the weights of the two trees. To combine the trees we create a new node, which becomes the root and has the
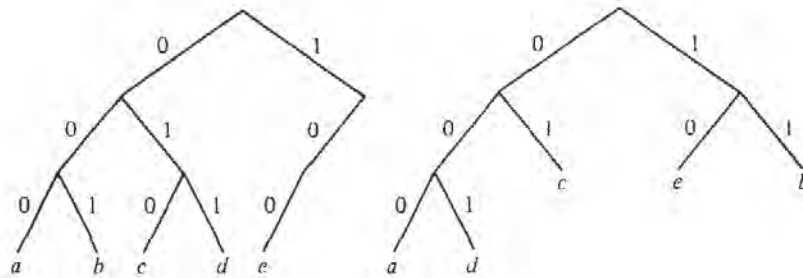
Fig. 3.23. Binary trees representing codes with the prefix property.

roots of the two given trees as left and right children (which is which doesn't matter). This process continues until only one tree remains. That tree represents a code that, for the probabilities given, has the minimum possible average code length.

Example 3.13. The sequence of steps taken for the characters and probabilities in our running example is shown in Fig. 3.24. From Fig. 3.24(e) we see the code words for a, b, c, d, and e are 1111, 0, 110, 1110, and 10. In this example, there is only one nontrivial tree, but in general, there can be many. For example, if the probabilities of b and e were .33 and .32, then after Fig. 3.24(c) we would combine b and e, rather than attaching e to the large tree as we did in Fig. 3.24(d). □

Let us now describe the needed data structures. First, we shall use an array *TREE* of records of the type

> **record**
> > *leftchild*: integer;
> > *rightchild*: integer;
> > *parent*: integer
> **end**

to represent binary trees. Parent pointers facilitate finding paths from leaves to roots, so we can discover the code for a character. Second, we use an array *ALPHABET* of records of type

> **record**
> > *symbol*: char;
> > *probability*: real;
> > *leaf*: integer { cursor into tree }
> **end**

.12 .40 .15 .08 .25

.

a   b   c   d   e

(a) Initial

.20   .40 .15 .25

d   a   b   c   e

(b) Merge a and d

.35

c

d   a

.40 .25

b   e

(c) Merge a, d with c

.60

e

c

d   a

.40

b

(d) Merge a, c, d with e

1.00

b

e

c

d   a

(e) Final tree
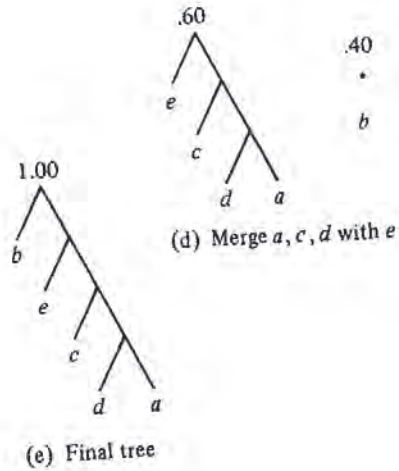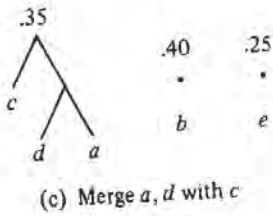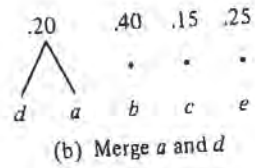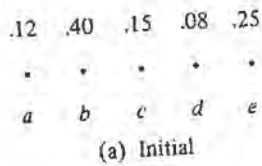
Fig. 3.24. Steps in the construction of a Huffman tree.

to associate, with each symbol of the alphabet being encoded, its correspond-
ing leaf. This array also records the probability of each character. Third, we
need an array *FOREST* of records that represent the trees themselves. The
type of these records is

```
record
    weight: real;
    root: integer  { cursor into tree }
end
```

The initial values of all these arrays, assuming the data of Fig. 3.24(a), are
shown in Fig. 3.25. A sketch of the program to build the Huffman tree is
shown in Fig. 3.26.

1	.12	1
2	.40	2
3	.15	3
4	.08	4
5	.25	5

weight   root

FOREST

1	a	.12	1
2	b	.40	2
3	c	.15	3
4	d	.08	4
5	e	.25	5

symbol   prob-   leaf
ability

ALPHABET

1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0

left-   right-   parent
child   child

TREE

Fig. 3.25. Initial contents of arrays.

(1)   **while** there is more then one tree in the forest **do**
         begin
(2) i := index of the tree in *FOREST* with smallest weight;
(3) j := index of the tree in *FOREST* with second smallest weight;
(4)   create a new node with left child *FOREST*[i].*root* and
             right child *FOREST*[j].*root*;
(5)   replace tree i in *FOREST* by a tree whose root
             is the new node and whose weight is
             FOREST[i].*weight* + *FOREST*[j].*weight*;
(6)   delete tree j from *FOREST*
       end;

Fig. 3.26. Sketch of Huffman tree construction.

To implement line (4) of Fig. 3.26, which increases the number of cells of
the *TREE* array used, and lines (5) and (6), which decrease the number of
utilized cells of *FOREST*, we shall introduce cursors *lasttree* and *lastnode*,
pointing to *FOREST* and *TREE*, respectively. We assume that cells 1 to *last-
tree* of *FOREST* and 1 to *lastnode* of *TREE* are occupied.† We assume that
arrays of Fig. 3.25 have some declared lengths, but in what follows we omit
comparisons between these limits and cursor values.

---

† For the data reading phase, which we omit, we also need a cursor for the array *ALPHA-
BET* as it fills with symbols and their probabilities.

```
procedure lightones ( var least, second: integer );
    { sets least and second to the indices in FOREST of
        the trees of smallest weight.  We assume lasttree ≥2. }
    var
        i: integer;
    begin    { initialize least and second, considering first two trees }
        if FOREST[1].weight <= FOREST[2].weight then
            begin least := 1; second := 2 end
        else
            begin least := 2; second := 1 end;
        { Now let i run from 3 to lasttree.  At each iteration
            least is the tree of smallest weight among the first i trees
            in FOREST, and second is the next smallest of these }
        for i := 3 to lasttree do
            if FOREST[i].weight < FOREST[least].weight then
                begin second := least; least := i end
            else if FOREST[i].weight < FOREST[second].weight then
                second := i
    end;  { lightones }


function create ( lefttree, righttree: integer ) : integer;
    { returns new node whose left and right children are
        FOREST[lefttree].root and FOREST[righttree].root }
    begin
        lastnode := lastnode + 1;
            { cell for new node is TREE[lastnode] }
        TREE[lastnode].leftchild := FOREST[lefttree].root;
        TREE[lastnode].rightchild := FOREST[righttree].root;
        { now enter parent pointers for new node and its children }
        TREE[lastnode].parent := 0;
        TREE[FOREST[lefttree].root].parent := lastnode;
        TREE[FOREST[righttree].root].parent := lastnode;
        return (lastnode)
    end;  { create }
```

Fig. 3.27. Two procedures.

Figure 3.27 shows two useful procedures. The first implements lines (2) and (3) of Fig. 3.26 to select indices of the two trees of smallest weight. The second is the command $create(n_1, n_2)$ that creates a new node and makes n_1 and n_2 its left and right children.

Now the steps of Fig. 3.26 can be described in greater detail. A procedure *Huffman*, which has no input or output, but works on the global structures of Fig. 3.25, is shown in Fig. 3.28.

```
procedure Huffman;
    var
        i, j: integer;  { the two trees of least weight in FOREST }
        newroot: integer;
    begin
        while lasttree > 1 do begin
            lightones(i, j);
            newroot := create(i, j);
            { Now replace tree i by the tree whose root is newroot }
            FOREST[i].weight := FOREST[i].weight + FOREST[j].weight;
            FOREST[i].root := newroot;
            { next, replace tree j, which is no longer needed, by lasttree,
                and shrink FOREST by one }
            FOREST[j] := FOREST[lasttree];
            lasttree := lasttree − 1
        end
    end;  { Huffman }
```

Fig. 3.28. Huffman's algorithm.

Figure 3.29 shows the data structure of Fig. 3.25 after *lasttree* has been reduced to 3, that is, when the forest looks like Fig. 3.24(c).
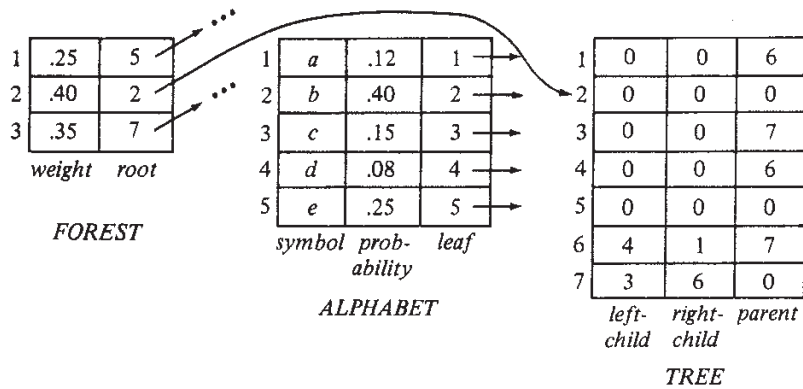
Fig. 3.29. Tree data structure after two iterations.

After completing execution of the algorithm, the code for each symbol can be determined as follows. Find the symbol in the *symbol* field of the *ALPHA-BET* array. Follow the *leaf* field of the same record to get to a record of the *TREE* array; this record corresponds to the leaf for that symbol. Repeatedly follow the *parent* pointer from the "current" record, say for node *n*, to the record of the *TREE* array for its parent *p*. Remember node *n*, so it is possible to examine the *leftchild* and *rightchild* pointers for node *p* and see which is *n*. In the former case, print 0, in the latter print 1. The sequence of bits printed is the code for the symbol, in reverse. If we wish the bits printed in the correct order, we could push each onto a stack as we go up the tree, and then repeatedly pop the stack, printing symbols as we pop them.

Pointer-Based Implementations of Binary Trees

Instead of using cursors to point to left and right children (and parents if we wish), we can use true Pascal pointers. For example, we might declare

```
type
    node = record
        leftchild: ↑ node;
        rightchild: ↑ node;
        parent: ↑ node
    end
```

For example, if we used this type for nodes of a binary tree, the function *create* of Fig. 3.27 could be written as in Fig. 3.30.

```
function create ( lefttree, righttree: ↑ node ) : ↑ node;
    var
        root: ↑ node;
    begin
        new(root);
        root↑.leftchild := lefttree;
        root↑.rightchild := righttree;
        root↑.parent := 0;
        lefttree↑.parent := root;
        righttree↑.parent := root;
        return (root)
    end; { create }
```

Fig. 3.30. Pointer-based implementation of binary trees.

34

Exercises

3.1 Answer the following questions about the tree of Fig. 3.31.

    a) Which nodes are leaves?

    b) Which node is the root?

    c) What is the parent of node C?

    d) Which nodes are children of C?

    e) Which nodes are ancestors of E?

    f) Which nodes are descendants of E?

    g) What are the right siblings of D and E?

    h) Which nodes are to the left and to the right of G?

    i) What is the depth of node C?

    j) What is the height of node C?
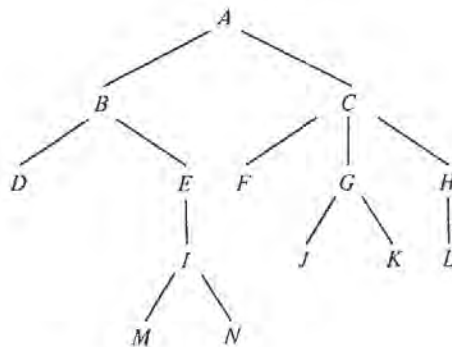
Fig. 3.31. A tree.

3.2 In the tree of Fig. 3.31 how many different paths of length three are there?

3.3 Write programs to compute the height of a tree using each of the three tree representations of Section 3.3.

3.4 List the nodes of Fig. 3.31 in

    a) preorder,

b) postorder, and

c) inorder.

3.5 If m and n are two different nodes in the same tree, show that exactly one of the following statements is true:

a) m is to the left of n

b) m is to the right of n

c) m is a proper ancestor of n

d) m is a proper descendant of n.

3.6 Place a check in row i and column j if the two conditions represented by row i and column j can occur simultaneously.

	preorder(n) < preorder(m)	inorder(n) < inorder(m)	postorder(n) < postorder(m)
n is to the left of m			
n is to the right of m			
n is a proper ancestor of m			
n is a proper descendant of m			

For example, put a check in row 3 and column 2 if you believe that n can be a proper ancestor of m and at the same time n can precede m in inorder.

3.7 Suppose we have arrays PREORDER[n], INORDER[n], and POSTORDER[n] that give the preorder, inorder, and postorder positions, respectively, of each node n of a tree. Describe an algorithm that tells whether node i is an ancestor of node j, for any pair of nodes i and j. Explain why your algorithm works.

*3.8 We can test whether a node m is a proper ancestor of a node n by testing whether m precedes n in X-order but follows n in Y-order, where X and Y are chosen from {pre, post, in}. Determine all those pairs X and Y for which this statement holds.

3.9 Write programs to traverse a binary tree in

a) preorder,

b) postorder,

c) inorder.

3.10 The *level-order* listing of the nodes of a tree first lists the root, then all nodes of depth 1, then all nodes of depth 2, and so on. Nodes at the same depth are listed in left-to-right order. Write a program to list the nodes of a tree in level-order.

3.11 Convert the expression $((a+b) + c * (d+e) + f) * (g+h)$ to a

  a) prefix expression

  b) postfix expression.

3.12 Draw tree representations for the prefix expressions

  a) $*a+b*c+de$

  b) $*a+*b+cde$

3.13 Let T be a tree in which every nonleaf node has two children. Write a program to convert

  a) a preorder listing of T into a postorder listing,

  b) a postorder listing of T into a preorder listing,

  c) a preorder listing of T into an inorder listing.

3.14 Write a program to evaluate

  a) preorder

  b) postorder

  arithmetic expressions.

3.15 We can define a binary tree as an ADT with the binary tree structure as a mathematical model and with operations such as LEFTCHILD(n), RIGHTCHILD(n), PARENT(n), and NULL(n). The first three operations return the left child, the right child, and the parent of node n (Λ if there is none) and the last returns true if and only if n is Λ. Implement these procedures using the binary tree representation of Fig. 3.21.

3.16 Implement the seven tree operations of Section 3.2 using the following tree implementations:

  a) parent pointers

  b) lists of children

  c) leftmost-child, right-sibling pointers.

3.17 The *degree* of a node is the number of children it has. Show that in any binary tree the number of leaves is one more than the number of nodes of degree two.

3.18 Show that the maximum number of nodes in a binary tree of height h is $2^{h+1}-1$. A binary tree of height h with the maximum number of nodes is called a *full* binary tree.

*3.19 Suppose trees are implemented by leftmost-child, right-sibling and parent pointers. Give nonrecursive preorder, postorder, and inorder traversal algorithms that do not use "states" or a stack, as Fig. 3.9 does.

3.20 Suppose characters a, b, c, d, e, f have probabilities .07, .09, .12, .22, .23, .27, respectively. Find an optimal Huffman code and draw the Huffman tree. What is the average code length?

*3.21 Suppose T is a Huffman tree, and that the leaf for symbol a has greater depth than the leaf for symbol b. Prove that the probability of symbol b is no less than that of a.

*3.22 Prove that Huffman's algorithm works, i.e., it produces an optimal code for the given probabilities. *Hint*: Use Exercise 3.21.

Bibliographic Notes

Berge [1958] and Harary [1969] discuss the mathematical properties of trees. Knuth [1973] and Nievergelt [1974] contain additional information on binary search trees. Many of the works on graphs and applications referenced in Chapter 6 also cover material on trees.

The algorithm given in Section 3.4 for finding a tree with a minimal weighted path length is from Huffman [1952]. Parker [1980] gives some more recent explorations into that algorithm.

CHAPTER 5

Advanced
Set
Representation
Methods

This chapter introduces data structures for sets that permit more efficient implementation of common collections of set operations than those of the previous chapter. These structures, however, are more complex and are often only appropriate for large sets. All are based on various kinds of trees, such as binary search trees, tries, and balanced trees.

5.1 Binary Search Trees

We shall begin with binary search trees, a basic data structure for representing sets whose elements are ordered by some linear order. We shall, as usual, denote that order by $<$. This structure is useful when we have a set of elements from a universe so large that it is impractical to use the elements of the set themselves as indices into arrays. An example of such a universe would be the set of possible identifiers in a Pascal program. A binary search tree can support the set operations INSERT, DELETE, MEMBER, and MIN, taking $O(\log n)$ steps per operation on the average for a set of n elements.

A *binary search tree* is a binary tree in which the nodes are labeled with elements of a set. The important property of a binary search tree is that all elements stored in the left subtree of any node x are all less than the element stored at x, and all elements stored in the right subtree of x are greater than the element stored at x. This condition, called the *binary search tree property*, holds for every node of a binary search tree, including the root.

Figure 5.1 shows two binary search trees representing the same set of integers. Note the interesting property that if we list the nodes of a binary search tree in inorder, then the elements stored at those nodes are listed in sorted order.

Suppose a binary search tree is used to represent a set. The binary search tree property makes testing for membership in the set simple. To determine whether x is a member of the set, first compare x with the element r at the root of the tree. If $x = r$ we are done and the answer to the membership query is "true." If $x < r$, then by the binary search tree property, x can only
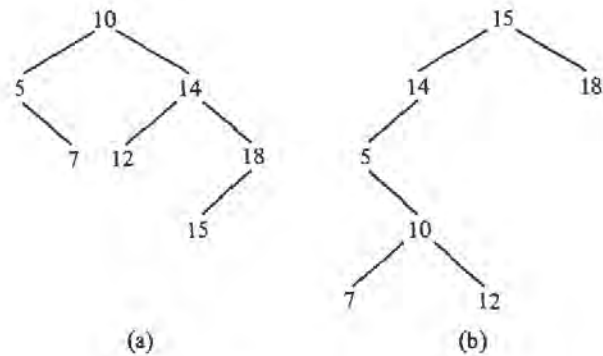
Fig. 5.1. Two binary search trees.

be a descendant of the left child of the root, if x is present at all.† Similarly, if $x > r$, then x could only be at a descendant of the right child of the root.

We shall write a simple recursive function MEMBER(x, A) to implement this membership test. We assume the elements of the set are of an unspeci- fied type that will be called elementtype. For convenience, we assume elementtype is a type for which $<$ and $=$ are defined. If not, we must define functions LT(a, b) and EQ(a, b), where a and b are of type elementtype, such that LT(a, b) is true if and only if a is "less than" b, and EQ(a, b) is true if and only if a and b are the same.

The type for nodes consists of an element and two pointers to other nodes:

```
type
    nodetype = record
        element: elementtype;
        leftchild, rightchild: ↑ nodetype
    end;
```

Then we can define the type SET as a pointer to a node, which we take to be the root of the binary search tree representing the set. That is:

---

† Recall the left child of the root is a descendant of itself, so we have not ruled out the possibility that x is at the left child of the root.

```
type
    SET = ↑ nodetype;
```

Now we can specify fully the function MEMBER, in Fig. 5.2. Notice that
since SET and "pointer to nodetype" are synonymous, MEMBER can call
itself on subtrees as if those subtrees represented sets. In effect, the set can
be subdivided into the subset of members less than x and the subset of
members greater than x.

```
function MEMBER ( x: elementtype; A: SET ) : boolean;
    { returns true if x is in A, false otherwise }
    begin
        if A = nil then
            return (false) { x is never in ∅ }
        else if x = A↑.element then
            return (true)
        else if x < A↑.element then
            return (MEMBER(x, A↑.leftchild))
        else { x > A↑.element }
            return (MEMBER(x, A↑.rightchild))
    end; { MEMBER }
```

Fig. 5.2. Testing membership in a binary search tree.

The procedure INSERT(x, A), which adds element x to set A, is also
easy to write. The first action INSERT must do is test whether A = nil, that
is, whether the set is empty. If so, we create a new node to hold x and make
A point to it. If the set is not empty, we search for x more or less as
MEMBER does, but when we find a nil pointer during our search, we replace
it by a pointer to a new node holding x. Then x will be in the right place,
namely, the place where the function MEMBER will find it. The code for
INSERT is shown in Fig. 5.3.

Deletion presents some problems. First, we must locate the element x to
be deleted in the tree. If x is at a leaf, we can delete that leaf and be done.
However, x may be at an interior node *inode*, and if we simply deleted *inode*,
we would disconnect the tree.

If *inode* has only one child, as the node numbered 14 in Fig. 5.1(b), we
can replace *inode* by that child, and we shall be left with the appropriate
binary search tree. If *inode* has two children, as the node numbered 10 in
Fig. 5.1(a), then we must find the lowest-valued element among the descen-
dants of the right child.† For example, in the case the element 10 is deleted
from Fig. 5.1(a), we must replace it by 12, the minimum-valued descendant of

† The highest-valued node among the descendants of the left child would do as well.

```
procedure INSERT ( x: elementtype; var A: SET );
    { add x to set A }
    begin
        if A = nil then begin
            new(A);
            A↑.element := x;
            A↑.leftchild := nil;
            A↑.rightchild := nil
        end
        else if x < A↑.element then
            INSERT(x, A↑.leftchild)
        else if x > A↑.element then
            INSERT (x, A↑.rightchild)
        { if x = A↑.element, we do nothing; x is already in the set }
    end; { INSERT }
```

Fig. 5.3. Inserting an element into a binary search tree.

the right child of 10.

To write DELETE, it is useful to have a function DELETEMIN(A) that removes the smallest element from a nonempty tree and returns the value of the element removed. The code for DELETEMIN is shown in Fig. 5.4. The code for DELETE uses DELETEMIN and is shown in Fig. 5.5.

```
function DELETEMIN ( var A: SET ) : elementtype;
    { returns and removes the smallest element from set A }
    begin
        if A↑.leftchild = nil then begin
            { A points to the smallest element }
            DELETEMIN := A↑.element;
            A := A↑.rightchild
                { replace the node pointed to by A by its right child }
        end
        else { the node pointed to by A has a left child }
            DELETEMIN := DELETEMIN(A↑.leftchild)
    end; { DELETEMIN }
```

Fig. 5.4. Deleting the smallest element.

Example 5.1. Suppose we try to delete 10 from Fig. 5.1(a). Then, in the last statement of DELETE we call DELETEMIN with argument a pointer to node 14. That pointer is the *rightchild* field in the root. That call results in another call to DELETEMIN. The argument is then a pointer to node 12; this pointer is found in the *leftchild* field of node 14. We find that 12 has no

42

```
procedure DELETE ( x: elementtype; var A: SET );
    { remove x from set A }
begin
    if A <> nil then
        if x < A↑.element then
            DELETE(x, A↑.leftchild)
        else if x > A↑.element then
            DELETE(x, A↑.rightchild)
        { if we reach here, x is at the node pointed to by A }
        else if (A↑.leftchild = nil) and (A↑.rightchild = nil) then
            A := nil  { delete the leaf holding x }
        else if A↑.leftchild = nil then
            A := A↑.rightchild
        else if A↑.rightchild = nil then
            A := A↑.leftchild
        else { both children are present }
            A↑.element := DELETEMIN(A↑.rightchild)
end; { DELETE }
```

Fig. 5.5. Deletion from a binary search tree.

left child, so we return element 12 and make the left child for 14 be the right child of 12, which happens to be **nil**. Then DELETE takes the value 12 returned by DELETEMIN and replaces 10 by it. The resulting tree is shown in Fig. 5.6. □
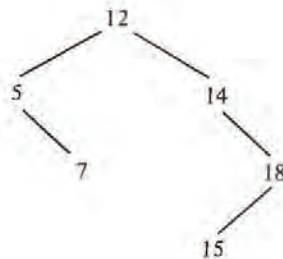

Fig. 5.6. Tree of Fig. 5.1(a) after deleting 10.

5.2 Time Analysis of Binary Search Tree Operations

In this section we analyze the average behavior of various binary search tree operations. We show that if we insert n random elements into an initially empty binary search tree, then the average path length from the root to a leaf is $O(\log n)$. Testing for membership, therefore, takes $O(\log n)$ time.

It is easy to see that if a binary tree of n nodes is complete (all nodes, except those at the lowest level have two children), then no path has more than $1+\log n$ nodes.† Thus, each of the procedures MEMBER, INSERT, DELETE, and DELETEMIN takes $O(\log n)$ time. To see this, observe that they all take a constant amount of time at a node, then may call themselves recursively on at most one child. Therefore, the sequence of nodes at which calls are made forms a path from the root. Since that path is $O(\log n)$ in length, the total time spent following the path is $O(\log n)$.

However, when we insert n elements in a "random" order, they do not necessarily arrange themselves into a complete binary tree. For example, if we happen to insert smallest first, in sorted order, then the resulting tree is a chain of n nodes, in which each node except the lowest has a right child but no left child. In this case, it is easy to show that, as it takes $O(i)$ steps to insert the ith element, and $\sum_{i=1}^{n} i = n(n+1)/2$, the whole process of n insertions takes $O(n^2)$ steps, or $O(n)$ steps per operation.

We must determine whether the "average" binary search tree of n nodes is closer to the complete tree in structure than to the chain, that is, whether the average time per operation on a "random" tree takes $O(\log n)$ steps, $O(n)$ steps, or something in between. As we cannot know the true frequency of insertions and deletions, or whether deleted elements have some special property (e.g., do we always delete the minimum?), we can only analyze the average path length of "random" trees if we make some assumptions. The particular assumptions we make are that trees are formed by insertions only, and all orders of the n inserted elements are equally likely.

Under these fairly natural assumptions, we can calculate $P(n)$, the average number of nodes on the path from the root to some node (not necessarily a leaf). We assume the tree was formed by inserting n random elements into an initially empty tree. Clearly $P(0) = 0$ and $P(1) = 1$. Suppose we have a list of $n \geq 2$ elements to insert into an empty tree. The first element on the list, call it a, is equally likely to be first, second, or nth in the sorted order. Suppose that i elements on the list are less than a, so $n-i-1$ are greater than a. When we build the tree, a will appear at the root, the i smaller elements will be descendants of the left child of the root, and the remaining $n-i-1$ will be descendants of the right child. This tree is sketched in Fig. 5.7.

As all orders for the i small elements and for the $n-i-1$ large elements are equally likely, we expect that the left and right subtrees of the root will

---

† Recall that all logarithms are to the base 2 unless otherwise noted.
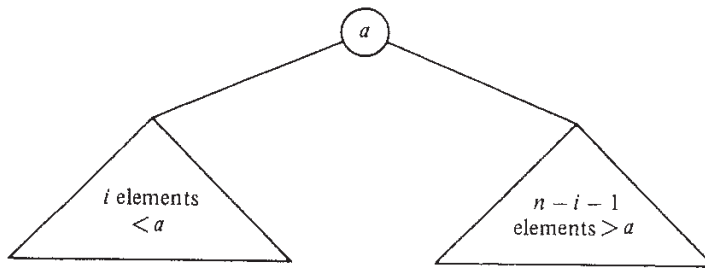
Fig. 5.7. Binary search tree.

have average path lengths $P(i)$ and $P(n-i-1)$, respectively. Since these elements are reached through the root of the complete tree, we must add 1 to the number of nodes on every path. Thus $P(n)$ can be calculated by averaging, for all i between 0 and $n-1$, the sum

$$\frac{i}{n}(P(i)+1) + \frac{(n-i-1)}{n} (P(n-i-1)+1) + \frac{1}{n}$$

The first term is the average path length in the left subtree, weighted by its size. The second term is the analogous quantity for the right subtree, and the $1/n$ term represents the contribution of the root. By averaging the above sum for all i between 1 and n, we obtain the recurrence

$$P(n) = 1 + \frac{1}{n^2} \sum_{i=0}^{n-1} \Big(iP(i) + (n-i-1) P(n-i-1)\Big) \qquad (5.1)$$

The first part of the summation (5.1), $\sum_{i=0}^{n-1} iP(i)$, can be made identical to the second part $\sum_{i=0}^{n-1}(n-i-1)P(n-i-1)$ if we substitute i for $n-i-1$ in the second part. Also, the term for $i=0$ in the summation $\sum_{i=0}^{n-1} iP(i)$ is zero, so we can begin the summation at 1. Thus (5.1) can be written

$$P(n) = 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} iP(i) \quad \text{for } n \geq 2 \qquad (5.2)$$

We shall show by induction on n, starting at $n=1$, that $P(n) \leq 1 + 4\log n$. Surely this statement is true for $n = 1$, since $P(1) = 1$. Suppose it is true for all $i < n$. Then by (5.2)

$$P(n) \leq 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} (4i\log i + i)$$

45

$$\leq 1 + \frac{2}{n^2}\sum_{i=1}^{n-1} 4i\log i + \frac{2}{n^2}\sum_{i=1}^{n-1} i$$

$$\leq 2 + \frac{8}{n^2}\sum_{i=1}^{n-1} i\log i \tag{5.3}$$

The last step is justified, since $\sum_{i=1}^{n-1} i \leq n^2/2$, and therefore, the last term of the second line is at most 1. We shall divide the terms in the summation of (5.3) into two parts, those for $i \leq \lceil n/2\rceil - 1$, which do not exceed $i\log(n/2)$, and those for $i > \lceil n/2\rceil - 1$, which do not exceed $i\log n$. Thus (5.3) can be rewritten

$$P(n) \leq 2 + \frac{8}{n^2}\left[\sum_{i=1}^{\lceil n/2\rceil-1} i\log(n/2) + \sum_{i=\lceil n/2\rceil}^{n-1} i\log n\right] \tag{5.4}$$

Whether n is even or odd, one can show that the first sum of (5.4) does not exceed $(n^2/8)\log(n/2)$, which is $(n^2/8)\log n - (n^2/8)$, and the second sum does not exceed $(3n^2/8)\log n$. Thus, we can rewrite (5.4) as

$$P(n) \leq 2 + \frac{8}{n^2}\left[\frac{n^2}{2}\log n - \frac{n^2}{8}\right]$$

$$\leq 1 + 4\log n$$

as we wished to prove. This step completes the induction and shows that the average time to follow a path from the root to a random node of a binary search tree constructed by random insertions is $O(\log n)$, that is to within a constant factor as good as if the tree were complete. A more careful analysis shows that the constant 4 above is really about 1.4.

We can conclude from the above that the time of membership testing for a random member of the set takes $O(\log n)$ time. A similar analysis shows that if we include in our average path length only those nodes that are missing both children, or only those missing left children, then the average path length still obeys an equation similar to (5.1), and is therefore $O(\log n)$. We may then conclude that testing membership of a random element not in the set, inserting a random new element, and deleting a random element also all take $O(\log n)$ time on the average.

Evaluation of Binary Search Tree Performance

Hash table implementations of dictionaries require constant time per operation on the average. Although this performance is better than that for a binary search tree, a hash table requires $O(n)$ steps for the MIN operation, so if MIN is used frequently, the binary search tree will be the better choice; if MIN is not needed, we would probably prefer the hash table.

The binary search tree should also be compared with the partially ordered tree used for priority queues in Chapter 4. A partially ordered tree with n

elements requires only $O(\log n)$ steps for each INSERT and DELETEMIN operation not only on the average, but also in the worst case. Moreover, the actual constant of proportionality in front of the $\log n$ factor will be smaller for a partially ordered tree than for a binary search tree. However, the binary search tree permits general DELETE and MIN operations, as well as the combination DELETEMIN, while the partially ordered tree permits only the latter. Moreover, MEMBER requires $O(n)$ steps on a partially ordered tree but only $O(\log n)$ steps on a binary search tree. Thus, while the partially ordered tree is well suited to implementing priority queues, it cannot do as efficiently any of the additional operations that the binary search tree can do.

5.3 Tries

In this section we shall present a special structure for representing sets of character strings. The same method works for representing data types that are strings of objects of any type, such as strings of integers. This structure is known as the *trie*, derived from the middle letters of the word "retrieval."† By way of introduction, consider the following use of a set of character strings.

Example 5.2. As indicated in Chapter 1, one way to implement a spelling checker is to read a text file, break it into words (character strings separated by blanks and new lines), and find those words not in a standard dictionary of words in common use. Words in the text but not in the dictionary are printed out as possible misspellings. In Fig. 5.8 we see a sketch of one possible program *spell*. It makes use of a procedure $getword(x, f)$ that sets x to be the next word in text file f; variable x is of a type called wordtype, which we shall define later. The variable A is of type SET; the SET operations we shall need are INSERT, DELETE, MAKENULL, and PRINT. The PRINT operator prints the members of the set. □

The trie structure supports these set operations when the elements of the set are words, i.e., character strings. It is appropriate when many words begin with the same sequences of letters, that is, when the number of distinct prefixes among all the words in the set is much less than the total length of all the words.

In a trie, each path from the root to a leaf corresponds to one word in the represented set. This way, the nodes of the trie correspond to the prefixes of words in the set. To avoid confusion between words like THE and THEN, let us add a special *endmarker* symbol, $, to the ends of all words, so no prefix of a word can be a word itself.

Example 5.3. In Fig. 5.9 we see a trie representing the set of words {THE, THEN, THIN, THIS, TIN, SIN, SING}. That is, the root corresponds to the empty string, and its two children correspond to the prefixes T and S. The

---

† Trie was originally intended to be a homonym of "tree" but to distinguish these two terms many people prefer to pronounce trie as though it rhymes with "pie."

```
program spell ( input, output, dictionary );
    type
        wordtype = { to be defined };
        SET = { to be defined, using the trie structure };
    var
        A: SET;  { holds input words not yet found in the dictionary }
        nextword: wordtype;
        dictionary: file of char;

    procedure  getword ( var x: wordtype; f: file of char );
        { a procedure to be defined that sets x
            to be the next word in file f }

    procedure INSERT ( x: wordtype; var S: SET );
        { to be defined }

    procedure DELETE (x: wordtype; var S: SET );
        { to be defined }

    procedure MAKENULL ( var S: SET );
        { to be defined }

    procedure PRINT ( var S: SET );
        { to be defined }

    begin
        MAKENULL(A);
        while not eof (input) do begin
            getword(nextword, input);
            INSERT(nextword, A)
        end;
        while not eof (dictionary) do begin
            getword(nextword, dictionary);
            DELETE(nextword, A)
        end;
        PRINT(A)
    end; { spell }
```

Fig. 5.8. Sketch of spelling checker.

leftmost leaf represents the word THE, the next leaf the word THEN, and so on. □

We can make the following observations about the trie in Fig. 5.9.

1. Each node has at most 27 children, one for each letter and $.
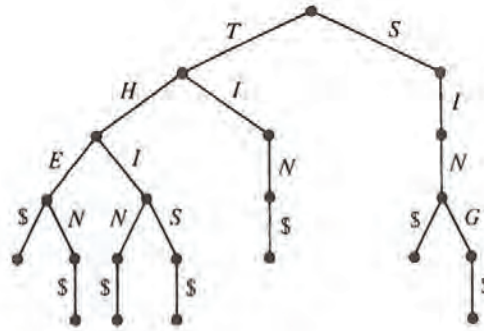
Fig. 5.9. A trie.

2. Most nodes will have many fewer than 27 children.

3. A leaf reached by an edge labeled $ cannot have any children, and may as well not be there.

Trie Nodes as ADT's

We can view a node of a trie as a mapping whose domain is {A, B, . . . , Z, $} (or whatever alphabet we choose) and whose value set is the type "pointer to trie node." Moreover, the trie itself can be identified with its root, so the ADT's TRIE and TRIENODE have the same data type, although their operations are substantially different. On TRIENODE we need the following operations:

1. procedure ASSIGN(*node, c, p*) that assigns value *p* (a pointer to a node) to character *c* in node *node*,

2. function VALUEOF(*node, c*) that produces the value associated with character *c* in *node*,† and

3. procedure GETNEW(*node, c*) to make the value of *node* for character *c* be a pointer to a new node.

Technically, we also need a procedure MAKENULL(*node*) to make *node* be the null mapping. One simple implementation of trie nodes is an array *node* of pointers to nodes, with the index set being {A, B, . . . , Z, $}. That is, we define

---

† VALUEOF is a function version of COMPUTE in Section 2.5.

```
type
    chars = ('A', 'B', . . . , 'Z', '$');
    TRIENODE = array[chars] of ↑ TRIENODE;
```

If *node* is a trie node, *node*[*c*] is VALUEOF(*node*, *c*) for any *c* in the set
chars. To avoid creating many leaves that are children corresponding to '$',
we shall adopt the convention that *node*['$'] is either **nil** or a pointer to the
node itself. In the former case, *node* has no child corresponding to '$', and in
the latter case it is deemed to have such a child, although we never create it.
Then we can write the procedures for trie nodes as in Fig. 5.10.

```
procedure MAKENULL ( var node: TRIENODE );
    { makes node a leaf, i.e., a null mapping }
    var
        c: chars;
    begin
        for c := 'A' to '$' do
            node[c] := nil
    end; { MAKENULL }

procedure ASSIGN ( var node: TRIENODE; c: chars; p: ↑ TRIENODE );
    begin
        node[c] := p
    end; { ASSIGN }

function VALUEOF ( var node: TRIENODE; c: chars ) : ↑ TRIENODE;
    begin
        return (node[c])
    end; { VALUEOF }

procedure GETNEW ( var node: TRIENODE; c: chars );
    begin
        new(node[c]);
        MAKENULL(node[c])
    end; { GETNEW }
```

Fig. 5.10. Operations on trie nodes.

Now let us define

```
type
    TRIE = ↑ TRIENODE;
```

We shall assume wordtype is an array of characters of some fixed length. The

value of such an array will always be assumed to have at least one '$'; we take the end of the represented word to be the first '$', no matter what follows (presumably more '$''s). On this assumption, we can write the procedure INSERT(x, *words*) to insert x into set *words* represented by a trie, as shown in Fig. 5.11. We leave the writing of MAKENULL, DELETE, and PRINT for tries represented as arrays for exercises.

```
procedure INSERT ( x: wordtype; var words: TRIE );
    var
        i: integer; { counts positions in word x }
        t: TRIE; { used to point to trie nodes
            corresponding to prefixes of x }
    begin
        i := 1;
        t := words;
        while x[i] <> '$' do begin
            if VALUEOF(t↑, x[i]) = nil then
                { if current node has no child for character x[i],
                    create one }
                GETNEW(t↑, x[i]);
            t := VALUEOF(t↑, x[i]);
                { proceed to the child of t for character x[i],
                    whether or not that child was just created }
            i := i+1  { move along the word x }
        end;
        { now we have reached the first '$' in x }
        ASSIGN(t↑, '$', t)
            { make loop for '$' to represent a leaf }
    end; {INSERT}
```

Fig. 5.11. The procedure INSERT.

A List Representation for Trie Nodes

The array representation of trie nodes takes a collection of words, having among them p different prefixes, and represents them with 27p bytes of storage. That amount of space could far exceed the total length of the words in the set. However, there is another implementation of tries that may save space. Recall that each trie node is a mapping, as discussed in Section 2.6. In principle, any implementation of mappings would do. Yet in practice, we want a representation suitable for mappings with a small domain and for mappings defined for comparatively few members of that domain. The linked list representation for mappings satisfies these requirements nicely. We may represent the mapping that is a trie node by a linked list of the characters for which the associated value is not the **nil** pointer. That is, a trie node is a

linked list of cells of the type

```
type
    celltype = record
        domain: chars;
        value: ↑ celltype;
            { pointer to first cell on list for the child node }
        next: ↑ celltype
            { pointer to next cell on the list }
    end;
```

We shall leave the procedures ASSIGN, VALUEOF, MAKENULL, and GETNEW for this implementation of trie nodes as exercises. After writing these procedures, the INSERT operations on tries, in Fig. 5.11, and the other operations on tries that we left as exercises, should work correctly.

Evaluation of the Trie Data Structure

Let us compare the time and space needed to represent n words with a total of p different prefixes and a total length of l using a hash table and a trie. In what follows, we shall assume that pointers require four bytes. Perhaps the most space-efficient way to store words and still support INSERT and DELETE efficiently is in a hash table. If the words are of varying length, the cells of the buckets should not contain the words themselves; rather, the cells consist of two pointers, one to link the cells of the bucket and the other pointing to the beginning of a word belonging in the bucket.

The words themselves are stored in a large character array, and the end of each word is indicated by an endmarker character such as '$'. For example, the words THE, THEN, and THIN could be stored as

<div align="center">THE$THEN$THIN$ . . .</div>

The pointers for the three words are cursors to positions 1, 5, and 10 of the array. The amount of space used in the buckets and character array is

1. $8n$ bytes for the cells of the buckets, there being one cell for each of the n words, and a cell has two pointers or 8 bytes,
2. $l + n$ bytes for the character array to store the n words of total length l and their endmarkers.

The total space is thus $9n + l$ bytes plus whatever amount is used for the bucket headers.

In comparison, a trie with nodes implemented by linked lists requires $p + n$ cells, one cell for each prefix and one cell for the end of each word. Each trie cell has a character and two pointers, and needs nine bytes, for a total space of $9n + 9p$. If l plus the space for the bucket headers exceeds $9p$, the trie uses less space. However, for applications such as storing a dictionary where l/p is typically less than 3, the hash table would use less space.

In favor of the trie, however, let us note that we can travel down a trie, and thus perform operations INSERT, DELETE, and MEMBER in time proportional to the length of the word involved. A hash function to be truly "random" should involve each character of the word being hashed. It is fair, therefore, to state that computing the hash function takes roughly as much time as performing an operation like MEMBER on the trie. Of course the time spent computing the hash function does not include the time spent resolving collisions or performing the insertion, deletion, or membership test on the hash table, so we can expect tries to be considerably faster than hash tables for dictionaries whose elements are character strings.

Another advantage to the trie is that it supports the MIN operation efficiently, while hash tables do not. Further, in the hash table organization described above, we cannot easily reuse the space in the character array when a word is deleted (but see Chapter 12 for methods of handling such a problem).

5.4 Balanced Tree Implementations of Sets

In Sections 5.1 and 5.2 we saw how sets could be implemented by binary search trees, and we saw that operations like INSERT could be performed in time proportional to the average depth of nodes in that tree. Further, we discovered that this average depth is $O(\log n)$ for a "random" tree of n nodes. However, some sequences of insertions and deletions can produce binary search trees whose average depth is proportional to n. This suggests that we might try to rearrange the tree after each insertion and deletion so that it is always complete; then the time for INSERT and similar operations would always be $O(\log n)$.

In Fig. 5.12(a) we see a tree of six nodes that becomes the complete tree of seven nodes in Fig. 5.12(b), when element 1 is inserted. Every element in Fig. 5.12(a), however, has a different parent in Fig. 5.12(b), so we must take n steps to insert 1 into a tree like Fig. 5.12(a), if we wish to keep the tree as balanced as possible. It is thus unlikely that simply insisting that the binary search tree be complete will lead to an implementation of a dictionary, priority queue, or other ADT that includes INSERT among its operations, in $O(\log n)$ time.

There are several other approaches that do yield worst case $O(\log n)$ time per operation for dictionaries and priority queues, and we shall consider one of them, called a "2-3 tree," in detail. A *2-3 tree* is a tree with the following two properties.

1. Each interior node has two or three children.
2. Each path from the root to a leaf has the same length.

We shall also consider a tree with zero nodes or one node as special cases of a 2-3 tree.

We represent sets of elements that are ordered by some linear order $<$, as follows. Elements are placed at the leaves; if element a is to the left of
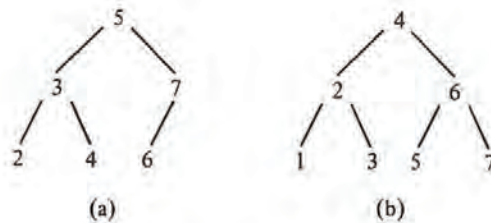
Fig. 5.12. Complete trees.

element b, then $a < b$ must hold. We shall assume that the "$<$" ordering of elements is based on one field of a record that forms the element type; this field is called the *key*. For example, elements might represent people and certain information about people, and in that case, the key field might be "social security number."

At each interior node we record the key of the smallest element that is a descendant of the second child and, if there is a third child, we record the key of the smallest element descending from that child as well.† Figure 5.13 is an example of a 2-3 tree. In that and subsequent examples, we shall identify an element with its key field, so the order of elements becomes obvious.
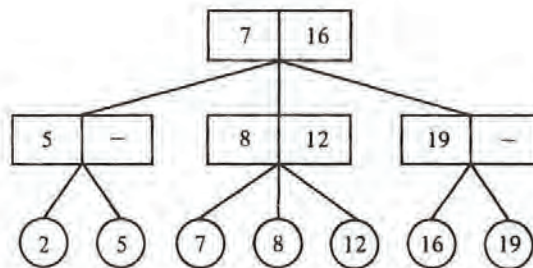
Fig. 5.13. A 2-3 tree.

† There is another version of 2-3 trees that places whole records at interior nodes, as a binary search tree does.

Observe that a 2-3 tree of k levels has between 2^{k-1} and 3^{k-1} leaves. Put another way, a 2-3 tree representing a set of n elements requires at least $1 + \log_3 n$ levels and no more than $1 + \log_2 n$ levels. Thus, path lengths in the tree are $O(\log n)$.

We can test membership of a record with key x in a set represented by a 2-3 tree in $O(\log n)$ time by simply moving down the tree, using the values of the elements recorded at the interior nodes to guide our path. At a node *node*, compare x with the value y that represents the smallest element descending from the second child of *node*. (Recall we are treating elements as if they consisted solely of a key field.) If $x < y$, move to the first child of *node*. If $x \geq y$, and *node* has only two children, move to the second child of *node*. If *node* has three children and $x \geq y$, compare x with z, the second value recorded at *node*, the value that indicates the smallest descendant of the third child of *node*. If $x < z$, go to the second child, and if $x \geq z$, go to the third child. In this manner, we find ourselves at a leaf eventually, and x is in the represented set if and only if x is at the leaf. Evidently, if during this process we find $x = y$ or $x = z$, we can stop immediately. However, we stated the algorithm as we did because in some cases we shall wish to find the leaf with x as well as to verify its existence.

Insertion into a 2-3 Tree

To insert a new element x into a 2-3 tree, we proceed at first as if we were testing membership of x in the set. However, at the level just above the leaves, we shall be at a node *node* whose children, we discover, do not include x. If *node* has only two children, we simply make x the third child of *node*, placing the children in the proper order. We then adjust the two numbers at *node* to reflect the new situation.

For example, if we insert 18 into Fig. 5.13, we wind up with *node* equal to the rightmost node at the middle level. We place 18 among the children of *node*, whose proper order is 16, 18, 19. The two values recorded at *node* become 18 and 19, the elements at the second and third children. The result is shown in Fig. 5.14.

Suppose, however, that x is the fourth, rather than the third child of *node*. We cannot have a node with four children in a 2-3 tree, so we split *node* into two nodes, which we call *node* and *node'*. The two smallest elements among the four children of *node* stay with *node*, while the two larger elements become children of *node'*. Now, we must insert *node'* among the children of p, the parent of *node*. This part of the insertion is analogous to the insertion of a leaf as a child of *node*. That is, if p had two children, we make *node'* the third and place it immediately to the right of *node*. If p had three children before *node'* was created, we split p into p and p', giving p the two leftmost children and p' the remaining two, and then we insert p' among the children of p's parent, recursively.

One special case occurs when we wind up splitting the root. In that case we create a new root, whose two children are the two nodes into which the

55
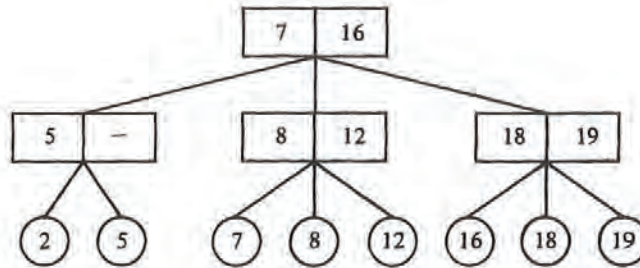
Fig. 5.14. 2-3 tree with 18 inserted.

old root was split. This is how the number of levels in a 2-3 tree increases.

Example 5.4. Suppose we insert 10 into the tree of Fig. 5.14. The intended parent of 10 already has children 7, 8, and 12, so we split it into two nodes. The first of these has children 7 and 8; the second has 10 and 12. The result is shown in Fig. 5.15(a). We must now insert the new node with children 10 and 12 in its proper place as a child of the root of Fig. 5.15(a). Doing so gives the root four children, so we split it, and create a new root, as shown in Fig. 5.15(b). The details of how information regarding smallest elements of subtrees is carried up the tree will be given when we develop the program for the command INSERT. □

Deletion in a 2-3 tree

When we delete a leaf, we may leave its parent *node* with only one child. If *node* is the root, delete *node* and let its lone child be the new root. Otherwise, let *p* be the parent of *node*. If *p* has another child, adjacent to *node* on either the right or the left, and that child of *p* has three children, we can transfer the proper one of those three to *node*. Then *node* has two children, and we are done.

If the children of *p* adjacent to *node* have only two children, transfer the lone child of *node* to an adjacent sibling of *node*, and delete *node*. Should *p* now have only one child, repeat all the above, recursively, with *p* in place of *node*.

Example 5.5. Let us begin with the tree of Fig. 5.15(b). If 10 is deleted, its parent has only one child. But the grandparent has another child that has three children, 16, 18, and 19. This node is to the right of the deficient node, so we pass the deficient node the smallest element, 16, leaving the 2-3 tree in Fig. 5.16(a).
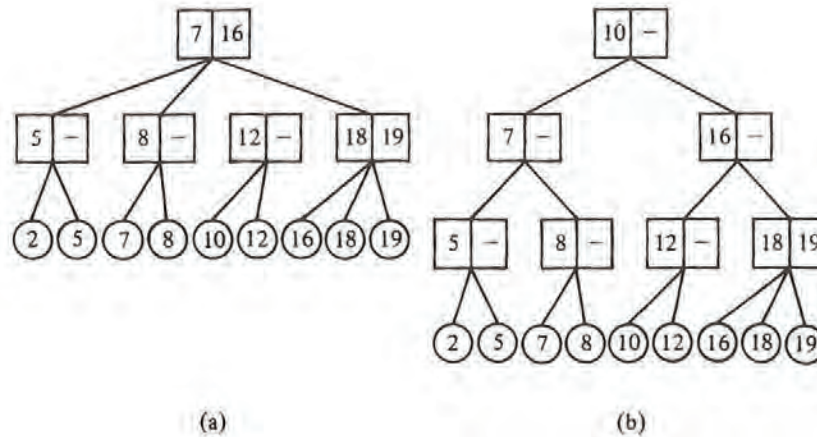
Fig. 5.15. Insertion of 10 into the tree of Fig. 5.14.

Next suppose we delete 7 from the tree of Fig. 5.16(a). Its parent now has only one child, 8, and the grandparent has no child with three children. We therefore make 8 be a sibling of 2 and 5, leaving the tree of Fig. 5.16(b). Now the node starred in Fig. 5.16(b) has only one child, and its parent has no other child with three children. Thus we delete the starred node, making its child be a child of the sibling of the starred node. Now the root has only one child, and we delete it, leaving the tree of Fig. 5.16(c).

Observe in the above examples, the frequent manipulation of the values at interior nodes. While we can always calculate these values by walking the tree, it can be done as we manipulate the tree itself, provided we remember the smallest value among the descendants of each node along the path from the root to the deleted leaf. This information can be computed by a recursive deletion algorithm, with the call at each node being passed, from above, the correct quantity (or the value "minus infinity" if we are along the leftmost path). The details require careful case analysis and will be sketched later when we consider the program for the DELETE operation. □

Data Types for 2-3 Trees

Let us restrict ourselves to representing by 2-3 trees sets of elements whose keys are real numbers. The nature of other fields that go with the field key, to make up a record of type elementtype, we shall leave unspecified, as it has no bearing on what follows.
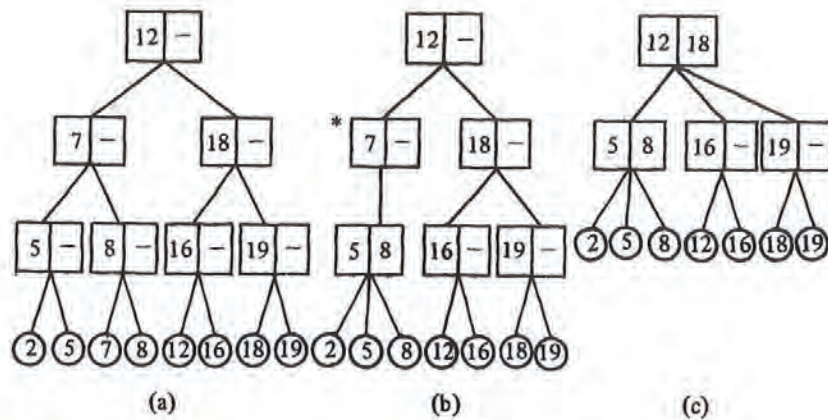
Fig. 5.16. Deletion in a 2-3 tree.

In Pascal, the parents of leaves must be records consisting of two reals (the keys of the smallest elements in the second and third subtrees) and of three pointers to elements. The parents of these nodes are records consisting of two reals and of three pointers to parents of leaves. This progression continues indefinitely; each level in a 2-3 tree is of a different type from all other levels. Such a situation would make programming 2-3 tree operations in Pascal impossible, but fortunately, Pascal provides a mechanism, the variant record structure, that enables us to regard all 2-3 tree nodes as having the same type, even though some are elements, and some are records with pointers and reals.† We can define nodes as in Fig. 5.17. Then we declare a set, represented by a 2-3 tree, to be a pointer to the root as in Fig. 5.17.

Implementation of INSERT

The details of operations on 2-3 trees are quite involved, although the principles are simple. We shall therefore describe only one operation, insertion, in detail; the others, deletion and membership testing, are similar in spirit, and finding the minimum is a trivial search down the leftmost path. We shall write the insertion routine as main procedure, INSERT, which we call at the root, and a procedure *insert*I, which gets called recursively down the tree.

---

† All nodes, however, take the largest amount of space needed for any of the variant types, so Pascal is not really the best language for implementing 2-3 trees in practice.

```
type
    elementtype = record
        key: real;
        {other fields as warranted}
    end;
    nodetypes = (leaf, interior);
    twothreenode = record
        case kind: nodetypes of
            leaf: (element: elementtype);
            interior: (firstchild, secondchild, thirdchild: ↑ twothreenode;
                lowofsecond, lowofthird: real)
    end;
    SET = ↑ twothreenode;
```

Fig. 5.17. Definition of a node in a 2-3 tree.

For convenience, we assume that a 2-3 tree is not a single node or empty.
These two cases require a straightforward sequence of steps which the reader
is invited to provide as an exercise.

```
procedure insert1 ( node: ↑ twothreenode;
    x: elementtype; { x is to be inserted into the subtree of node }
    var pnew: ↑ twothreenode; { pointer to new node created to right of node }
    var low: real ); { smallest element in the subtree pointed to by pnew }

    begin
        pnew := nil;
        if node is a leaf then begin
            if x is not the element at node then begin
                create new node pointed to by pnew;
                put x at the new node;
                low := x.key
            end
        end
        else begin { node is an interior node }
            let w be the child of node to whose subtree x belongs;
            insert1(w, x, pback, lowback);
            if pback <> nil then begin
                insert pointer pback among the children of node just
                    to the right of w;
                if node has four children then begin
                    create new node pointed to by pnew;
                    give the new node the third and fourth children of node;
                    adjust lowofsecond and lowofthird in node and the new node;
```

59

```
                        set low to be the lowest key among the
                            children of the new node
                end
            end
        end
end; { insert 1 }
```

Fig. 5.18. Sketch of 2-3 tree insertion program.

We would like *insert* 1 to return both a pointer to a new node, if it must create one, and the key of the smallest element descended from that new node. As the mechanism in Pascal for creating such a function is awkward, we shall instead declare *insert* 1 to be a procedure that assigns values to parameters *pnew* and *low* in the case that it must "return" a new node. We sketch *insert* 1 in Fig. 5.18. The complete procedure is shown in Fig. 5.19; some comments in Fig. 5.18 have been omitted from Fig. 5.19 to save space.

```
procedure insert 1 ( node : ↑ twothreenode; x: elementtype;
    var pnew : ↑ twothreenode; var low : real );

    var
        pback : ↑ twothreenode;
        lowback : real;
        child : 1..3; { indicates which child of node is followed
            in recursive call (cf. w in Fig. 5.18) }
        w : ↑ twothreenode; { pointer to the child }

    begin
        pnew := nil;
        if node ↑.kind = leaf then begin
            if node ↑.element.key <> x .key then begin
                { create new leaf holding x and "return" this node }
                new (pnew , leaf );
                if (node ↑.element .key < x .key) then
                    { place x in new node to right of current node }
                    begin pnew ↑.element := x; low := x .key end
                else begin  { x belongs to left of element at current node }
                    pnew ↑.element := node ↑.element ;
                    node ↑.element := x;
                    low := pnew ↑.element .key
                end
            end
        end
        else begin  { node is an interior node }
            { select the child of node that we must follow }
            if x.key < node ↑.lowofsecond then
                begin child := 1; w := node ↑.firstchild end
```

```
        else if (node t.thirdchild = nil) or (x.key < node t.lowofthird) then begin
            { x is in second subtree }
            child := 2;
            w := node t.secondchild
    end
    else begin  { x is in third subtree }
        child := 3;
        w := node t.thirdchild
    end;
    insert 1(w, x, pback, lowback);
    if pback <> nil then
        { a new child of node must be inserted }
        if node t.thirdchild = nil then
            { node had only two children, so insert new node in proper place }
            if child = 2 then begin
                node t.thirdchild := pback;
                node t.lowofthird := lowback
            end
            else begin  { child = 1 }
                node t.thirdchild := node t.secondchild;
                node t.lowofthird := node t.lowofsecond;
                node t.secondchild := pback;
                node t.lowofsecond := lowback
            end
        else begin  { node already had three children }
            new (pnew, interior);
            if child = 3 then begin
                { pback and third child become children of new node }
                pnew t.firstchild := node t.thirdchild;
                pnew t.secondchild := pback;
                pnew t.thirdchild := nil;
                pnew t.lowofsecond := lowback;
                    { lowofthird is undefined for pnew }
                low := node t.lowofthird;
                node t.thirdchild := nil
            end
            else begin  { child ≤ 2; move third child of node to pnew}
                pnew t.secondchild := node t.thirdchild;
                pnew t.lowofsecond := node t.lowofthird;
                pnew t.thirdchild := nil;
                node t.thirdchild := nil
            end;
            if child = 2 then begin
                { pback becomes first child of pnew }
                pnew t.firstchild := pback;
```

61

```
                    low := lowback
                end;
                if child = 1 then begin
                    { second child of node is moved to pnew;
                        pback becomes second child of node }
                    pnew↑.firstchild := node↑.secondchild;
                    low := node↑.lowofsecond;
                    node↑.secondchild := pback;
                    node↑.lowofsecond := lowback
                end
            end
    end
end; { insert 1 }
```

Fig. 5.19. The procedure *insert* 1.

Now we can write the procedure INSERT, which calls *insert* 1. If *insert* 1
"returns" a new node, then INSERT must create a new root. The code is
shown in Fig. 5.20 on the assumption that the type SET is ↑ twothreenode,
i.e., a pointer to the root of a 2-3 tree whose leaves contain the members of
the set.

```
procedure INSERT ( x: elementtype; var S: SET );
    var
        pback: ↑ twothreenode; { pointer to new node returned by insert 1 }
        lowback: real; { low value in subtree of pback }
        saveS: SET; { place to store a temporary copy of the pointer S }
    begin
        { checks for S being empty or a single node should occur here,
            and an appropriate insertion procedure should be included }
        insert 1(S, x, pback, lowback);
        if pback <> nil then begin
            { create new root; its children are now pointed to by S and pback }
            saveS := S;
            new(S);
            S↑.firstchild := saveS;
            S↑.secondchild := pback;
            S↑.lowofsecond := lowback;
            S↑.thirdchild := nil;
        end
    end; { INSERT }
```

Fig. 5.20. INSERT for sets represented by 2-3 trees.

Implementation of DELETE

We shall sketch a function *delete* 1 that takes a pointer to a node *node* and an element *x*, and deletes a leaf descended from *node* having value *x*, if there is one.† Function *delete* 1 returns true if after deletion *node* has only one child, and it returns false if *node* still has two or three children. A sketch of the code for *delete* 1 is shown in Fig. 5.21.

We leave the detailed code for function *delete* 1 for the reader. Another exercise is to write a procedure DELETE(S, x) that checks for the special cases that the set S consists only of a single leaf or is empty, and otherwise calls *delete* 1(S, x); if *delete* 1 returns true, the procedure removes the root (the node pointed to by S) and makes S point to its lone child.

```
function delete 1 ( node : ↑ twothreenode; x : elementtype ) : boolean;
    var
        onlyone : boolean;  { to hold the value returned by a call to delete 1 }
    begin
        delete 1 := false;
        if the children of node are leaves then begin
            if x is among those leaves then begin
                remove x;
                shift children of node to the right of x one position left;
                if node now has one child then
                    delete 1 := true
            end
        end
        else begin  { node is at level two or higher }
            determine which child of node could have x as a descendant;
            onlyone := delete 1(w, x);  { w stands for node ↑.firstchild,
                node ↑.secondchild, or node ↑.thirdchild, as appropriate }
            if onlyone then begin  { fix children of node }
                if w is the first child of node then
                    if y, the second child of node, has three children then
                        make the first child of y be the second child of w
                    else begin  { y has two children }
                        make the child of w be the first child of y;
                        remove w from among the children of node;
                        if node now has one child then
                            delete 1 := true
                    end;
                if w is the second child of node then
                    if y, the first child of node, has three children then
                        make the third child of y be the first child of w
```

---

† A useful variant would take only a key value and delete any element with that key.

```
else { y has two children }
    if z, the third child of node, exists
        and has three children then
            make first child of z be the second child of w
    else begin { no other child of node has three children }
        make the child of w be the third child of y;
        remove w from among the children of node;
        if node now has one child then
            delete 1 := true
    end;
if w is the third child of node then
    if y, the second child of node, has three children then
        make the third child of y be the first child of w
    else begin { y has two children }
        make the child of w be the third child of y;
        remove w from among the children of node
    end { note node surely has two children left in this case }

    end

end
end; { delete 1 }
```

Fig. 5.21. Recursive deletion procedure.

5.5 Sets with the MERGE and FIND Operations

In certain problems we start with a collection of objects, each in a set by itself; we then combine sets in some order, and from time to time ask which set a particular object is in. These problems can be solved using the operations MERGE and FIND. The operation MERGE(A, B, C) makes C equal to the union of sets A and B, provided A and B are disjoint (have no member in common); MERGE is undefined if A and B are not disjoint. FIND(x) is a function that returns the set of which x is a member; in case x is in two or more sets, or in no set, FIND is not defined.

Example 5.6. An *equivalence relation* is a reflexive, symmetric, and transitive relation. That is, if $\equiv$ is an equivalence relation on set S, then for any (not necessarily distinct) members a, b, and c in S, the following properties hold:

1. $a \equiv a$ *(reflexivity)*
2. If $a \equiv b$, then $b \equiv a$ *(symmetry)*.
3. If $a \equiv b$ and $b \equiv c$, then $a \equiv c$ *(transitivity)*.

   The relation "is equal to" ($=$) is the paradigm equivalence relation on any set S. For a, b, and c in S, we have (1) $a = a$, (2) if $a = b$, then $b = a$, and (3) if $a = b$ and $b = c$, then $a = c$. There are many other equivalence relations, however, and we shall shortly see several additional examples.

64

In general, whenever we partition a collection of objects into disjoint groups, the relation $a \equiv b$ if and only if a and b are in the same group is an equivalence relation. "Is equal to" is the special case where every element is in a group by itself.

More formally, if a set S has an equivalence relation defined on it, then the set S can be partitioned into disjoint subsets $S_1, S_2, \ldots$, called *equivalence classes*, whose union is S. Each subset S_i consists of equivalent members of S. That is, $a \equiv b$ for all a and b in S_i, and $a \not\equiv b$ if a and b are in different subsets. For example, the relation congruence modulo n† is an equivalence relation on the set of integers. To check that this is so, note that $a - a = 0$, which is a multiple of n (reflexivity); if $a - b = dn$, then $b - a = (-d)n$ (symmetry); and if $a - b = dn$ and $b - c = en$, then $a - c = (d + e)n$ (transitivity). In the case of congruence modulo n there are n equivalence classes, which are the set of integers congruent to 0, the set of integers congruent to 1, $\ldots$, the set of integers congruent to $n - 1$.

The *equivalence problem* can be formulated in the following manner. We are given a set S and a sequence of statements of the form "a is equivalent to b." We are to process the statements in order in such a way that at any time we are able to determine in which equivalence class a given element belongs. For example, suppose $S = \{1, 2, \ldots, 7\}$ and we are given the sequence of statements

$$1 \equiv 2 \quad 5 \equiv 6 \quad 3 \equiv 4 \quad 1 \equiv 4$$

to process. The following sequence of equivalence classes needs to be constructed, assuming that initially each element of S is in an equivalence class by itself.

$1 \equiv 2$	$\{1,2\}$ $\{3\}$ $\{4\}$ $\{5\}$ $\{6\}$ $\{7\}$
$5 \equiv 6$	$\{1,2\}$ $\{3\}$ $\{4\}$ $\{5,6\}$ $\{7\}$
$3 \equiv 4$	$\{1,2\}$ $\{3,4\}$ $\{5,6\}$ $\{7\}$
$1 \equiv 4$	$\{1,2,3,4\}$ $\{5,6\}$ $\{7\}$

We can "solve" the equivalence problem by starting with each element in a named set. When we process statement $a \equiv b$, we FIND the equivalence classes of a and b and then MERGE them. We can at any time use FIND to tell us the current equivalence class of any element.

The equivalence problem arises in several areas of computer science. For example, one form occurs when a Fortran compiler has to process "equivalence declarations" such as

---

† We say a is *congruent to b modulo n* if a and b have the same remainders when divided by n, or put another way, $a - b$ is a multiple of n.

EQUIVALENCE (A(1),B(1,2),C(3)), (A(2),D,E), (F,G)

Another example, presented in Chapter 7, uses solutions to the equivalence problem to help find minimum-cost spanning trees. □

A Simple Implementation of MFSET

Let us begin with a simplified version of the MERGE-FIND ADT. We shall define an ADT, called MFSET, consisting of a set of subsets, which we shall call *components*, together with the following operations:

1. MERGE(A, B) takes the union of the components A and B and calls the result either A or B, arbitrarily.
2. FIND(x) is a function that returns the name of the component of which x is a member.
3. INITIAL(A, x) creates a component named A that contains only the element x.

To make a reasonable implementation of MFSET, we must restrict our underlying types, or alternatively, we should recognize that MFSET really has two other types as "parameters" — the type of set names and the type of members of these sets. In many applications we can use integers as set names. If we take n to be the number of elements, we may also use integers in the range $[1..n]$ for the members of components. For the implementation we have in mind, it is important that the type of set members be a subrange type, because we want to index into an array defined over that subrange. The type of set names is not important, as this type is the type of array elements, not their indices. Observe, however, that if we wanted the member type to be other than a subrange type, we could create a mapping, with a hash table, for example, that assigned these to unique integers in a subrange. We only need to know the total number of elements in advance.

The implementation we have in mind is to declare

```
const
    n = { number of elements };
type
    MFSET = array[1..n] of integer;
```

as a special case of the more general type

```
        array[subrange of members] of (type of set names);
```

Suppose we declare *components* to be of type MFSET with the intention that *components*[x] holds the name of the set currently containing x. Then the three MFSET operations are easy to write. For example, the operation MERGE is shown in Fig. 5.22. INITIAL(A, x) simply sets *components*[x] to A, and FIND(x) returns *components*[x].

The time performance of this implementation of MFSET is easy to

```
procedure MERGE ( A, B: integer; var C: MFSET );
    var
        x: 1..n;
    begin
        for x := 1 to n do
            if C [x] = B then
                C [x] := A
    end;  { MERGE }
```

Fig. 5.22. The procedure MERGE.

analyze. Each execution of the procedure MERGE takes $O(n)$ time. On the other hand, the obvious implementations of INITIAL(A, x) and FIND(x) have constant running times.

A Faster Implementation of MFSET

Using the algorithm in Fig. 5.22, a sequence of $n-1$ MERGE instructions will take $O(n^2)$ time.† One way to speed up the MERGE operation is to link together all members of a component in a list. Then, instead of scanning all members when we merge component B into A, we need only run down the list of members of B. This arrangement saves time on the average. However, it could happen that the ith merge is of the form MERGE(A, B) where A is a component of size 1 and B is a component of size i, and that the result is named A. This merge operation would require $O(i)$ steps, and a sequence of $n-1$ such merge instructions would take on the order of $\sum_{i=1}^{n-1} i = n(n-1)/2$ time.

One way to avoid this worst case situation is to keep track of the size of each component and always merge the smaller into the larger.‡ Thus, every time a member is merged into a bigger component, it finds itself in a component at least twice as big. Thus, if there are initially n components, each with one member, none of the n members can have its component changed more than $1 + \log n$ times. As the time spent by this new version of MERGE is proportional to the number of members whose component names are changed, and the total number of such changes is at most $n(1 + \log n)$, we see that $O(n \log n)$ work suffices for all merges.

Now let us consider the data structure needed for this implementation. First, we need a mapping from set names to records consisting of

---

† Note that $n-1$ is the largest number of merges that can be performed before all elements are in one set.

‡ Note that our ability to call the resulting component by the name of either of its constituents is important here, although in the simpler implementation, the name of the first argument was always picked.

1. a *count* giving the number of members in the set and

2. the index in the array of the first element of that set.

We also need another array of records, indexed by members, to indicate

1. the set of which each element is a member and

2. the next array element on the list for that set.

We use 0 to serve as NIL, the end-of-list marker. In a language that lent itself to such constructs, we would prefer to use pointers in this array, but Pascal does not permit pointers into arrays.

In the special case where set names, as well as members, are chosen from the subrange $1..n$, we can use an array for the mapping described above. That is, we define

```
type
    nametype = 1..n;
    elementtype = 1..n;
    MFSET = record
        setheaders: array[1..n] of record
            { headers for set lists }
            count: 0..n;
            firstelement: 0..n
        end;
        names: array[1..n] of record
            { table giving set containing each member }
            setname: nametype;
            nextelement: 0..n
        end
    end;
```

The procedures INITIAL, MERGE, and FIND are shown in Fig. 5.23.

Figure 5.24 shows an example of the data structure used in Fig. 5.23, where set 1 is {1, 3, 4}, set 2 is {2}, and set 5 is {5, 6}.

A Tree Implementation of MFSET's

Another, completely different, approach to the implementation of MFSET's uses trees with pointers to parents. We shall describe this approach informally. The basic idea is that nodes of trees correspond to set members, with an array or other implementation of a mapping leading from set members to their nodes. Each node, except the root of each tree, has a pointer to its parent. The roots hold the name of the set, as well as an element. A mapping from set names to roots allows access to any given set, when merges are done.

Figure 5.25 shows the sets $A = \{1, 2, 3, 4\}$, $B = \{5, 6\}$, and $C = \{7\}$ represented in this form. The rectangles are assumed to be part of the root node, not separate nodes.

68

```
procedure INITIAL ( A: nametype; x: elementtype; var C: MFSET );
    { initialize A to a set containing x only }
    begin
        C.names [x].setname := A;
        C.names [x].nextelement := 0;
            { null pointer at end of list of members of A }
        C.setheaders [A].count := 1;
        C.setheaders [A].firstelement := x
    end;  { INITIAL }

procedure  MERGE ( A, B: nametype; var C: MFSET );
    { merge A and B, calling the result A or B, arbitrarily }
    var
        i: 0..n;  { used to find end of smaller list }
    begin
        if C.setheaders [A].count > C.setheaders [B].count then begin
            { A is the larger set; merge B into A }
            { find end of B, changing set names to A as we go }
            i := C.setheaders[B].firstelement;
            while C.names [i].nextelement <> 0 do begin
                C.names [i].setname := A;
                i := C.names [i].nextelement
            end;
            { append list A to the end of B and call the result A }
            { now i is the index of the last member of B }
            C.names [i].setname := A;
            C.names [i].nextelement := C.setheaders [A].firstelement;
            C.setheaders [A].firstelement := C.setheaders [B].firstelement;
            C.setheaders [A].count := C.setheaders [A].count +
                        C.setheaders [B].count;
            C.setheaders [B].count := 0;
            C.setheaders [B].firstelement := 0
                { above two steps not really necessary, as set B no longer exists }
        end
        else { B is at least as large as A }
            { code similar to case above, but with A and B interchanged }
    end;  { MERGE }

function FIND ( x: 1..n; var C: MFSET );
    { return the name of the set of which x is a member }
    begin
        return (C.names [x].setname)
    end;  { FIND }
```

Fig. 5.23. The operations of an MFSET.

69

To find the set containing an element x, we first consult a mapping (e.g., an array) not shown in Fig. 5.25, to obtain a pointer to the node for x. We then follow the path from that node to the root of its tree and read the name of the set there.

The basic merge operation is to make the root of one tree be a child of the root of the other. For example, we could merge A and B of Fig. 5.25 and call the result A, by making node 5 a child of node 1. The result is shown in Fig. 5.26. However, indiscriminate merging could result in a tree of n nodes that is a single chain. Then doing a FIND operation on each of those nodes would take $O(n^2)$ time. Observe that although a merge can be done in $O(1)$ steps, the cost of a reasonable number of FIND's will dominate the total cost, and this approach is not necessarily better than the simplest one for executing n merges and n finds.

However, a simple improvement guarantees that if n is the number of elements, then no FIND will take more than $O(\log n)$ steps. We simply keep at each root a count of the number of elements in the set, and when called upon to merge two sets, we make the root of the smaller tree be a child of the root of the larger. Thus, every time a node is moved to a new tree, two things happen: the distance from the node to its root increases by one, and the node will be in a set with at least twice as many elements as before. Thus, if n is the total number of elements, no node can be moved more than $\log n$ times; hence, the distance to its root can never exceed $\log n$. We conclude that each FIND requires at most $O(\log n)$ time.

Path Compression

Another idea that may speed up this implementation of MFSET's is *path compression*. During a FIND, when following a path from some node to the root, make each node encountered along the path be a child of the root. The easiest way to do this is in two passes. First, find the root, and then retraverse the same path, making each node a child of the root.

Example 5.7. Figure 5.27(a) shows a tree before executing a FIND operation on the node for element 7 and Fig. 5.27(b) shows the result after 5 and 7 are made children of the root. Nodes 1 and 2 on the path are not moved because 1 is the root, and 2 is already a child of the root. □

Path compression does not affect the cost of MERGE's; each MERGE still takes a constant amount of time. There is, however, a subtle speedup in FIND's since path compression tends to shorten a large number of paths from various nodes to the root with relatively little effort.

Unfortunately, it is very difficult to analyze the average cost of FIND's when path compression is used. It turns out that if we do not require that smaller trees be merged into larger ones, we require no more than $O(n \log n)$ time to do n FIND's. Of course, the first FIND may take $O(n)$ time by itself for a tree consisting of one chain. But path compression can change a tree very rapidly and no matter in what order we apply FIND to elements of any tree no more than $O(n)$ time is spent on n FIND's. However, there are
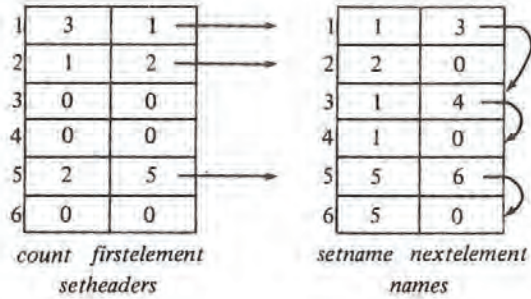
1	3	1		1	1	3
2	1	2		2	2	0
3	0	0		3	1	4
4	0	0		4	1	0
5	2	5		5	5	6
6	0	0		6	5	0

count firstelement   setname nextelement
setheaders   names

Fig. 5.24. Example of the MFSET data structure.
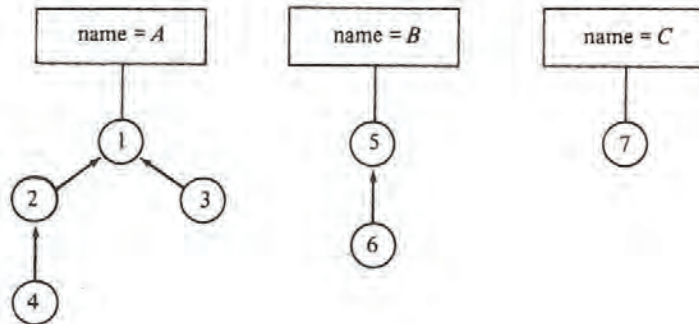
name = A   name = B   name = C

Fig. 5.25. MFSET represented by a collection of trees.

sequences of MERGE and FIND instructions that require $\Omega(n \log n)$ time.

The algorithm that both uses path compression and merges the smaller tree into the larger is asymptotically the most efficient method known for implementing MFSET's. In particular, n FIND's require no more than $O(n\alpha(n))$ time, where $\alpha(n)$ is a function that is not constant, yet grows much more slowly than $\log n$. We shall define $\alpha(n)$ below, but the analysis that leads to this bound is beyond the scope of this book.
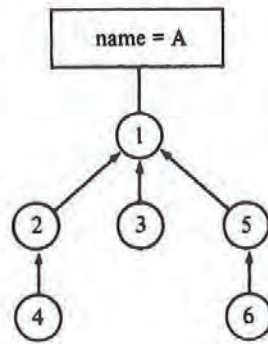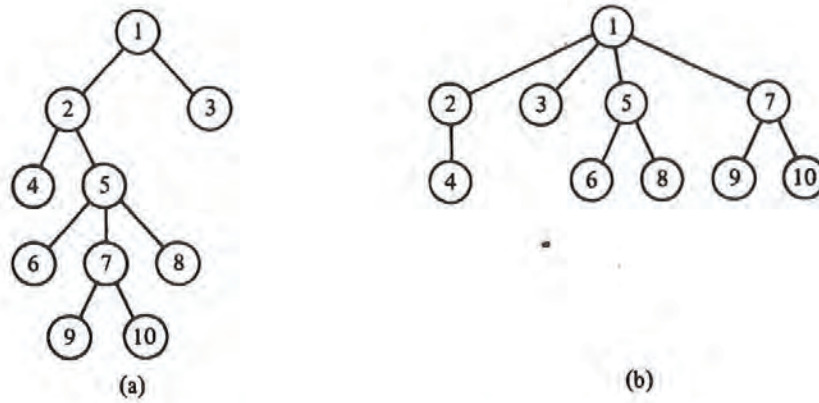
71

Fig. 5.26. Merging *B* into *A*.

(a)   (b)

Fig. 5.27. An example of path compression.

The Function $\alpha(n)$

The function $\alpha(n)$ is closely related to a very rapidly growing function $A(x, y)$, known as *Ackermann's function*. $A(x, y)$ is defined recursively by:

$$A(0, y) = 1 \text{ for } y \geq 0$$

$$A(1, 0) = 2$$

$$A(x, 0) = x+2 \text{ for } x \geq 2$$

$$A(x, y) = A(A(x-1, y), y-1) \text{ for } x, y \geq 1$$

Each value of y defines a function of one variable. For example, the third line above tells us that for $y=0$, this function is "add 2." For $y = 1$, we have $A(x, 1) = A(A(x-1, 1), 0) = A(x-1, 1) + 2$, for $x > 1$, with $A(1, 1) = A(A(0, 1), 0) = A(1, 0) = 2$. Thus $A(x, 1) = 2x$ for all $x \geq 1$. In other words, $A(x, 1)$ is "multiply by 2." Then, $A(x, 2) = A(A(x-1, 2), 1) = 2A(x-1, 2)$ for $x > 1$. Also, $A(1, 2) = A(A(0, 2), 1) = A(1, 1) = 2$. Thus $A(x, 2) = 2^x$. Similarly, we can show that $A(x, 3) = 2^{2^{\cdot^{\cdot^{\cdot^2}}}}$ (stack of x 2's), while $A(x, 4)$ is so rapidly growing there is no accepted mathematical notation for such a function.

A single-variable Ackermann's function can be defined by letting $A(x) = A(x, x)$. The function $\alpha(n)$ is a pseudo-inverse of this single variable function. That is, $\alpha(n)$ is the least x such that $n \leq A(x)$. For example, $A(1) = 2$, so $\alpha(1) = \alpha(2) = 1$. $A(2) = 4$, so $\alpha(3) = \alpha(4) = 2$. $A(3) = 8$, so $\alpha(5) = \cdots = \alpha(8) = 3$. So far, $\alpha(n)$ seems to be growing rather steadily.

However, $A(4)$ is a stack of 65536 2's. Since $\log(A(4))$ is a stack of 65535 2's, we cannot hope even to write $A(4)$ explicitly, as it would take $\log(A(4))$ bits to do so. Thus $\alpha(n) \leq 4$ for all integers n one is ever likely to encounter. Nevertheless, $\alpha(n)$ eventually reaches 5, 6, 7, . . . on its unimaginably slow course toward infinity.

5.6 An ADT with MERGE and SPLIT

Let S be a set whose members are ordered by the relation $<$. The operation SPLIT(S, S_1, S_2, x) partitions S into two sets: $S_1 = \{a \mid a \text{ is in } S \text{ and } a < x\}$ and $S_2 = \{a \mid a \text{ is in } S \text{ and } a \geq x\}$. The value of S after the split is undefined, unless it is one of S_1 or S_2. There are several situations where the operation of splitting sets by comparing each member with a fixed value x is essential. We shall consider one such problem here.

The Longest Common Subsequence Problem

A *subsequence* of a sequence x is obtained by removing zero or more (not necessarily contiguous) elements from x. Given two sequences x and y, a *longest common subsequence (LCS)* is a longest sequence that is a subsequence of both x and y.

73

For example, an LCS of 1, 2, 3, 2, 4, 1, 2 and 2, 4, 3, 1, 2, 1 is the subsequence 2, 3, 2, 1, formed as shown in Fig. 5.28. There are other LCS's as well, such as 2, 4, 1, 2, but there are no common subsequences of length 5.
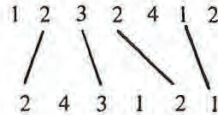

Fig. 5.28. A longest common subsequence.

There is a UNIX command called *diff* that compares files line-by-line, finding a longest common subsequence, where a line of a file is considered an element of the subsequence. That is, whole lines are analogous to the integers 1, 2, 3, and 4 in Fig. 5.28. The assumption behind the command *diff* is that the lines of each file that are not in this LCS are lines inserted, deleted or modified in going from one file to the other. For example, if the two files are versions of the same program made several days apart, *diff* will, with high probability, find the changes.

There are several general solutions to the LCS problem that work in $O(n^2)$ steps on sequences of length n. The command *diff* uses a different strategy that works well when the files do not have too many repetitions of any line. For example, programs will tend to have lines "begin" and "end" repeated many times, but other lines are not likely to repeat.

The algorithm used by *diff* for finding an LCS makes use of an efficient implementation of sets with operations MERGE and SPLIT, to work in time $O(p\log n)$, where n is the maximum number of lines in a file and p is the number of pairs of positions, one from each file, that have the same line. For example, p for the strings in Fig. 5.28 is 12. The two 1's in each string contribute four pairs, the 2's contribute six pairs, and 3 and 4 contribute one pair each. In the worst case, p could be n^2, and this algorithm would take $O(n^2\log n)$ time. However, in practice, p is usually closer to n, so we can expect an $O(n\log n)$ time complexity.

To begin the description of the algorithm let $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$ be the two strings whose LCS we desire. The first step is to tabulate for each value a, the positions of the string A at which a appears. That is, we define PLACES$(a) = \{ i \mid a = a_i \}$. We can compute the sets PLACES(a) by constructing a mapping from symbols to headers of lists of positions. By using a hash table, we can create the sets PLACES(a) in $O(n)$ "steps" on the average, where a "step" is the time it takes to operate on a symbol, say to hash it or compare it with another. This time could be a constant if symbols are characters or integers, say. However, if the symbols of A

and B are really lines of text, then steps take an amount of time that depends on the average length of a line of text.

Having computed PLACES(a) for each symbol a that occurs in string A, we are ready to find an LCS. To simplify matters, we shall only show how to find the length of the LCS, leaving the actual construction of the LCS as an exercise. The algorithm considers each b_j, for $j = 1, 2, \ldots, m$, in turn. After considering b_j, we need to know, for each i between 0 and n, the length of the LCS of strings $a_1 \cdots a_i$ and $b_1 \cdots b_j$.

We shall group values of i into sets S_k, for $k = 0, 1, \ldots, n$, where S_k consists of all those integers i such that the LCS of $a_1 \cdots a_i$ and $b_1 \cdots b_j$ has length k. Note that S_k will always be a set of consecutive integers, and the integers in S_{k+1} are larger than those in S_k, for all k.

Example 5.8. Consider Fig. 5.28, with $j = 5$. If we try to match zero symbols from the first string with the first five symbols of the second (24312), we naturally have an LCS of length 0, so 0 is in S_0. If we use the first symbol from the first string, we can obtain an LCS of length 1, and if we use the first two symbols, 12, we can obtain an LCS of length 2. However, using 123, the first three symbols, still gives us an LCS of length 2 when matched against 24312. Proceeding in this manner, we discover $S_0 = \{0\}$, $S_1 = \{1\}$, $S_2 = \{2, 3\}$, $S_3 = \{4, 5, 6\}$, and $S_4 = \{7\}$. □

Suppose that we have computed the S_k's for position $j-1$ of the second string and we wish to modify them to apply to position j. We consider the set PLACES(b_j). For each r in PLACES(b_j), we consider whether we can improve some of the LCS's by adding the match between a_r and b_j to the LCS of $a_1 \cdots a_{r-1}$ and $b_1 \cdots b_j$. That is, if both $r-1$ and r are in S_k, then all $s \geq r$ in S_k really belong in S_{k+1} when b_j is considered. To see this we observe that we can obtain k matches between $a_1 \cdots a_{r-1}$ and $b_1 \cdots b_{j-1}$, to which we add a match between a_r and b_j. We can modify S_k and S_{k+1} by the following steps.

1. FIND(r) to get S_k.

2. If FIND($r-1$) is not S_k, then no benefit can be had by matching b_j with a_r. Skip the remaining steps and do not modify S_k or S_{k+1}.

3. If FIND($r-1$) = S_k, apply SPLIT(S_k, S_k, S_k', r) to separate from S_k those members greater than or equal to r.

4. MERGE(S_k', S_{k+1}, S_{k+1}) to move these elements into S_{k+1}.

It is important to consider the members of PLACES(b_j) largest first. To see why, suppose for example that 7 and 9 are in PLACES(b_j), and before b_j is considered, $S_3 = \{6, 7, 8, 9\}$ and $S_4 = \{10, 11\}$.

If we consider 7 before 9, we split S_3 into $S_3 = \{6\}$ and $S_3' = \{7, 8, 9\}$, then make $S_4 = \{7, 8, 9, 10, 11\}$. If we then consider 9, we split S_4 into $S_4 = \{7, 8\}$ and $S_4' = \{9, 10, 11\}$, then merge 9, 10 and 11 into S_5. We have thus moved 9 from S_3 to S_5 by considering only one more position in the second string, representing an impossibility. Intuitively, what has happened is that we have erroneously matched b_j against both a_7 and a_9 in creating an

imaginary LCS of length 5.

In Fig. 5.29, we see a sketch of the algorithm that maintains the sets S_k as we scan the second string. To determine the length of an LCS, we need only execute FIND(n) at the end.

```
        procedure LCS;
          begin
(1)           initialize S_0 = {0, 1, . . . ,n} and S_i = Ø for i = 1, 2, . . . ,n;
(2)           for j := 1 to n do  { compute S_k's for position j }
(3)               for r in PLACES(b_j), largest first do begin
(4)                   k := FIND(r);
(5)                   if k = FIND(r−1) then begin  { r is not smallest in S_k }
(6)                       SPLIT(S_k, S_k, S_k', r);
(7)                       MERGE(S_k', S_{k+1}, S_{k+1})
                      end
                  end
          end;  { LCS }
```

Fig. 5.29. Sketch of longest common subsequence program.

Time Analysis of the LCS Algorithm

As we mentioned earlier, the algorithm of Fig. 5.29 is a useful approach only if there are not too many matches between symbols of the two strings. The measure of the number of matches is

$$p = \sum_{j=1}^{m} |\, PLACES(b_j)|$$

where $|PLACES(b_j)|$ denotes the number of elements in set PLACES(b_j). In other words, p is the sum over all b_j of the number of positions in the first string that match b_j. Recall that in our discussion of file comparison, we expect p to be of the same order as m and n, the lengths of the two strings (files).

It turns out that the 2-3 tree is a good structure for the sets S_k. We can initialize these sets, as in line (1) of Fig. 5.29, in $O(n)$ steps. The FIND operation requires an array to serve as a mapping from positions r to the leaf for r and also requires pointers to parents in the 2-3 tree. The name of the set, i.e., k for S_k, can be kept at the root, so we can execute FIND in $O(\log n)$ steps by following parent pointers until we reach the root. Thus all executions of lines (4) and (5) together take $O(p \log n)$ time, since those lines are each executed exactly once for each match found.

The MERGE operation of line (5) has the special property that every member of S_k' is lower than every member of S_{k+1}, and we can take advantage of this fact when using 2-3 trees for an implementation.† To begin the

---

† Strictly speaking we should use a different name for the MERGE operation, as the implementation we propose will not work to compute the arbitrary union of disjoint sets,

MERGE, place the 2-3 tree for S_k' to the left of that for S_{k+1}. If both are of the same height, create a new root with the roots of the two trees as children. If S_k' is shorter, insert the root of that tree as the leftmost child of the leftmost node of S_{k+1} at the appropriate level. If this node now has four children, we modify the tree exactly as in the INSERT procedure of Fig. 5.20. An example is shown in Fig. 5.30. Similarly, if S_{k+1} is shorter, make its root the rightmost child of the rightmost node of S_k' at the appropriate level.


(a) Before restructuring   (b) After restructuring
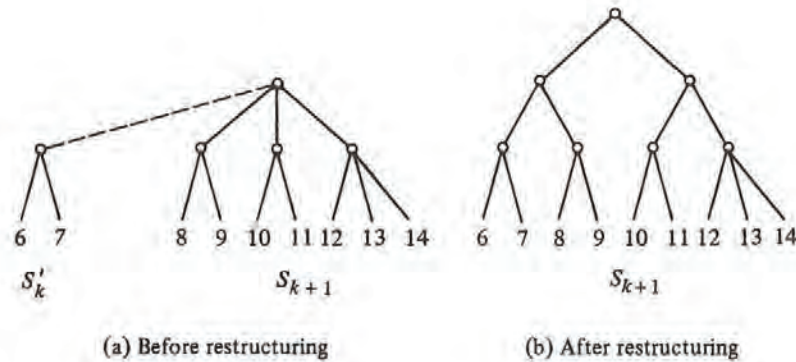
Fig. 5.30. Example of MERGE.

The SPLIT operation at r requires that we travel up the tree from leaf r, duplicating every interior node along the path and giving one copy to each of the two resulting trees. Nodes with no children are eliminated, and nodes with one child are removed and have that child inserted into the proper tree at the proper level.

Example 5.9. Suppose we split the tree of Fig. 5.30(b) at node 9. The two trees, with duplicated nodes, are shown in Fig. 5.31(a). On the left, the parent of 8 has only one child, so 8 becomes a child of the parent of 6 and 7. This parent now has three children, so all is as it should be; if it had four children, a new node would have been created and inserted into the tree. We need only eliminate nodes with zero children (the old parent of 8) and the chain of nodes with one child leading to the root. The parent of 6, 7, and 8 becomes the new root, as shown in Fig. 5.31(b). Similarly, in the right-hand tree, 9 becomes a sibling of 10 and 11, and unnecessary nodes are eliminated, as is also shown in Fig. 5.31(b). □

---

while keeping the elements sorted so operations like SPLIT and FIND can be performed.
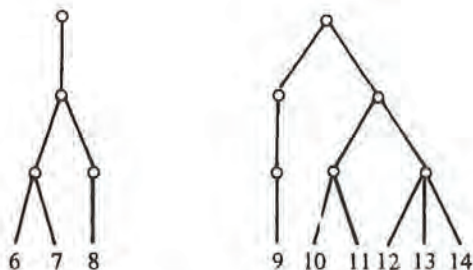
(a) Split trees.
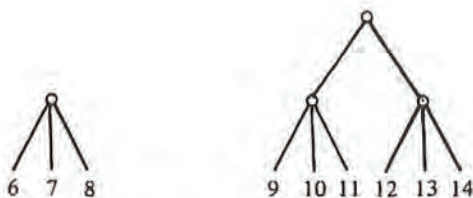
(b) Result of repairs.

Fig. 5.31. An example of SPLIT.

If we do the splitting and reorganization of the 2-3 tree bottom up, it can be shown by consideration of a large number of cases that $O(\log n)$ steps suffices. Thus, the total time spent in lines (6) and (7) of Fig. 5.29 is $O(p\log n)$, and hence the entire algorithm takes $O(p\log n)$ steps. We must add in the preprocessing time needed to compute and sort PLACES(a) for symbols a. As we mentioned, if the symbols a are "large" objects, this time can be much greater than any other part of the algorithm. As we shall see in Chapter 8, if the symbols can be manipulated and compared in single "steps," then $O(n\log n)$ time suffices to sort the first string $a_1 a_2 \cdots a_n$ (actually, to sort objects (i, a_i) on the second field), whereupon PLACES(a) can be read off from this list in $O(n)$ time. Thus, the length of the LCS can be computed in $O(\max(n, p) \log n)$ time which, since $p \geq n$ is normal, can be taken as $O(p\log n)$.

Exercises

5.1 Draw all possible binary search trees containing the four elements 1, 2, 3, 4.

5.2 Insert the integers 7, 2, 9, 0, 5, 6, 8, 1 into a binary search tree by repeated application of the procedure INSERT of Fig. 5.3.

5.3 Show the result of deleting 7, then 2 from the final tree of Exercise 5.2.

*5.4 When deleting two elements from a binary search tree using the procedure of Fig. 5.5, does the final tree ever depend on the order in which you delete them?

5.5 We with to keep track of all 5-character substrings that occur in a given string, using a trie. Show the trie that results when we insert the 14 substrings of length five of the string ABCDABACDEBA-CADEBA.

*5.6 To implement Exercise 5.5, we could keep a pointer at each leaf, which, say, represents string *abcde*, to the interior node representing the suffix *bcde*. That way, if the next symbol, say *f*, is received, we don't have to insert all of *bcdef*, starting at the root. Furthermore, having seen *abcde*, we may as well create nodes for *bcde*, *cde*, *de*, and *e*, since we shall, unless the sequence ends abruptly, need those nodes eventually. Modify the trie data structure to maintain such pointers, and modify the trie insertion algorithm to take advantage of this data structure.

5.7 Show the 2-3 tree that results if we insert into an empty set, represented as a 2-3 tree, the elements 5, 2, 7, 0, 3, 4, 6, 1, 8, 9.

5.8 Show the result of deleting 3 from the 2-3 tree that results from Exercise 5.7.

5.9 Show the successive values of the various S_i's when implementing the LCS algorithm of Fig. 5.29 with first string *abacabada*, and second string *bdbacbad*.

5.10 Suppose we use 2-3 trees to implement the MERGE and SPLIT operations as in Section 5.6.

   a) Show the result of splitting the tree of Exercise 5.7 at 6.

   b) Merge the tree of Exercise 5.7 with the tree consisting of leaves for elements 10 and 11.

5.11 Some of the structures discussed in this chapter can be modified easily to support the MAPPING ADT. Write procedures MAKENULL, ASSIGN, and COMPUTE to operate on the following data structures.

   a) Binary search trees. The "<" ordering applies to domain elements.

            b) 2-3 trees. At interior nodes, place only the key field of domain elements.

5.12 Show that in any subtree of a binary search tree, the minimum element is at a node without a left child.

5.13 Use Exercise 5.12 to produce a nonrecursive version of DELETE-MIN.

5.14 Write procedures ASSIGN, VALUEOF, MAKENULL and GETNEW for trie nodes represented as lists of cells.

*5.15 How do the trie (list of cells implementation), the open hash table, and the binary search tree compare for speed and for space utilization when elements are strings of up to ten characters?

*5.16 If elements of a set are ordered by a "$<$" relation, then we can keep one or two elements (not just their keys) at interior nodes of a 2-3 tree, and we then do not have to keep these elements at the leaves. Write INSERT and DELETE procedures for 2-3 trees of this type.

5.17 Another modification we could make to 2-3 trees is to keep only keys at interior nodes, but do not require that the keys k_1 and k_2 at a node truly be the minimum keys of the second and third subtrees, just that all keys k of the third subtree satisfy $k \geq k_2$, all keys k of the second satisfy $k_1 \leq k < k_2$, and all keys k of the first satisfy $k < k_1$.

     a) How does this convention simplify the DELETE operation?

     b) Which of the dictionary and mapping operations are made more complicated or less efficient?

*5.18 Another data structure that supports dictionaries with the MIN operation is the *AVL tree* (named for the inventors' initials) or *height−balanced tree*. These trees are binary search trees in which the heights of two siblings are not permitted to differ by more than one. Write procedures to implement INSERT and DELETE, while maintaining the AVL-tree property.

5.19 Write the Pascal program for procedure *delete*1 of Fig. 5.21.

*5.20 A *finite automaton* consists of a set of states, which we shall take to be the integers $1..n$ and a table *transitions*[*state*, *input*] giving a *next state* for each *state* and each *input* character. For our purposes, we shall assume that the input is always either 0 or 1. Further, certain of the states are designated *accepting states*. For our purposes, we shall assume that all and only the even numbered states are accepting. Two states p and q are *equivalent* if either they are the same state, or (i) they are both accepting or both nonaccepting, (ii) on input 0 they transfer to equivalent states, and (iii) on input 1 they transfer to equivalent states. Intuitively, equivalent states behave the same on all sequences of inputs; either both or neither lead to accepting states. Write a program using the MFSET operations that computes the sets of equivalent states of a given finite automaton.

∗∗5.21 In the tree implementation of MFSET:

    a) Show that $\Omega(n \log n)$ time is needed for certain lists of n operations if path compression is used but larger trees are permitted to be merged into smaller ones.

    b) Show that $O(n\alpha(n))$ is the worst case running time for n operations if path compression is used, and the smaller tree is always merged into the larger.

5.22 Select a data structure and write a program to compute PLACES (defined in Section 5.6) in average time $O(n)$ for strings of length n.

∗5.23 Modify the LCS procedure of Fig. 5.29 to compute the LCS, not just its length.

∗5.24 Write a detailed SPLIT procedure to work on 2-3 trees.

∗5.25 If elements of a set represented by a 2-3 tree consist only of a key field, an element whose key appears at an interior node need not appear at a leaf. Rewrite the dictionary operations to take advantage of this fact and avoid storing any element at two different nodes.

Bibliographic Notes

Tries were first proposed by Fredkin [1960]. Bayer and McCreight [1972] introduced B-trees, which, as we shall see in Chapter 11, are a generalization of 2-3 trees. The first uses of 2-3 trees were by J. E. Hopcroft in 1970 (unpublished) for insertion, deletion, concatenation, and splitting, and by Ullman [1974] for a code optimization problem.

The tree structure of Section 5.5, using path compression and merging smaller into larger, was first used by M. D. McIlroy and R. Morris to construct minimum-cost spanning trees. The performance of the tree implementation of MFSET's was analyzed by Fischer [1972] and by Hopcroft and Ullman [1973]. Exercise 5.21(b) is from Tarjan [1975].

The solution to the LCS problem of Section 5.6 is from Hunt and Szymanski [1977]. An efficient data structure for FIND, SPLIT, and the restricted MERGE (where all elements of one set are less than those of the other) is described in van Emde Boas, Kaas, and Zijlstra [1977].

Exercise 5.6 is based on an efficient algorithm for matching patterns developed by Weiner [1973]. The 2-3 tree variant of Exercise 5.16 is discussed in detail in Wirth [1976]. The AVL tree structure in Exercise 5.18 is from Adel'son-Vel'skii and Landis [1962].

CHAPTER 10

Algorithm Design Techniques

Over the years computer scientists have identified a number of general techniques that often yield effective algorithms in solving large classes of problems. This chapter presents some of the more important techniques, such as divide-and-conquer, dynamic programming, greedy techniques, backtracking, and local search. In trying to devise an algorithm to solve a gi problem, it is often useful to ask a question such as "What kind of solution does divide-and-conquer, dynamic programming, a greedy approach, or some other standard technique yield?"

It should be emphasized, however, that there are problems, such as the NP-complete problems, for which these or any other known techniques will not produce efficient solutions. When such a problem is encountered, it is often useful to determine if the inputs to the problem have special characteristics that could be exploited in trying to devise a solution, or if an easily found approximate solution could be used in place of the difficult-to-compute exact solution.

10.1 Divide-and-Conquer Algorithms

Perhaps the most important, and most widely applicable, technique for designing efficient algorithms is a strategy called "divide-and-conquer." It consists of breaking a problem of size n into smaller problems in such a way that from solutions to the smaller problems we can easily construct a solution to the entire problem. We have already seen a number of applications of this technique, such as *mergesort* or binary search trees.

To illustrate the method consider the familiar "towers of Hanoi" puzzle. It consists of three pegs A, B, and C. Initially peg A has on it some number of disks, starting with the largest one on the bottom and successively smaller ones on top, as shown in Fig. 10.1. The object of the puzzle is to move the disks one at a time from peg to peg, never placing a larger disk on top of a smaller one, eventually ending with all disks on peg B.

One soon learns that the puzzle can be solved by the following simple algorithm. Imagine the pegs arranged in a triangle. On odd-numbered moves, move the smallest disk one peg clockwise. On even-numbered moves
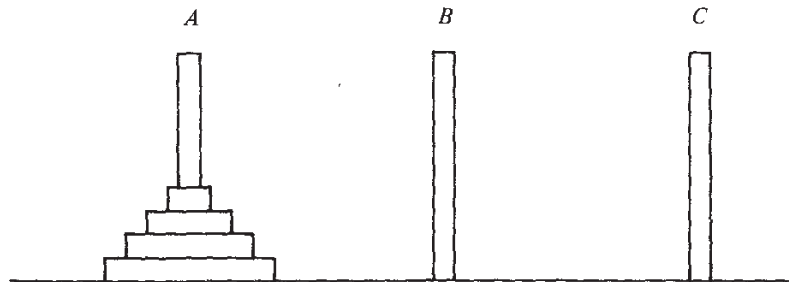
Fig. 10.1. Initial position in towers of Hanoi puzzle.

make the only legal move not involving the smallest disk.

The above algorithm is concise, and correct, but it is hard to understand why it works, and hard to invent on the spur of the moment. Consider instead the following divide-and-conquer approach. The problem of moving the n smallest disks from A to B can be thought of as consisting of two sub-problems of size $n-1$. First move the $n-1$ smallest disks from peg A to peg C, exposing the nth smallest disk on peg A. Move that disk from A to B. Then move the $n-1$ smallest disks from C to B. Moving the $n-1$ smallest disks is accomplished by a recursive application of the method. As the n disks involved in the moves are smaller than any other disks, we need not concern ourselves with what is below them on pegs A, B, or C. Although the actual movement of individual disks is not obvious, and hand simulation is hard because of the stacking of recursive calls, the algorithm is conceptually simple to understand, to prove correct and, we would like to think, to invent in the first place. It is probably the ease of discovery of divide-and-conquer algorithms that makes the technique so important, although in many cases the algorithms are also more efficient than more conventional ones.†

The Problem of Multiplying Long Integers

Consider the problem of multiplying two n-bit integers X and Y. Recall that the algorithm for multiplication of n-bit (or n-digit) integers usually taught in elementary school involves computing n partial products of size n and thus is an $O(n^2)$ algorithm, if we count single bit or digit multiplications and additions as one step. One divide-and-conquer approach to integer multiplication would break each of X and Y into two integers of $n/2$ bits each as shown in

---

† In the towers of Hanoi case, the divide-and-conquer algorithm is really the same as the one given initially.

Fig. 10.2. (For simplicity we assume n is a power of 2 here.)

$$X := \boxed{\begin{array}{c|c} A & B \end{array}} \qquad X = A2^{n/2} + B$$

$$Y := \boxed{\begin{array}{c|c} C & D \end{array}} \qquad Y = C2^{n/2} + D$$

Fig. 10.2. Breaking n-bit integers into $\frac{n}{2}$-bit pieces.

The product of X and Y can now be written

$$XY = AC2^n + (AD+BC)\, 2^{n/2} + BD \qquad (10.1)$$

If we evaluate XY in this straightforward way, we have to perform four multiplications of $(n/2)$-bit integers (AC, AD, BC, and BD), three additions of integers with at most $2n$ bits (corresponding to the three $+$ signs in (10.1)), and two shifts (multiplication by 2^n and $2^{n/2}$). As these additions and shifts take $O(n)$ steps, we can write the following recurrence for $T(n)$, the total number of bit operations needed to multiply n-bit integers according to (10.1).

$$T(1) = 1$$
$$T(n) = 4T(n/2) + cn \qquad (10.2)$$

Using reasoning like that in Example 9.4, we can take the constant c in (10.2) to be 1, so the driving function $d(n)$ is just n, and then deduce that the homogeneous and particular solutions are both $O(n^2)$.

In the case that formula (10.1) is used to multiply integers, the asymptotic efficiency is thus no greater than for the elementary school method. But recall that for equations like (10.2) we get an asymptotic improvement if we decrease the number of subproblems. It may be a surprise that we can do so, but consider the following formula for multiplying X by Y.

$$XY = AC2^n + [(A-B)(D-C) + AC + BD]2^{n/2} + BD \qquad (10.3)$$

Although (10.3) looks more complicated than (10.1) it requires only three multiplications of $(n/2)$-bit integers, AC, BD, and $(A-B)(D-C)$, six additions or subtractions, and two shifts. Since all but the multiplications take $O(n)$ steps, the time $T(n)$ to multiply n-bit integers by (10.3) is given by

$$T(1) = 1$$
$$T(n) = 3T(n/2) + cn$$

whose solution is $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$.

The complete algorithm, including the details implied by the fact that (10.3) requires multiplication of negative, as well as positive, $(n/2)$-bit

integers, is given in Fig. 10.3. Note that lines (8)−(11) are performed by copying bits, and the multiplication by 2^n and $2^{n/2}$ in line (16) by shifting. Also, the multiplication by s in line (16) simply introduces the proper sign into the result.

```
        function mult ( X, Y, n: integer ): integer;
            { X and Y are signed integers ≤ 2ⁿ.
              n is a power of 2. The function returns XY }
            var
                s: integer;   { holds the sign of XY }
                m1, m2, m3: integer; { hold the three products }
                A, B, C, D: integer;  { hold left and right halves of X and Y }
            begin
(1)             s := sign(X) * sign(Y);
(2)             X := abs(X);
(3)             Y := abs(Y); { make X and Y positive }
(4)             if n = 1 then
(5)                 if (X = 1) and (Y = 1) then
(6)                     return (s)
                    else
(7)                     return (0)
                else begin
(8)                 A := left n/2 bits of X;
(9)                 B := right n/2 bits of X;
(10)                C := left n/2 bits of Y;
(11)                D := right n/2 bits of Y;
(13)                m1 := mult(A, C, n/2);
(14)                m2 := mult(A−B, D−C, n/2);
(15)                m3 := mult(B, D, n/2);
(16)                return (s * (m1*2ⁿ + (m1 + m2 + m3) * 2^{n/2} + m3))
                end
        end; { mult }
```

Fig. 10.3. Divide-and-conquer integer multiplication algorithm.

Observe that the divide-and-conquer algorithm of Fig. 10.3 is asymptotically faster than the method taught in elementary school, taking $O(n^{1.59})$ steps against $O(n^2)$. We may thus raise the question: if this algorithm is so superior why don't we teach it in elementary school? There are two answers. First, while easy to implement on a computer, the description of the algorithm is sufficiently complex that if we attempted to teach it in elementary school students would not learn to multiply. Furthermore, we have ignored constants of proportionality. While procedure *mult* of Fig. 10.3 is asymptotically superior to the usual method, the constants are such that for small problems (actually up to about 500 bits) the elementary school method is superior, and we rarely

85

ask elementary school children to multiply such numbers.

Constructing Tennis Tournaments

The technique of divide-and-conquer has widespread applicability, not only in algorithm design but in designing circuits, constructing mathematical proofs and in other walks of life. We give one example as an illustration. Consider the design of a round robin tennis tournament schedule, for $n = 2^k$ players. Each player must play every other player, and each player must play one match per day for $n-1$ days, the minimum number of days needed to complete the tournament.

The tournament schedule is thus an n row by $n-1$ column table whose entry in row i and column j is the player i must contend with on the jth day.

The divide-and-conquer approach constructs a schedule for one-half of the players. This schedule is designed by a recursive application of the algorithm by finding a schedule for one half of these players and so on. When we get down to two players, we have the base case and we simply pair them up.

Suppose there are eight players. The schedule for players 1 through 4 fills the upper left corner (4 rows by 3 columns) of the schedule being constructed. The lower left corner (4 rows by 3 columns) of the schedule must pit the high numbered players (5 through 8) against one another. This subschedule is obtained by adding 4 to each entry in the upper left.

We have now simplified the problem. All that remains is to have lower-numbered players play high-numbered players. This is easily accomplished by having players 1 through 4 play 5 through 8 respectively on day 4 and cyclically permuting 5 through 8 on subsequent days. The process is illustrated in Fig. 10.4. The reader should now be able to generalize the ideas of this figure to provide a schedule for 2^k players for any k.

Balancing Subproblems

In designing algorithms one is always faced with various trade-offs. One rule that has emerged is that it is generally advantageous to balance competing costs wherever possible. For example in Chapter 5 we saw that the 2-3 tree balanced the costs of searching with those of inserting, while more straightforward methods take $O(n)$ steps either for each lookup or for each insertion, even though the other operation can be done in a constant number of steps.

Similarly, for divide-and-conquer algorithms, we are generally better off if the subproblems are of approximately equal size. For example, insertion sort can be viewed as partitioning a problem into two subproblems, one of size 1 and one of size $n-1$, with a maximum cost of n steps to merge. This gives a recurrence

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

which has an $O(n^2)$ solution. Mergesort, on the other hand, partitions the problems into two subproblems each of size $n/2$ and has $O(n\log n)$ performance. As a general principle, we often find that partitioning a problem into
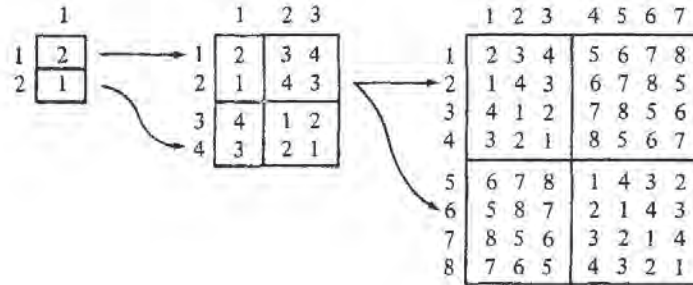
	1	2	3		1	2	3		4	5	6	7
1	2	3	4		5	6	7	8
2	1	4	3		6	7	8	5
3	4	1	2		7	8	5	6
4	3	2	1		8	5	6	7
5	6	7	8		1	4	3	2
6	5	8	7		2	1	4	3
7	8	5	6		3	2	1	4
8	7	6	5		4	3	2	1

Fig. 10.4. A round-robin tournament for eight players.

equal or nearly equal subproblems is a crucial factor in obtaining good performance.

10.2 Dynamic Programming

Often there is no way to divide a problem into a small number of subproblems whose solution can be combined to solve the original problem. In such cases we may attempt to divide the problem into as many subproblems as necessary, divide each subproblem into smaller subproblems and so on. If this is all we do, we shall likely wind up with an exponential-time algorithm.

Frequently, however, there are only a polynomial number of subproblems, and thus we must be solving some subproblem many times. If instead we keep track of the solution to each subproblem solved, and simply look up the answer when needed, we would obtain a polynomial-time algorithm.

It is sometimes simpler from an implementation point of view to create a table of the solutions to all the subproblems we might ever have to solve. We fill in the table without regard to whether or not a particular subproblem is actually needed in the overall solution. The filling-in of a table of subproblems to get a solution to a given problem has been termed *dynamic programming*, a name that comes from control theory.

The form of a dynamic programming algorithm may vary, but there is the common theme of a table to fill and an order in which the entries are to be filled. We shall illustrate the techniques by two examples, calculating odds on a match like the World Series, and the "triangulation problem."

87

World Series Odds

Suppose two teams, A and B, are playing a match to see who is the first to win n games for some particular n. The World Series is such a match, with $n = 4$. We may suppose that A and B are equally competent, so each has a 50% chance of winning any particular game. Let $P(i, j)$ be the probability that if A needs i games to win, and B needs j games, that A will eventually win the match. For example, in the World Series, if the Dodgers have won two games and the Yankees one, then $i = 2$, $j = 3$, and $P(2, 3)$, we shall discover, is 11/16.

To compute $P(i, j)$, we can use a recurrence equation in two variables. First, if $i = 0$ and $j > 0$, then team A has won the match already, so $P(0, j) = 1$. Similarly, $P(i, 0) = 0$ for $i > 0$. If i and j are both greater than 0, at least one more game must be played, and the two teams each win half the time. Thus, $P(i, j)$ must be the average of $P(i-1, j)$ and $P(i, j-1)$, the first of these being the probability A will win the match if it wins the next game and the second being the probability A wins the match even though it loses the next game. To summarize:

$$P(i, j) = 1 \quad \text{if } i = 0 \text{ and } j > 0$$
$$= 0 \quad \text{if } i > 0 \text{ and } j = 0$$
$$= (P(i-1, j) + P(i, j-1))/2 \quad \text{if } i > 0 \text{ and } j > 0 \qquad (10.4)$$

If we use (10.4) recursively as a function, we can show that $P(i, j)$ takes no more than time $O(2^{i+j})$. Let $T(n)$ be the maximum time taken by a call to $P(i, j)$, where $i+j = n$. Then from (10.4),

$$T(1) = c$$
$$T(n) = 2T(n-1) + d$$

for some constants c and d. The reader may check by the means discussed in the previous chapter that $T(n) \le 2^{n-1}c + (2^{n-1}-1)d$, which is $O(2^n)$ or $O(2^{i+j})$.

We have thus proven an exponential upper bound on the time taken by the recursive computation of $P(i, j)$. However, to convince ourselves that the recursive formula for $P(i, j)$ is a bad way to compute it, we need to get a big-omega lower bound. We leave it as an exercise to show that when we call $P(i, j)$, the total number of calls to P that gets made is $\binom{i+j}{i}$, the number of ways to choose i things out of $i+j$. If $i = j$, that number is $\Omega(2^n/\sqrt{n})$, where $n = i+j$. Thus, $T(n)$ is $\Omega(2^n/\sqrt{n})$, and in fact, we can show it is $O(2^n/\sqrt{n})$ also. While $2^n/\sqrt{n}$ grows asymptotically more slowly than 2^n, the difference is not great, and $T(n)$ grows far too fast for the recursive calculation of $P(i, j)$ to be practical.

The problem with the recursive calculation is that we wind up computing the same $P(i, j)$ repeatedly. For example, if we want to compute $P(2, 3)$, we compute, by (10.4), $P(1, 3)$ and $P(2, 2)$. $P(1, 3)$ and $P(2, 2)$ both require

the computation of $P(1, 2)$, so we compute that value twice.

A better way to compute $P(i, j)$ is to fill in the table suggested by Fig. 10.5. The bottom row is all 0's and the rightmost column all 1's by the first two lines of (10.4). By the last line of (10.4), each other entry is the average of the entry below it and the entry to the right. Thus, an appropriate way to fill in the table is to proceed in diagonals beginning at the lower right corner, and proceeding up and to the left along diagonals representing entries with a constant value of $i+j$, as suggested in Fig. 10.6. This program is given in Fig. 10.7, assuming it works on a two-dimensional array P of suitable size.

1/2	21/32	13/16	15/16	1	4	
11/32	1/2	11/16	7/8	1	3	↑
3/16	5/16	1/2	3/4	1	2	j
1/16	1/8	1/4	1/2	1	1	
0	0	0	0		0	
4	3	2	1	0		

$\leftarrow i$
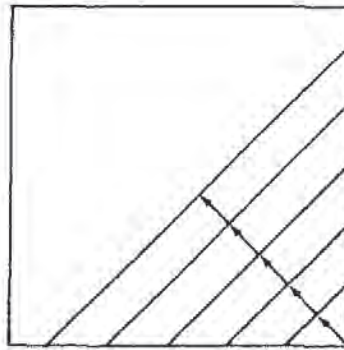
Fig. 10.5. Table of odds.

Fig. 10.6. Pattern of computation.

The analysis of function *odds* is easy. The loop of lines (4)–(5) takes $O(s)$ time, and that dominates the $O(1)$ time for lines (2)–(3). Thus, the outer loop takes time $O(\sum_{s=1}^{n} s)$ or $O(n^2)$, where $i+j = n$. Thus dynamic pro-

```
      function odds ( i, j: integer ) : real;
          var
              s, k: integer;
          begin
(1)           for s := 1 to i + j do begin
                  { compute diagonal of entries whose indices sum to s }
(2)               P[0, s] := 1.0;
(3)               P[s, 0] := 0.0;
(4)               for k := 1 to s-1 do
(5)                   P[k, s-k] := (P[k-1, s-k] + P[k, s-k-1])/2.0
              end;
(6)           return (P[i, j])
          end; { odds }
```

Fig. 10.7. Odds calculation.

gramming takes $O(n^2)$ time, compared with $O(2^n/\sqrt{n})$ for the straightforward approach. Since $2^n/\sqrt{n}$ grows wildly faster than n^2, we would prefer dynamic programming to the recursive approach under essentially any circumstances.

The Triangulation Problem

As another example of dynamic programming, consider the problem of *triangulating* a polygon. We are given the vertices of a polygon and a distance measure between each pair of vertices. The distance may be the ordinary (Euclidean) distance in the plane, or it may be an arbitrary cost function given by a table. The problem is to select a set of *chords* (lines between nonadjacent vertices) such that no two chords cross each other, and the entire polygon is divided into triangles. The total length (distance between endpoints) of the chords selected must be a minimum. We call such a set of chords a *minimal triangulation*.

Example 10.1. Figure 10.8 shows a seven-sided polygon and the (x, y) coordinates of its vertices. The distance function is the ordinary Euclidean distance. A triangulation, which happens not to be minimal, is shown by dashed lines. Its cost is the sum of the lengths of the chords (v_0, v_2), (v_0, v_3), (v_0, v_5), and (v_3, v_5), or $\sqrt{8^2+16^2} + \sqrt{15^2+16^2} + \sqrt{22^2+2^2} + \sqrt{7^2+14^2} = 77.56$. □

As well as being interesting in its own right, the triangulation problem has a number of useful applications. For example, Fuchs, Kedem, and Uselton [1977] used a generalization of the triangulation problem for the following purpose. Consider the problem of shading a two-dimensional picture of an object whose surface is defined by a collection of points in 3-space. The light source comes from a given direction, and the brightness of a point on the surface depends on the angles between the direction of light, the direction of the viewer's eye, and a perpendicular to the surface at that point. To estimate the direction of the surface at a point, we can compute a minimum triangulation
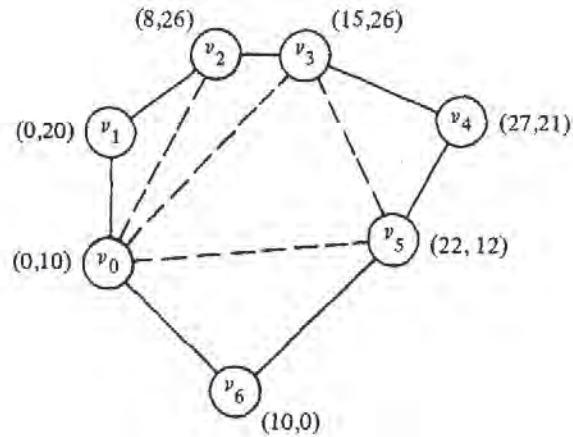
Fig. 10.8. A heptagon and a triangulation.

of the points defining the surface.

Each triangle defines a plane in a 3-space, and since a minimum triangulation was found, the triangles are expected to be very small. It is easy to find the direction of a perpendicular to a plane, so we can compute the light intensity for the points of each triangle, on the assumption that the surface can be treated as a triangular plane in a given region. If the triangles are not sufficiently small to make the light intensity look smooth, then local averaging can improve the picture.

Before proceeding with the dynamic programming solution to the triangulation problem, let us state two observations about triangulations that will help us design the algorithm. Throughout we assume we have a polygon with n vertices $v_0, v_1, \ldots, v_{n-1}$, in clockwise order.

Fact 1. In any triangulation of a polygon with more than three vertices, every pair of adjacent vertices is touched by at least one chord. To see this, suppose neither v_i nor v_{i+1}[†] were touched by a chord. Then the region that edge (v_i, v_{i+1}) bounds would have to include edges (v_{i-1}, v_i), (v_{i+1}, v_{i+2}) and at least one additional edge. This region then would not be a triangle.

Fact 2. If (v_i, v_j) is a chord in a triangulation, then there must be some v_k

---

† In what follows, we take all subscripts to be computed modulo n. Thus, in Fig. 10.8, v_i and v_{i+1} could be v_6 and v_0, respectively, since $n = 7$.

such that (v_i, v_k) and (v_k, v_j) are each either edges of the polygon or chords. Otherwise, (v_i, v_j) would bound a region that was not a triangle.

To begin searching for a minimum triangulation, we pick two adjacent vertices, say v_0 and v_1. By the two facts we know that in any triangulation, and therefore in the minimum triangulation, there must be a vertex v_k such that (v_1, v_k) and (v_k, v_0) are chords or edges in the triangulation. We must therefore consider how good a triangulation we can find after selecting each possible value for k. If the polygon has n vertices, there are a total of $(n-2)$ choices to make.

Each choice of k leads to at most two *subproblems*, which we define to be polygons formed by one chord and the edges in the original polygon from one end of the chord to the other. For example, Fig. 10.9 shows the two subproblems that result if we select the vertex v_3.
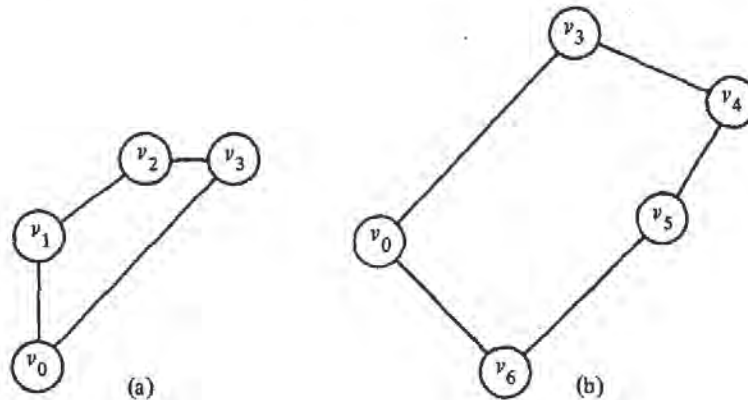

Fig. 10.9. The two subproblems after selecting v_3.

Next, we must find minimum triangulations for the polygons of Fig. 10.9(a) and (b). Our first instinct is that we must again consider all chords emanating from two adjacent vertices. For example, in solving Fig. 10.9(b), we might consider choosing chord (v_3, v_5), which leaves subproblem (v_0, v_3, v_5, v_6), a polygon two of whose sides, (v_0, v_3) and (v_3, v_5), are chords of the original polygon. This approach leads to an exponential-time algorithm.

However, by considering the triangle that involves the chord (v_0, v_k) we never have to consider polygons more than one of whose sides are chords of the original polygon. Fact 2 tells us that, in the minimal triangulation, the chord in the subproblem, such as (v_0, v_3) in Fig. 10.9(b), must make a triangle with one of the other vertices. For example, if we select v_4, we get the

triangle (v_0, v_3, v_4) and the subproblem (v_0, v_4, v_5, v_6) which has only one chord of the original polygon. If we try v_5, we get the subproblems (v_3, v_4, v_5) and (v_0, v_5, v_6), with chords (v_3, v_5) and (v_0, v_5) only.

In general, define *the subproblem of size s beginning at vertex* v_i, denoted S_{is}, to be the minimal triangulation problem for the polygon formed by the s vertices beginning at v_i and proceeding clockwise, that is, v_i, v_{i+1}, . . . , v_{i+s-1}. The chord in S_{is} is (v_i, v_{i+s-1}). For example, Fig. 10.9(a) is S_{04} and Fig. 10.9(b) is S_{35}. To solve S_{is} we must consider the following three options.

1. We may pick vertex v_{i+s-2} to make a triangle with the chords (v_i, v_{i+s-1}) and (v_i, v_{i+s-2}) and third side (v_{i+s-2}, v_{i+s-1}), and then solve the subproblem $S_{i,s-1}$.

2. We may pick vertex v_{i+1} to make a triangle with the chords (v_i, v_{i+s-1}) and (v_{i+1}, v_{i+s-1}) and third side (v_i, v_{i+1}), and then solve the subproblem $S_{i+1,s-1}$.

3. For some k between 2 and $s-3$ we may pick vertex v_{i+k} and form a triangle with sides (v_i, v_{i+k}), (v_{i+k}, v_{i+s-1}), and (v_i, v_{i+s-1}) and then solve subproblems $S_{i,k+1}$ and $S_{i+k,s-k}$.

If we remember that "solving" any subproblem of size three or less requires no action, we can summarize (1)–(3) by saying that we pick some k between 1 and $s-2$ and solve subproblems $S_{i,k+1}$ and $S_{i+k,s-k}$. Figure 10.10 illustrates this division into subproblems.
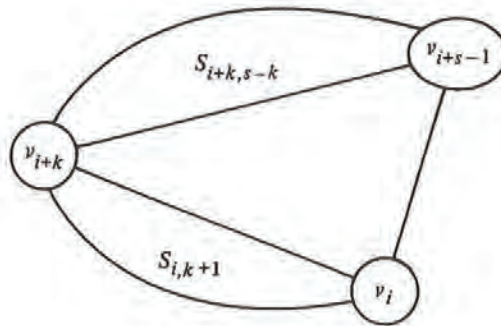

Fig. 10.10. Division of S_{is} into subproblems.

If we use the obvious recursive algorithm implied by the above rules to solve subproblems of size four or more, then it is possible to show that each call on a subproblem of size s gives rise to a total of 3^{s-4} recursive calls, if we "solve" subproblems of size three or less directly and count only calls on

subproblems of size four or more. Thus the number of subproblems to be solved is exponential in s. Since our initial problem is of size n, where n is the number of vertices in the given polygon, the total number of steps performed by this recursive procedure is exponential in n.

Yet something is clearly wrong in this analysis, because we know that besides the original problem, there are only $n(n-4)$ different subproblems that ever need to be solved. They are represented by S_{is}, where $0 \leq i < n$ and $4 \leq s < n$. Evidently not all the subproblems solved by the recursive procedure are different. For example, if in Fig. 10.8 we choose chord (v_0, v_3), and then in the subproblem of Fig. 10.9(b) we pick v_4, we have to solve subproblem S_{44}. But we would also have to solve this problem if we first picked chord (v_0, v_4), or if we picked (v_1, v_4) and then, when solving subproblem S_{45}, picked vertex v_0 to complete a triangle with v_1 and v_4.

This suggests an efficient way to solve the triangulation problem. We make a table giving the cost C_{is} of triangulating S_{is} for all i and s. Since the solution to any given problem depends only on the solution to problems of smaller size, the logical order in which to fill in the table is in size order. That is, for sizes $s = 4, 5, \ldots, n-1$ we fill in the minimum cost for problems S_{is}, for all vertices i. It is convenient to include problems of size $0 \leq s < 4$ as well, but remember that S_{is} has cost 0 if $s < 4$.

By rules (1)–(3) above for finding subproblems, the formula for computing C_{is} for $s \geq 4$ is:

$$C_{is} = \min_{1 \leq k \leq s-2} \left[C_{i,k+1} + C_{i+k,s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1}) \right] \quad (10.5)$$

where $D(v_p, v_q)$ is the length of the chord between vertices v_p and v_q, if v_p and v_q are not adjacent points on the polygon; $D(v_p, v_q)$ is 0 if v_p and v_q are adjacent.

Example 10.2. Figure 10.11 holds the table of costs for $S_{i,s}$ for $0 \leq i \leq 6$ and $4 \leq s \leq 6$, based on the polygon and distances of Fig. 10.8. The costs for the rows with $s < 3$ are all zero. We have filled in the entry C_{07}, in column 0 and the row for $s = 7$. This entry, like all in that row, represents the triangulation of the entire polygon. To see that, just notice that we can, if we wish, consider the edge (v_0, v_6) to be a chord of a larger polygon and the polygon of Fig. 10.8 to be a subproblem of this polygon, which has a series of additional vertices extending clockwise from v_6 to v_0. Note that the entire row for $s = 7$ has the same value as C_{07}, to within the accuracy of the computation.

Let us, as an example, show how the entry 38.09 in the column for $i = 6$ and row for $s = 5$ is filled in. According to (10.5) the value of this entry, C_{65}, is the minimum of three sums, corresponding to $k = 1, 2,$ or 3. These sums are:

$$C_{62} + C_{04} + D(v_6, v_0) + D(v_0, v_3)$$

$$C_{63} + C_{13} + D(v_6, v_1) + D(v_1, v_3)$$

$$C_{64} + C_{22} + D(v_6, v_2) + D(v_2, v_3)$$

7	$C_{07} =$ 75.43						
6	$C_{06} =$ 53.34	$C_{16} =$ 55.22	$C_{26} =$ 57.54	$C_{36} =$ 59.67	$C_{46} =$ 59.78	$C_{56} =$ 59.78	$C_{66} =$ 63.61
5	$C_{05} =$ 37.54	$C_{15} =$ 31.81	$C_{25} =$ 35.45	$C_{35} =$ 37.74	$C_{45} =$ 45.50	$C_{55} =$ 39.98	$C_{65} =$ 38.09
4	$C_{04} =$ 16.16	$C_{14} =$ 16.16	$C_{24} =$ 15.65	$C_{34} =$ 15.65	$C_{44} =$ 22.09	$C_{54} =$ 22.09	$C_{64} =$ 17.89
s	$i = 0$	1	2	3	4	5	6

Fig. 10.11. Table of C_{is}'s.

The distances we need are calculated from the coordinates of the vertices as:

$$D(v_2, v_3) = D(v_6, v_0) = 0$$

(since these are polygon edges, not chords, and are present "for free")

$$D(v_6, v_2) = 26.08$$
$$D(v_1, v_3) = 16.16$$
$$D(v_6, v_1) = 22.36$$
$$D(v_0, v_3) = 21.93$$

The three sums above are 38.09, 38.52, and 43.97, respectively. We may conclude that the minimum cost of the subproblem S_{65} is 38.09. Moreover, since the first sum was smallest, we know that to achieve this minimum we must utilize the subproblems S_{62} and S_{04}, that is, select chord (v_0, v_3) and then solve S_{64} as best we can; chord (v_1, v_3) is the preferred choice for that subproblem. □

There is a useful trick for filling out the table of Fig. 10.11 according to the formula (10.5). Each term of the min operation in (10.5) requires a pair of entries. The first pair, for $k = 1$, can be found in the table (a) at the "bottom" (the row for $s = 2$)† of the column of the element being computed, and (b) just below and to the right‡ of the element being computed. The second pair is (a) next to the bottom of the column, and (b) two positions down and to the right. Fig. 10.12 shows the two lines of entries we follow to get all the pairs of entries we need to consider simultaneously. The pattern — up the

---

† Remember that the table of Fig. 10.11 has rows of 0's below those shown.
‡ By "to the right" we mean in the sense of a table that wraps around. Thus, if we are at the rightmost column, the column "to the right" is the leftmost column.

column and down the diagonal — is a common one in filling tables during dynamic programming.
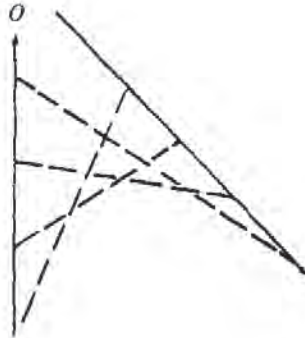

Fig. 10.12. Pattern of table scan to compute one element.

Finding Solutions from the Table

While Fig. 10.11 gives us the cost of the minimum triangulation, it does not immediately give us the triangulation itself. What we need, for each entry, is the value of k that produced the minimum in (10.5). Then we can deduce that the solution consists of chords (v_i, v_{i+k}), and (v_{i+k}, v_{i+s-1}) (unless one of them is not a chord, because $k = 1$ or $k = s-2$), plus whatever chords are implied by the solutions to $S_{i,k+1}$ and $S_{i+k,s-k}$. It is useful, when we compute an element of the table, to include with it the value of k that gave the best solution.

Example 10.3. In Fig. 10.11, the entry C_{07}, which represents the solution to the entire problem of Fig. 10.8, comes from the terms for $k = 5$ in (10.5). That is, the problem S_{07} is split into S_{06} and S_{52}; the former is the problem with six vertices $v_0, v_1, \ldots, v_5$, and the latter is a trivial "problem" of cost 0. Thus we introduce the chord (v_0, v_5) of cost 22.09 and must solve S_{06}.

The minimum cost for C_{06} comes from the terms for $k = 2$ in (10.5), whereby the problem S_{06} is split into S_{03} and S_{24}. The former is the triangle with vertices v_0, v_1, and v_2, while the latter is the quadrilateral defined by v_2, v_3, v_4, and v_5. S_{03} need not be solved, but S_{24} must be, and we must include the costs of chords (v_0, v_2) and (v_2, v_5) which are 17.89 and 19.80, respectively. We find the minimum value for C_{24} is assumed when $k = 1$ in (10.5), giving us the subproblems C_{22} and C_{33}, both of which have size less than or equal to three and therefore cost 0. The chord (v_3, v_5) is introduced, with a cost of 15.65. □
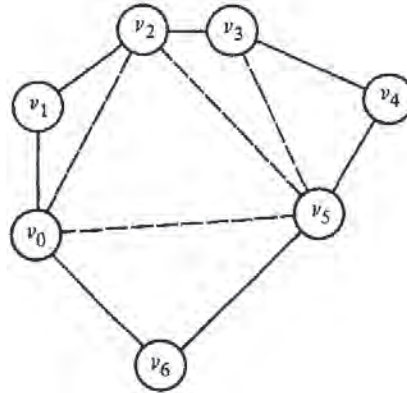
96

Fig. 10.13. A minimal cost triangulation.

10.3 Greedy Algorithms

Consider the problem of making change. Assume coins of values 25¢ (quarter), 10¢ (dime), 5¢ (nickel) and 1¢ (penny), and suppose we want to return 63¢ in change. Almost without thinking we convert this amount to two quarters, one dime and three pennies. Not only were we able to determine quickly a list of coins with the correct value, but we produced the shortest list of coins with that value.

The algorithm the reader probably used was to select the largest coin whose value was not greater than 63¢ (a quarter), add it to the list and subtract its value from 63 getting 38¢. We then selected the largest coin whose value was not greater than 38¢ (another quarter) and added it to the list, and so on.

This method of making change is a *greedy algorithm*. At any individual stage a greedy algorithm selects that option which is "locally optimal" in some particular sense. Note that the greedy algorithm for making change produces an overall optimal solution only because of special properties of the coins. If the coins had values 1¢, 5¢, and 11¢ and we were to make change of 15¢, the greedy algorithm would first select an 11¢ coin and then four 1¢ coins, for a total of five coins. However, three 5¢ coins would suffice.

We have already seen several greedy algorithms, such as Dijkstra's shortest path algorithm and Kruskal's minimum cost spanning tree algorithm. Dijkstra's shortest path algorithm is "greedy" in the sense that it always

97

chooses the closest vertex to the source among those whose shortest path is not yet known. Kruskal's algorithm is also "greedy"; it picks from the remaining edges the shortest among those that do not create a cycle.

We should emphasize that not every greedy approach succeeds in producing the best result overall. Just as in life, a greedy strategy may produce a good result for a while, yet the overall result may be poor. As an example, we might consider what happens when we allow negative-weight edges in Dijkstra's and Kruskal's algorithms. It turns out that Kruskal's spanning tree algorithm is not affected; it still produces the minimum cost tree. But Dijkstra's algorithm fails to produce shortest paths in some cases.

Example 10.4. We see in Fig. 10.14 a graph with a negative cost edge between b and c. If we apply Dijkstra's algorithm with source s, we correctly discover first that the minimum path to a has length 1. Now, considering only edges from s or a to b or c, we expect that b has the shortest path from s, namely $s \to a \to b$, of length 3. We then discover that c has a shortest path from s of length 1.
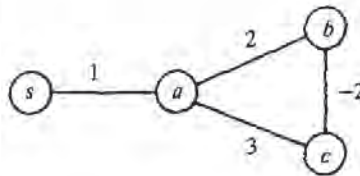

Fig. 10.14. Graph with negative weight edge.

However, our "greedy" selection of b before c was wrong from a global point of view. It turns out that the path $s \to a \to c \to b$ has length only 2, so our minimum distance of 3 for b was wrong.† □

Greedy Algorithms as Heuristics

For some problems no known greedy algorithm produces an optimal solution, yet there are greedy algorithms that can be relied upon to produce "good" solutions with high probability. Frequently, a suboptimal solution with a cost a few percent above optimal is quite satisfactory. In these cases, a greedy algorithm often provides the fastest way to get a "good" solution. In fact, if the problem is such that the only way to get an optimal solution is to use an

---

† In fact, we should be careful what we mean by "shortest path" when there are negative edges. If we allow negative cost cycles, then we could traverse such a cycle repeatedly to get arbitrarily large negative distances, so presumably we want to restrict ourselves to acyclic paths.

exhaustive search technique, then a greedy algorithm or other heuristic for getting a good, but not necessarily optimal, solution may be our only real choice.

Example 10.5. Let us introduce a famous problem where the only known algorithms that produce optimal solutions are of the "try-all-possibilities" variety and can have running times that are exponential in the size of the input. The problem, called the *traveling salesman problem*, or *TSP*, is to find, in an undirected graph with weights on the edges, a *tour* (a simple cycle that includes all the vertices) the sum of whose edge-weights is a minimum. A tour is often called a Hamilton (or Hamiltonian) cycle.

Figure 10.15(a) shows one instance of the traveling salesman problem, a graph with six vertices (often called "cities"). The coordinates of each vertex are given, and we take the weight of each edge to be its length. Note that, as is conventional with the TSP, we assume all edges exist, that is, the graph is complete. In more general instances, where the weight of edges is not based on Euclidean distance, we might find a weight of infinity on an edge that really was not present.

Figure 10.15(b)−(e) shows four tours of the six "cities" of Fig. 10.15(a). The reader might ponder which of these four, if any, is optimal. The lengths of these four tours are 50.00, 49.73, 48.39, and 49.78, respectively; (d) is the shortest of all possible tours.

(a) six "cities"

$c \bullet (1, 7)$ $d \bullet (15, 7)$
$e \bullet (15, 4)$
$b \bullet (4, 3)$
$a \bullet (0, 0)$ $f \bullet (18, 0)$
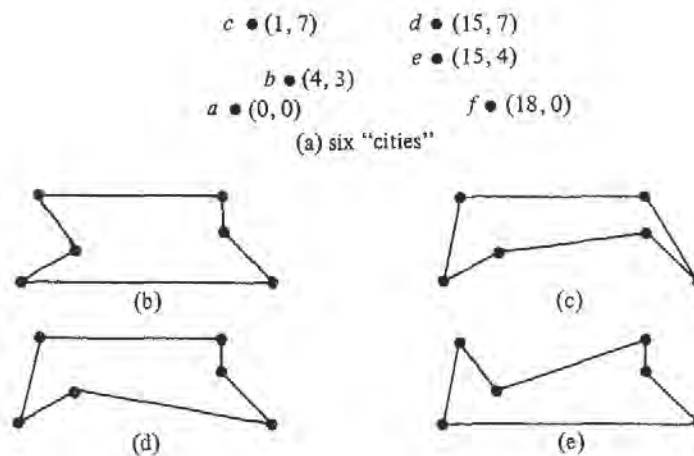
Fig. 10.15. An instance of the traveling salesman problem.

The TSP has a number of practical applications. As its name implies, it

can be used to route a person who must visit a number of points and return to his starting point. For example, the TSP has been used to route collectors of coins from pay phones. The vertices are the phones and the "home base." The cost of each edge is the travel time between the two points in question.

Another "application" of the TSP is in solving the *knight's tour problem*: find a sequence of moves whereby a knight can visit each square of the chessboard exactly once and return to its starting point. Let the vertices be the chessboard squares and let the edges between two squares that are a knight's move apart have weight 0; all other edges have weight infinity. An optimal tour has weight 0 and must be a knight's tour. Surprisingly, good heuristics for the TSP have no trouble finding knight's tours, although finding one "by hand" is a challenge.

The greedy algorithm for the TSP we have in mind is a variant of Kruskal's algorithm. Like that algorithm, we shall consider edges shortest first. In Kruskal's algorithm we accept an edge in its turn if it does not form a cycle with the edges already accepted, and we reject the edge otherwise. For the TSP, the acceptance criterion is that an edge under consideration, together with already selected edges,

1. does not cause a vertex to have degree three or more, and

2. does not form a cycle, unless the number of selected edges equals the number of vertices in the problem.

Collections of edges selected under these criteria will form a collection of unconnected paths, until the last step, when the single remaining path is closed to form a tour.

In Fig. 10.15(a), we would first pick edge (d, e), since it is the shortest, having length 3. Then we consider edges (b, c), (a, b), and (e, f), all of which have length 5. It doesn't matter in which order we consider them; all meet the conditions for selection, and we must select them if we are to follow the greedy approach. Next shortest edge is (a, c), with length 7.08. However, this edge would form a cycle with (a, b) and (b, c), so we reject it. Edge (d, f) is next rejected on similar grounds. Edge (b, e) is next to be considered, but it must be rejected because it would raise the degrees of b and e to three, and could then never form a tour with what we had selected. Similarly we reject (b, d). Next considered is (c, d), which is accepted. We now have one path, $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f$, and eventually accept (a, f) to complete the tour. The resulting tour is Fig. 10.15(b), which is fourth best of all the tours, but less than 4% more costly than the optimal. □

10.4 Backtracking

Sometimes we are faced with the task of finding an optimal solution to a problem, yet there appears to be no applicable theory to help us find the optimum, except by resorting to exhaustive search. We shall devote this section to a systematic, exhaustive searching technique called backtracking and a technique called alpha-beta pruning, which frequently reduces the search substantially.

Consider a game such as chess, checkers, or tic-tac-toe, where there are two players. The players alternate moves, and the state of the game can be represented by a board position. Let us assume that there are a finite number of board positions and that the game has some sort of stopping rule to ensure termination. With each such game, we can associate a tree called the *game tree*. Each node of the tree represents a board position. With the root we associate the starting position. If board position *x* is associated with node *n*, then the children of *n* correspond to the set of allowable moves from board position *x*, and with each child is associated the resulting board position. For example, Fig. 10.16 shows part of the tree for tic-tac-toe.
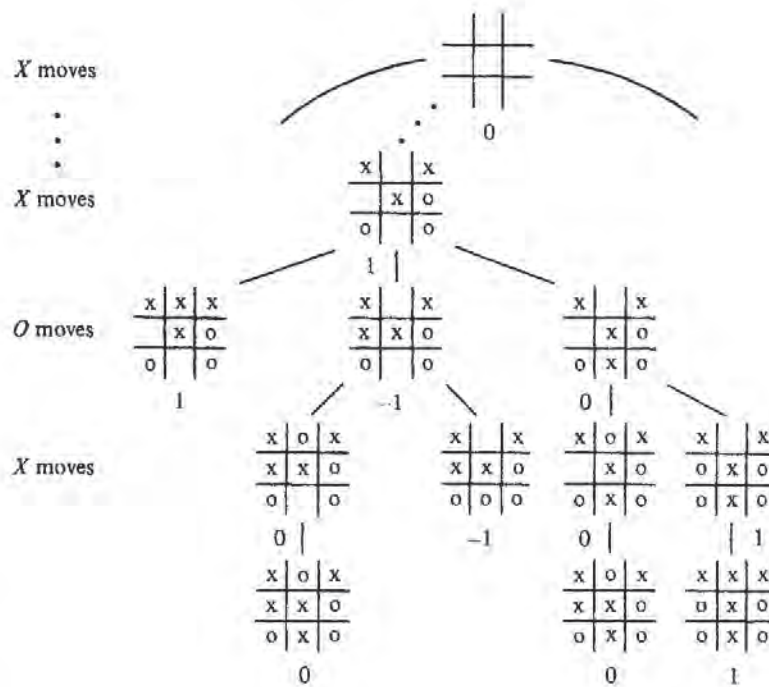

Fig. 10.16. Part of the tic-tac-toe game tree.

The leaves of the tree correspond to board positions where there is no move, either because one of the players has won or because all squares are filled and a draw resulted. We associate a value with each node of the tree.

101

First we assign values to the leaves. Say the game is tic-tac-toe. Then a leaf is assigned -1, 0 or 1 depending on whether the board position corresponds to a loss, draw or win for player 1 (playing X).

The values are propagated up the tree according to the following rule. If a node corresponds to a board position where it is player 1's move, then the value is the maximum of the values of the children of that node. That is, we assume player 1 will make the move most favorable to himself i.e., that which produces the highest-valued outcome. If the node corresponds to player 2's move, then the value is the minimum of the values of the children. That is, we assume player 2 will make his most favorable move, producing a loss for player 1 if possible, and a draw as next preference.

Example 10.6. The values of the boards have been marked in Fig. 10.16. The leaves that are wins for O get value -1, while those that are draws get 0, and wins for X get $+1$. Then we proceed up the tree. On level 8, where only one empty square remains, and it is X's move, the values for the unresolved boards is the "maximum" of the one child at level 9.

On level 7, where it is O's move and there are two choices, we take as a value for an interior node the minimum of the values of its children. The leftmost board shown on level 7 is a leaf and has value 1, because it is a win for X. The second board on level 7 has value $\min(0, -1) = -1$, while the third board has value $\min(0, 1) = 0$. The one board shown at level 6, it being X's move on that level, has value $\max(1, -1, 0) = 1$, meaning that there is some choice X can make that will win; in this case the win is immediate. □

Note that if the root has value 1, then player 1 has a winning strategy. Whenever it is his turn he is guaranteed that he can select a move that leads to a board position of value 1. Whenever it is player 2's move he has no real choice but to select a moving leading to a board position of value 1, a loss for him. The fact that a game is assumed to terminate guarantees an eventual win for the first player. If the root has value 0, as it does in tic-tac-toe, then neither player has a winning strategy but can only guarantee himself a draw by playing as well as possible. If the root has value -1, then player 2 has a winning strategy.

Payoff Functions

The idea of a game tree, where nodes have values -1, 0, and 1, can be generalized to trees where leaves are given any number (called the *payoff*) as a value, and the same rules for evaluating interior nodes applies: take the maximum of the children on those levels where player 1 is to move, and the minimum of the children on levels where player 2 moves.

As an example where general payoffs are useful, consider a complex game, like chess, where the game tree, though finite, is so huge that evaluating it from the bottom up is not feasible. Chess programs work, in essence, by building for each board position from which it must move, the game tree with that board as root, extending downward for several levels; the exact

number of levels depends on the speed with which the computer can work. As most of the leaves of the tree will be ambiguous, neither wins, losses, nor draws, each program uses a function of board positions that attempts to estimate the probability of the computer winning in that position. For example, the difference in material figures heavily into such an estimation, as do such factors as the defensive strength around the kings. By using this payoff function, the computer can estimate the probability of a win after making each of its possible next moves, on the assumption of subsequent best play by each side, and chose the move with the highest payoff.†

Implementing Backtrack Search

Suppose we are given the rules for a game,‡ that is, its legal moves and rules for termination. We wish to construct its game tree and evaluate the root. We could construct the tree in the obvious way, and then visit the nodes in postorder. The postorder traversal assures that we visit an interior node n after all its children, whereupon we can evaluate n by taking the min or max of the values of its children, as appropriate.

The space to store the tree can be prohibitively large, but by being careful we need never store more than one path, from the root to some node, at any one time. In Fig. 10.17 we see the sketch of a recursive program that represents the path in the tree by the sequence of active procedure calls at any time. That program assumes the following:

1. Payoffs are real numbers in a limited range, for example −1 to +1.

2. The constant ∞ is larger than any positive payoff and its negation is smaller than any negative payoff.

3. The type modetype is declared

    type
    modetype = (MIN, MAX)

4. There is a type boardtype declared in some manner suitable for the representation of board positions.

5. There is a function *payoff* that computes the payoff for any board that is a *leaf* (i.e., won, lost, or drawn position).

---

† Incidentally, some of the other things good chessplaying programs do are:

1. Use heuristics to eliminate from consideration certain moves that are unlikely to be good. This helps expand the tree to more levels in a fixed time.

2. Expand "capture chains", which are sequences of capturing moves beyond the last level to which the tree is normally expanded. This helps estimate the relative material strength of positions more accurately.

3. Prune the tree search by alpha-beta pruning, as discussed later in this section.

‡ We should not imply that only "games" can be solved in this manner. As we shall see in subsequent examples, the "game" could really represent the solution to a practical problem.

```
       function search ( B: boardtype; mode: modetype ) : real;
              { evaluates the payoff for board B, assuming it is
                player 1's move if mode = MAX and player 2's move
                if mode = MIN. Returns the payoff }
              var
                  C: boardtype;  { a child of board B }
                  value: real;  { temporary minimum or maximum value }
              begin
(1)               if B is a leaf then
(2)                   return (payoff(B))
                  else begin
                      { initialize minimum or maximum value of children }
(3)                   if mode = MAX then
(4)                       value := −∞
                      else
(5)                       value := ∞;
(6)                   for each child C of board B do
(7)                       if mode = MAX then
(8)                           value := max(value, search(C, MIN))
                          else
(9)                           value := min(value, search(C, MAX));
(10)                  return (value)
                  end
       end; { search }
```

Fig. 10.17. Recursive backtrack search program.

Another implementation we might consider is to use a nonrecursive program that keeps a stack of boards corresponding to the sequence of active calls to *search*. The techniques discussed in Section 2.6 can be used to construct such a program.

Alpha-Beta Pruning

There is a simple observation that allows us to eliminate from consideration much of a typical game tree. In terms of Fig. 10.17, the for-loop of line (6) can skip over certain children, often many of the children. Suppose we have a node n, as in Fig. 10.18, and we have already determined that c_1, the first of n's children, has a value of 20. As n is a max node, we know its value is at least 20. Now suppose that continuing with our search we find that d, a child of c_2 has value 15. As c_2, another child of n, is a min node, we know the value of c_2 cannot exceed 15. Thus, whatever value c_2 has, it cannot affect the value of n or any parent of n.

It is thus possible in the situation of Fig. 10.18, to skip consideration of the children of c_2 that we have not already examined. The general rules for
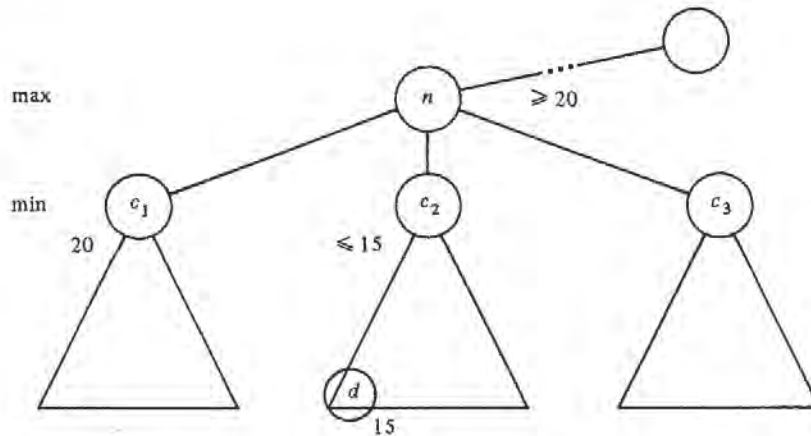
Fig. 10.18. Pruning the children of a node.

skipping or "pruning" nodes involves the notion of final and tentative values for nodes. The *final* value is what we have simply been calling the "value." A *tentative* value is an upper bound on the value of a min node, or a lower bound on the value of a max node. The rules for computing final and tentative values are the following.

1. If all the children of a node n have been considered or pruned, make the tentative value of n final.

2. If a max node n has tentative value v_1 and a child with final value v_2, then set the tentative value of n to $\max(v_1, v_2)$. If n is a min node, set its tentative value to $\min(v_1, v_2)$.

3. If p is a min node with parent q (a max node), and p and q have tentative values v_1 and v_2, respectively, with $v_1 \leq v_2$, then we may prune all the unconsidered children of p. We may also prune the unconsidered children of p if p is a max node (and therefore q is a min node) and $v_2 \leq v_1$.

Example 10.7. Consider the tree in Fig. 10.19. Assuming values for the leaves as shown, we wish to calculate the value for the root. We begin a post-order traversal. After reaching node D, by rule (2) we assign a tentative value of 2, which is the final value of D, to node C. We then search E and return to C and then to B. By rule (1), the final value of C is fixed at 2 and the value of B is tentatively set to 2. The search continues down to G and then back to F. The value F is tentatively set to 6. By rule (3), with p and q equal to F and B, respectively, we may prune H. That is, there is no need to search node H, since the tentative value of F can never decrease and it is already greater than the value of B, which can never increase.
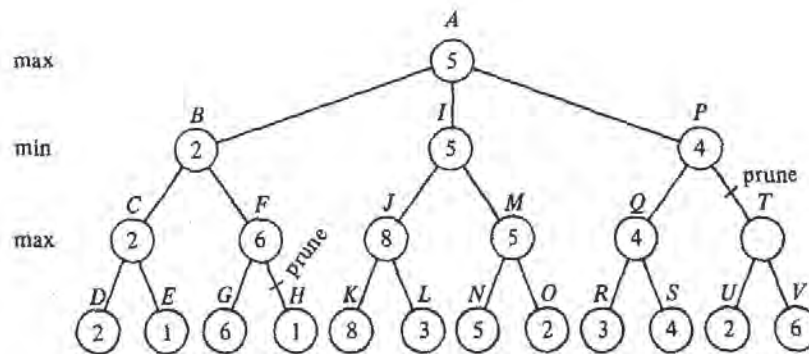
Fig. 10.19. A game tree.

Continuing our example, A is assigned a tentative value of 2 and the search proceeds to K. J is assigned a tentative value of 8. L does not determine the value of max node J. I is assigned a tentative value of 8. The search goes down to N, and M is assigned a tentative value of 5. Node O must be searched, since 5, the tentative value of M, is less than the tentative value of I. The tentative values of I and A are revised, and the search goes down to R. Eventually R and S are searched, and P is assigned a tentative value of 4. We need not search T or below, since that can only lower P's value and it is already too low to affect the value of A. □

Branch-and-Bound Search

Games are not the only sorts of "problems" that can be solved exhaustively by searching a complete tree of possibilities. A wide variety of problems where we are asked to find a minimum or maximum configuration of some sort are amenable to solution by backtracking search over a tree of all possibilities. The nodes of the tree can be thought of as sets of configurations, and the children of a node n each represent a subset of the configurations that n represents. Finally, the leaves each represent single configurations, or solutions to the problem, and we may evaluate each such configuration to see if it is the best solution found so far.

If we are reasonably clever in how we design the search, the children of a node will each represent far fewer configurations than the node itself, so we need not go to too great a depth before reaching leaves. Lest this notion of searching appear too vague, let us take a concrete example.

Example 10.8. Recall from the previous section our discussion of the

traveling salesman problem. There we gave a "greedy algorithm" for finding a good but not necessarily optimum tour. Now let us consider how we might find the optimum tour by systematically considering all tours. One way is to consider all permutations of the nodes, and evaluate the tour that visits the nodes in that order, remembering the best found so far. The time for such an approach is $O(n!)$ on an n node graph, since we must consider $(n-1)!$ different permutations,† and each permutation takes $O(n)$ time to evaluate.

We shall consider a somewhat different approach that is no better than the above in the worst case, but on the average, when coupled with a technique called "branch-and-bound" that we shall discuss shortly, produces the answer far more rapidly than the "try all permutations" method. Start constructing a tree, with a root that represents all tours. Tours are what we called "configurations" in the prefatory material. Each node has two children, and the tours that a node represents are divided by these children into two groups — those that have a particular edge and those that do not. For example, Fig. 10.20 shows portions of the tree for the TSP instance from Fig. 10.15(a).

In Fig. 10.20 we have chosen to consider the edges in lexicographic order $(a, b), (a, c), \ldots, (a, f), (b, c), \ldots, (b, f), (c, d)$, and so on. We could, of course pick any other order. Observe that not every node in the tree has two children. We can eliminate some children because the edges selected do not form part of a tour. Thus, there is no node for "tours containing $(a, b), (a, c)$, and (a, d)," because a would have degree 3 and the result would not be a tour. Similarly, as we go down the tree we shall eliminate some nodes because some city would have degree less than 2. For example, we shall find no node for tours without $(a, b), (a, c), (a, d)$, or (a, e). □

Bounding Heuristics Needed for Branch-and-Bound

Using ideas similar to those in alpha-beta pruning, we can eliminate far more nodes of the search tree than would be suggested by Example 10.8. Suppose, to be specific, that our problem is to minimize some function, e.g., the cost of a tour in the TSP. Suppose also that we have a method for getting a lower bound on the cost of any solution among those in the set of solutions represented by some node n. If the best solution found so far costs less than the lower bound for node n, we need not explore any of the nodes below n.

Example 10.9. We shall discuss one way to get a lower bound on certain sets of solutions for the TSP, those sets represented by nodes in a tree of solutions as suggested in Fig. 10.20. First of all, suppose we wish a lower bound on all solutions to a given instance of the TSP. Observe that the cost of any tour can be expressed as one half the sum over all nodes n of the cost of the two tour edges adjacent to n. This remark leads to the following rule. The sum

---

† Note that we need not consider all $n!$ permutations, since the starting point of a tour is immaterial. We may therefore consider only those permutations that begin with 1.
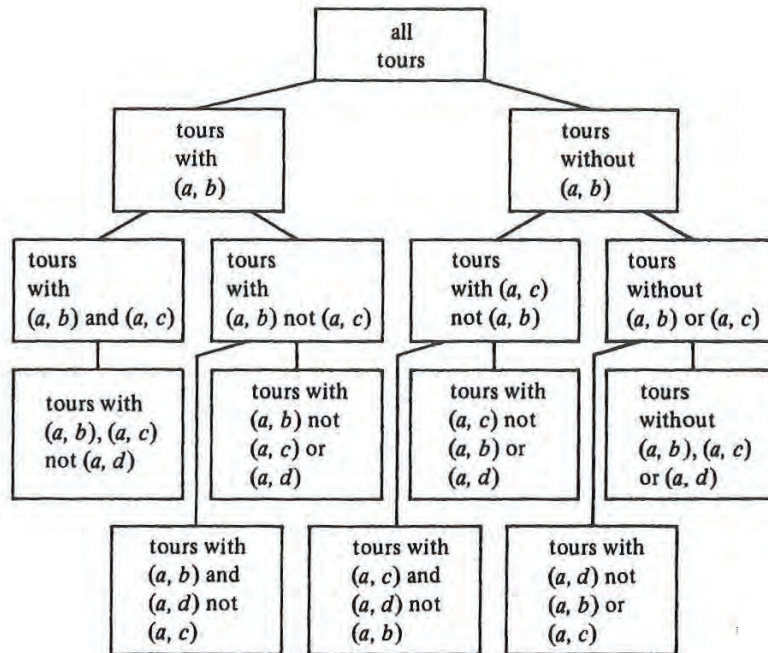
Fig. 10.20. Beginning of a solution tree for a TSP instance.

of the two tour edges adjacent to node n is no less than the sum of the two edges of least cost adjacent to n. Thus, no tour can cost less than one half the sum over all nodes n of the two lowest cost edges incident upon n.

For example, consider the TSP instance in Fig. 10.21. Unlike the instance in Fig. 10.15, the distance measure for edges is not Euclidean; that is, it bears no relation to the distance in the plane between the cities it connects. Such a cost measure might be traveling time or fare, for example. In this instance, the least cost edges adjacent to node a are (a, d), and (a, b), with a total cost of 5. For node b, we have (a, b) and (b, e), with a total cost of 6. Similarly, the two lowest cost edges adjacent to c, d, and e, total 8, 7, and 9, respectively. Our lower bound on the cost of a tour is thus $(5+6+8+7+9)/2 = 17.5$.
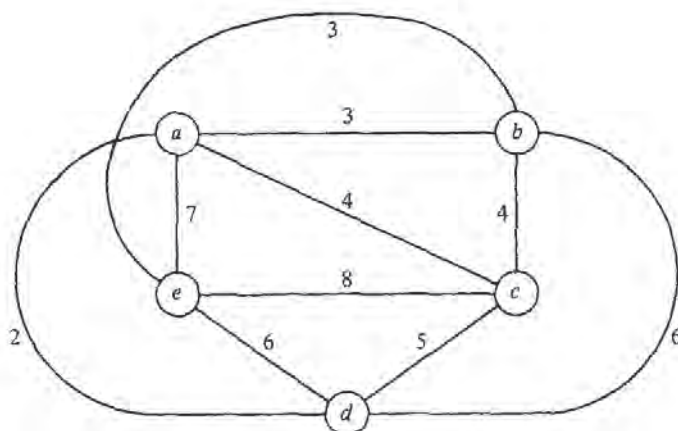
Fig. 10.21. An instance of TSP.

Now suppose we want a lower bound on the cost of a subset of tours defined by some node in the search tree. If the search tree is constructed as in Example 10.8, each node represents tours defined by a set of edges that must be in the tour and a set of edges that may not be in the tour. These *constraints* alter our choices for the two lowest-cost edges at each node. Obviously an edge constrained to be in any tour must be included among the two edges selected, no matter whether they are or are not lowest or second lowest in cost.† Similarly, an edge constrained to be out cannot be selected, even if its cost is lowest.

Thus, if we are constrained to include edge (a, e), and exclude (b, c), the two edges for node a are (a, d) and (a, e), with a total cost of 9. For b we select (a, b) and (b, e), as before, with a total cost of 6. For c, we cannot select (b, c), and so select (a, c) and (c, d), with a total cost of 9. For d we select (a, d) and (c, d), as before, while for e we must select (a, e), and choose to select (b, e). The lower bound for these constraints is thus $(9+6+9+7+10)/2 = 20.5$. □

Now let us construct the search tree along the lines suggested in Example 10.8. We consider the edges in lexicographic order, as in that example. Each

---

† The rules for constructing the search tree will be seen to eliminate any set of constraints that cannot yield any tour, e.g., because three edges adjacent to one node are required to be in the tour.

time we *branch*, by considering the two children of a node, we try to infer additional decisions regarding which edges must be included or excluded from tours represented by those nodes. The rules we use for these inference are:

1. If excluding an edge (x, y) would make it impossible for x or y to have as many as two adjacent edges in the tour, then (x, y) must be included.

2. If including (x, y) would cause x or y to have more than two edges adjacent in the tour, or would complete a non-tour cycle with edges already included, then (x, y) must be excluded.

When we branch, after making what inferences we can, we compute lower bounds for both children. If the lower bound for a child is as high or higher than the lowest cost tour found so far, we can "prune" that child and need not construct or consider its descendants. Interestingly, there are situations where the lower bound for a node n is lower than the best solution so far, yet both children of n can be pruned because their lower bounds exceed the cost of the best solution so far.

If neither child can be pruned, we shall, as a heuristic, consider the child with the smaller lower bound first, in the hope of more rapidly reaching a solution that is cheaper than the one so far found best.† After considering one child, we must consider again whether its sibling can be pruned, since a new best solution may have been found. For the instance of Fig. 10.21, we get the search tree of Fig. 10.22. To interpret nodes of that tree, it helps to understand that the capital letters are names of the search tree nodes. The numbers are the lower bounds, and we list the constraints applying to that node but none of its ancestors by writing xy if edge (x, y) must be included and $\overline{xy}$ if (x, y) must be excluded. Also note that the constraints introduced at a node apply to all its descendants. Thus to get all the constraints applying at a node we must follow the path from that node to the root.

Lastly, let us remark that as for backtrack search in general, we construct the tree one node at a time, retaining only one path, as in the recursive algorithm of Fig. 10.17, or its nonrecursive counterpart. The nonrecursive version is probably to be preferred, so that we can maintain the list of constraints conveniently on a stack.

Example 10.10. Figure 10.22 shows the search tree for the TSP instance of Fig. 10.21. To see how it is constructed, we begin at the root A of Fig. 10.22. The first edge in lexicographic order is (a, b), so we consider the two children B and C, corresponding to the constraints ab and $\overline{ab}$, respectively. There is, as yet, no "best solution so far," so we shall consider both B and C eventually.‡ Forcing (a, b) to be included does not raise the lower bound, but

---

† An alternative is to use a heuristic to obtain a good solution using the constraints required for each child. For example, the reader should be able to modify the greedy TSP algorithm to respect constraints.

‡ We could start with some heuristically found solution, say the greedy one, although that would not affect this example. The greedy solution for Fig. 10.21 has cost 21.

excluding it raises the bound to 18.5, since the two cheapest legal edges for nodes a and b now total 6 and 7, respectively, compared with 5 and 6 with no constraints. Following our heuristic, we shall consider the descendants of node B first.

The next edge in lexicographic order is (a, c). We thus introduce children D and E corresponding to tours where (a, c) is included and excluded, respectively. In node D, we can infer that neither (a, d) nor (a, e) can be in a tour, else a would have too many edges incident. Following our heuristic we consider E before D, and branch on edge (a, d). The children F and G are introduced with lower bounds 18 and 23, respectively. For each of these children we know about three of the edges incident upon a, and so can infer something about the remaining edge (a, e).

Consider the children of F first. The first remaining edge in lexicographic order is (b, c). If we include (b, c), then, as we have included (a, b), we cannot include (b, d) or (b, e). As we have eliminated (a, e) and (b, e), we must have (c, e) and (d, e). We cannot have (c, d) or c and d would have three incident edges. We are left with one tour (a, b, c, e, d, a), whose cost is 23. Similarly, node I, where (b, c) is excluded, can be proved to represent only the tour (a, b, e, c, d, a), of cost 21. That tour has the lowest cost found so far.

We now backtrack to E and consider its second child, G. But G has a lower bound of 23, which exceeds the best cost so far, 21. Thus we prune G. We now backtrack to B and consider its other child, D. The lower bound on D is 20.5, but since costs are integers, we know no tour represented by D can have cost less than 21. Since we already have a tour that cheap, we need not explore the descendants of D, and therefore prune D. Now we backtrack to A and consider its second child, C.

At the level of node C, we have only considered edge (a, b). Nodes J and K are introduced as children of C. J corresponds to those tours that have (a, c) but not (a, b), and its lower bound in 18.5. K corresponds to tours having neither (a, b) nor (a, c), and we may infer that those tours have (a, d) and (a, e). The lower bound for K is 21, and we may immediately prune K, since we already know a tour that is low in cost.

We next consider the children of J, which are L and M, and we prune M because its lower bound exceeds the best tour cost so far. The children of L are N and P, corresponding to tours that have (b, c), and that exclude (b, c). By considering the degree of nodes b and c, and remembering that the selected edges cannot form a cycle of fewer than all five cities, we can infer that nodes N and P each represent single tours. One of these, (a, c, b, e, d, a), has the lowest cost of any tour, 19. We have explored or pruned the entire tree and therefore end. □
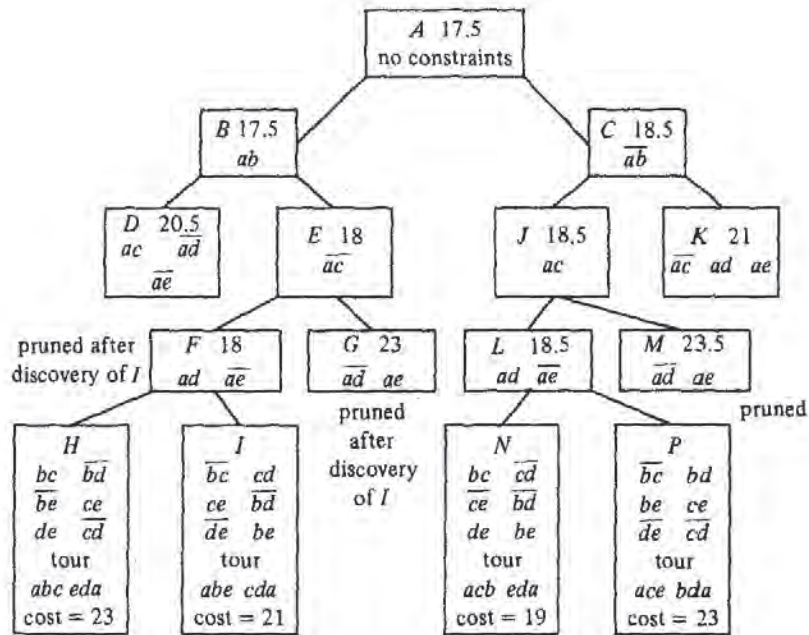
Fig. 10.22. Search tree for TSP solution.

10.5 Local Search Algorithms

Sometimes the following strategy will produce an optimal solution to a problem.

1. Start with a random solution.

2. Apply to the current solution a transformation from some given set of transformations to improve the solution. The improvement becomes the new "current" solution.

3. Repeat until no transformation in the set improves the current solution.

   The resulting solution may or may not be optimal. In principle, if the

"given set of transformations" includes all the transformations that take one solution and replace it by any other, then we shall never stop until we reach an optimal solution. But then the time to apply (2) above is the same as the time needed to examine all solutions, and the whole approach is rather point-less.

The method makes sense when we can restrict our set of transformations to a small set, so we can consider all transformations in a short time; perhaps $O(n^2)$ or $O(n^3)$ transformations should be allowed when the problem is of "size" n. If the transformation set is small, it is natural to view the solutions that can be transformed to one another in one step as "close." The transfor-mations are called "local transformations," and the method is called *local search*.

Example 10.11. One problem we can solve exactly by local search is the minimal spanning tree problem. The local transformations are those in which we take some edge not in the current spanning tree, add it to the tree, which must produce a unique cycle, and then eliminate exactly one edge of the cycle (presumably that of highest cost) to form a new tree.

For example, consider the graph of Fig. 10.21. We might start with the tree shown in Fig. 10.23(a). One transformation we could perform is to add edge (d, e) and remove another edge in the cycle formed, which is (e, a, c, d, e). If we remove edge (a, e), we decrease the cost of the tree from 20 to 19. That transformation can be made, leaving the tree of Fig. 10.23(b), to which we again try to apply an improving transformation. One such is to insert edge (a, d) and delete edge (c, d) from the cycle formed. The result is shown in Fig. 10.23(c). Then we might introduce (a, b) and delete (b, c) as in Fig. 10.23(d), and subsequently introduce (b, e) in favor of (d, e). The resulting tree of Fig. 10.23(e) is minimal. We can check that every edge not in that tree has the highest cost of any edge in the cycle it forms. Thus no transformation is applicable to Fig. 10.23(e). □

The time taken by the algorithm of Example 10.11 on a graph of n nodes and e edges depends on the number of times we need to improve the solution. Just testing that no transformation is applicable could take $O(ne)$ time, since e edges must be tried, and each could form a cycle of length nearly n. Thus the algorithm is not as good as Prim's or Kruskal's algorithms, but serves as an example where an optimal solution can be obtained by local search.

Local Search Approximation Algorithms

Local search algorithms have had their greatest effectiveness as heuristics for the solution to problems whose exact solutions require exponential time. A common method of search is to start with a number of random solutions, and apply the local transformations to each, until reaching a *locally optimal* solu-tion, one that no transformation can improve. We shall frequently reach dif-ferent locally optimal solutions, from most or all of the random starting solu-tions, as suggested in Fig. 10.24. If we are lucky, one of them will be
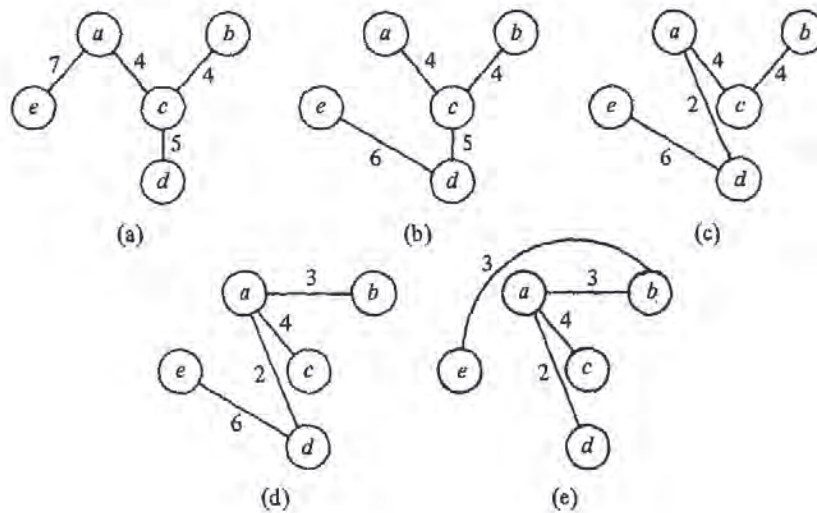
113

Fig. 10.23. Local search for a minimal spanning tree.

globally optimal, that is, as good as any solution.

In practice, we may not find a globally optimal solution as suggested in Fig. 10.24, since the number of locally optimal solutions may be enormous. However, we may at least choose that locally optimal solution that has the least cost among all those found. As the number of kinds of local transformations that have been used to solve various problems is great, we shall close the section with two examples — the TSP, and a simple problem of "package placement."

The Traveling Salesman Problem

The TSP is one for which local search techniques have been remarkably successful. The simplest transformation that has been used is called "2-opting." It consists of taking any two edges, such as (A, B) and (C, D) in Fig. 10.25, removing them, and reconnecting their endpoints to form a new tour. In Fig. 10.25, the new tour runs from B, clockwise to C, then along the edge (C, A), counterclockwise from A to D, and finally along the edge (D, B). If the sum of the lengths of (A, C) and (B, D) is less than the sum of the lengths of (A, B) and (C, D), then we have an improved tour.† Note that we cannot

---

† Do not be fooled by the picture of Fig. 10.25. True, if lengths of edges are distances in the plane, then the dashed edges in Fig. 10.25 must be longer than those they replace. In
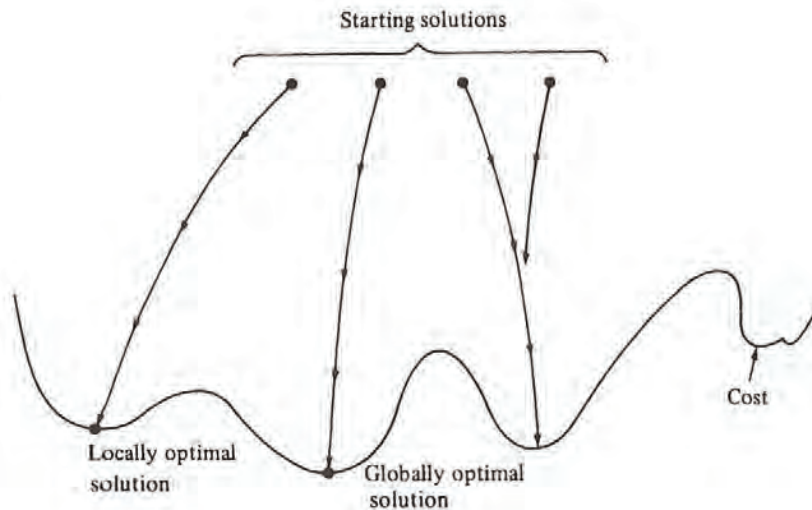
Fig. 10.24. Local search in the space of solutions.

connect A to D and B to C, as the result would not be a tour, but two disjoint cycles.

To find a locally optimal tour, we start with a random tour, and consider all pairs of nonadjacent edges, such as (A, B) and (C, D) in Fig. 10.25. If the tour can be improved by replacing these edges with (A, C) and (B, D), do so, and continue considering pairs of edges that have not been considered before. Note that the introduced edges (A, C) and (B, D) must each be paired with all the other edges of the tour, as additional improvements could result.

Example 10.12. Reconsider Fig. 10.21, and suppose we start with the tour of Fig. 10.26(a). We might replace (a, e) and (c, d), with a total cost of 12, by (a, d) and (c, e), with a total cost of 10, as shown in Fig. 10.26(b). Then we might replace (a, b) and (c, e) by (a, c) and (b, e), giving the optimal tour shown in Fig. 10.26(c). One can check that no pair of edges can be removed from Fig. 10.26(c) and be profitably replaced by crossing edges with the same endpoints. As one case in point, (b, c) and (d, e) together have the relatively high cost of 10. But (c, e) and (b, d) are worse, costing 14 together. □

---

general, however, there is no reason to assume the distances in Fig. 10.25 are distances in the plane, or if they are, it could have been (A, B) and (C, D) that crossed, not (A, C) and (B, D).
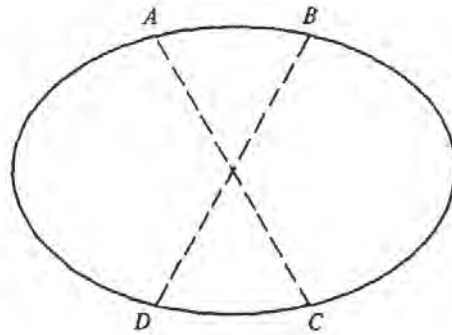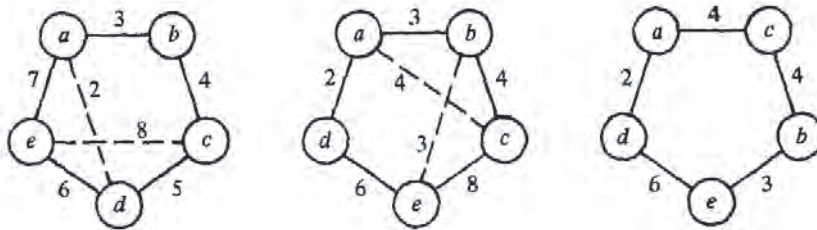
115

Fig. 10.25. 2-opting.

Fig. 10.26. Optimizing a TSP instance by 2-opting.

We can generalize 2-opting to k-opting for any constant k, where we remove up to k edges and reconnect the remaining pieces in any order so that result is a tour. Note that we do not require the removed edges to be nonadjacent in general, although for the 2-opting case there was no point in considering the removal of two adjacent edges. Also note that for $k > 2$, there is more than one way to connect the pieces. For example, Fig. 10.27 shows the general process of 3-opting using any of the following eight sets of edges.

116

1. $(A, F), (D, E), (B, C)$ (as the tour was)
2. $(A, F), (C, E), (D, B)$ (a 2-opt)
3. $(A, E), (F, D), (B, C)$ (another 2-opt)
4. $(A, E), (F, C), (B, D)$ (a true 3-opt)
5. $(A, D), (C, E), (B, F)$ (another true 3-opt)
6. $(A, D), (C, F), (B, E)$ (another true 3-opt)
7. $(A, C), (D, E), (B, F)$ (a 2-opt)
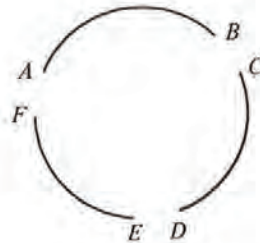8. $(A, C), (D, F), (B, E)$ (a 3-opt)

Fig. 10.27. Pieces of a tour after removing three edges.

It is easy to check that, for fixed k, the number of different k-opting transformations we need to consider if there are n vertices is $O(n^k)$. For example, the exact number is $n(n-3)/2$ for $k = 2$. The time taken to obtain a locally optimal tour may be considerably higher than this, however, since we could make many local transformations before reaching a locally optimum tour, and each improving transformation introduces new edges that may participate in later transformations that improve the tour still further. Lin and Kernighan [1973] have found that variable-depth-opting is in practice a very powerful method and has a good chance of getting the optimum tour on 40-100 city problems.

Package Placement

The *one-dimensional package placement* problem can be stated as follows. We have an undirected graph, whose vertices we call "packages." The edges are labeled by "weights," and the weight $w(a, b)$ of edge (a, b) is the number of "wires" between packages a and b. The problem is to order the vertices $p_1, p_2, \dots, p_n$, such that the sum of $|i-j| \, w(p_i, p_j)$ over all pairs i and j is minimized. That is, we want to minimize the sum of the lengths of the wires needed to connect all the packages with the required number of wires.

The package placement problem has had a number of applications. For

117

example, the "packages" could be logic cards on a rack, and the weight of an interconnection between cards is the number of wires connecting them. A similar problem comes up when we try to design integrated circuits from arrays of standard modules and interconnections between them. A generalization of the one-dimensional package placement problem allows placement of "packages," which have height and width, in a two-dimensional region, while minimizing the sum of the lengths of the wires between packages. This problem also has application to the design of integrated circuits, among other areas.

There are a number of local transformations we could use to find local optima for instances of the one-dimensional package placement problem. Here are several.

1. Interchange adjacent packages p_i and p_{i+1} if the resulting order is less costly. Let $L(j)$ be the sum of the weights of the edges extending to the left of p_j, i.e., $\sum_{k=1}^{j-1} w(p_k, p_j)$. Similarly, let $R(j)$ be $\sum_{k=j+1}^{n} w(p_k, p_j)$. Improvement results if $L(i) - R(i) + R(i+1) - L(i+1) + 2w(p_i, p_{i+1})$ is negative. The reader should verify this formula by computing the costs before and after the interchange and taking the difference.

2. Take a package p_i and insert it between p_j and p_{j+1} for some i and j.

3. Interchange any two packages p_i and p_j.

Example 10.13. Suppose we take the graph of Fig. 10.21 to represent a package placement instance. We shall restrict ourselves to the simple transformation set (1). An initial placement, a, b, c, d, e, is shown in Fig. 10.28(a); it has a cost of 97. Note that the cost function weights edges by their distance, so (a, e) contributes $4 \times 7 = 28$ to the cost of 97. Let us consider interchanging d with e. We have $L(d) = 13$, $R(d) = 6$, $L(e) = 24$, and $R(e) = 0$. Thus $L(d) - R(d) + R(e) - L(e) + 2w(d, e) = -5$, and we can interchange d and e to improve the placement to (a, b, c, e, d), with a cost of 92 as shown in Fig. 10.28(b).

In Fig. 10.28(b), we can interchange c with e profitably, producing Fig. 10.28(c), whose placement (a, b, e, c, d) has a cost of 91. Fig. 10.28(c) is locally optimal for set of transformations (1). It is not globally optimal; (a, c, e, d, b) has a cost of 84. □

As with the TSP, we cannot estimate closely the time it takes to reach a local optimum. We can only observe that for set of transformations (1), there are only $n-1$ transformations to consider. Further, if we compute $L(i)$ and $R(i)$ once, we only have to change them when p_i is interchanged with p_{i-1} or p_{i+1}. Moreover, the recalculation is easy. If p_i and p_{i+1} are interchanged, for example, then the new $L(i)$ and $R(i)$ are, respectively, $L(i+1) - w(p_i, p_{i+1})$ and $R(i+1) + w(p_i, p_{i+1})$. Thus $O(n)$ time suffices to test for an improving transformation and to recalculate the $L(i)$'s and $R(i)$'s. We also need only $O(n)$ time to initialize the $L(i)$'s and $R(i)$'s, if we use the recurrence
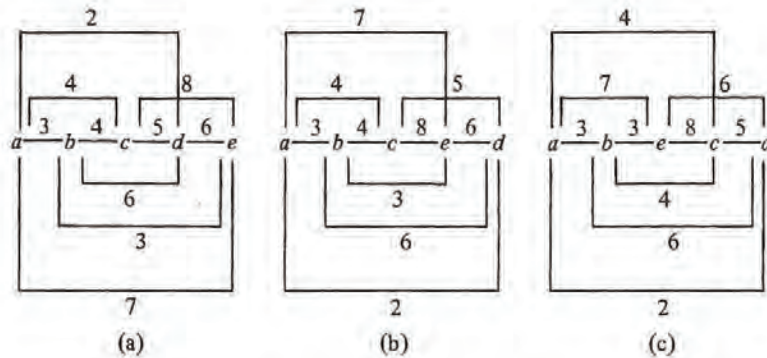
118

Fig. 10.28. Local optimizations.

$$L(1) = 0$$

$$L(i) = L(i-1) + w(p_{i-1}, p_i)$$

and a similar recurrence for R.

In comparison, the sets of transformations (2) and (3) each have $O(n^2)$ members. It will therefore take $O(n^2)$ time just to confirm that we have a locally optimal solution. However, as for set (1), we cannot closely bound the total time taken when a sequence of improvements are made, since each improvement can create additional opportunities for improvement. □

Exercises

10.1 How many moves do the algorithms for moving n disks in the Towers of Hanoi problem take?

*10.2 Prove that the recursive (divide and conquer) algorithm for the Tower of Hanoi and the simple nonrecursive algorithm described at the beginning of Section 10.1 perform the same steps.

10.3 Show the actions of the divide-and-conquer integer multiplication algorithm of Fig. 10.3 when multiplying 1011 by 1101.

*10.4 Generalize the tennis tournament construction of Section 10.1 to tournaments where the number of players is not a power of two. *Hint.* If the number of players n is odd, then one player must receive a *bye*

119

(not play) on any given day, and it will take n days, rather than $n-1$ to complete the tournament. However, if there are two groups with an odd number of players, then the players receiving the bye from each group may actually play each other.

10.5 We can recursively define the number of combinations of m things out of n, denoted $\binom{n}{m}$, for $n \geq 1$ and $0 \leq m \leq n$, by

$$\binom{n}{m} = 1 \quad \text{if } m = 0 \text{ or } m = n$$

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} \quad \text{if } 0 < m < n$$

   a) Give a recursive function to compute $\binom{n}{m}$.

   b) What is its worst-case running time as a function of n?

   c) Give a dynamic programming algorithm to compute $\binom{n}{m}$. *Hint.* The algorithm constructs a table generally known as Pascal's triangle.

   d) What is the running time of your answer to (c) as a function of n.

10.6 One way to compute the number of combinations of m things out of n is to calculate $(n)(n-1)(n-2) \cdots (n-m+1)/(1)(2) \cdots (m)$.

   a) What is the worst case running time of this algorithm as a function of n?

   *b) Is it possible to compute the "World Series Odds" function $P(i, j)$ from Section 10.2 in a similar manner? How fast can you perform this computation?

10.7 a) Rewrite the odds calculation of Fig. 10.7 to take into account the fact that the first team has a probability p of winning any given game.

   b) If the Dodgers have won one game and the Yankees two, but the Dodgers have a .6 probability of winning any given game, who is more likely to win the World Series?

10.8 The odds calculation of Fig. 10.7 requires $O(n^2)$ space. Rewrite the algorithm to use only $O(n)$ space.

*10.9 Prove that Equation (10.4) results in exactly $2\binom{i+j}{i}-1$ calls to P.

10.10 Find a minimal triangulation for a regular octagon, assuming distances are Euclidean.

10.11 The *paragraphing problem*, in a very simple form, can be stated as follows: We are given a sequence of words $w_1, w_2, \ldots, w_k$ of lengths $l_1, l_2, \ldots, l_k$, which we wish to break into lines of length L. Words are separated by blanks whose ideal width is b, but blanks can stretch or shrink if necessary (but without overlapping words), so that a line $w_i\ w_{i+1} \cdots w_j$ has length exactly L. However, the *penalty*

for stretching or shrinking is the magnitude of the total amount by which blanks are stretched or shrunk. That is, the cost of setting line $w_i \ w_{i+1} \ \cdots \ w_j$ for $j > i$ is $(j-i) \ |b'-b|$, where b', the actual width of the blanks, is $(L - l_i - l_{i+1} - \cdots - l_j)/(j-i)$. However, if $j = k$ (we have the last line), the cost is zero unless $b' < b$, since we do not have to stretch the last line. Give a dynamic programming algorithm to find a least-cost separation of $w_1, w_2, \ldots, w_k$ into lines of length L. *Hint.* For $i = k, k-1, \ldots, 1$, compute the least cost of setting $w_i, w_{i+1}, \ldots, w_k$.

10.12 Suppose we are given n elements $x_1, x_2, \ldots, x_n$ related by a linear order $x_1 < x_2 < \cdots < x_n$, and that we wish to arrange these elements m in a binary search tree. Suppose that p_i is the probability that a request to find an element will be for x_i. Then for any given binary search tree, the average cost of a lookup is $\sum_{i=1}^{n} p_i(d_i + 1)$, where d_i is the depth of the node holding x_i. Given the p_i's, and assuming the x_i's never change, we can find a binary search tree that minimizes the lookup cost. Find a dynamic programming algorithm to do so. What is the running time of your algorithm? *Hint.* Compute for all i and j the optimal lookup cost among all trees containing only $x_i, x_{i+1}, \ldots, x_{i+j-1}$, that is, the j elements beginning with x_i.

10.13 For what values of coins does the greedy change-making algorithm of Section 10.3 produce an optimal solution?

10.14 a) Write the recursive triangulation algorithm discussed in Section 10.2.

b) Show that the recursive algorithm results in exactly 3^{s-4} calls on nontrivial problems when started on a problem of size $s \geq 4$.

10.15 Describe a greedy algorithm for

a) The one-dimensional package placement problem.

b) The paragraphing problem (Exercise 10.11).

Give an example where your algorithm does not produce an optimal answer, or show that no such example exists.

10.16 Give a nonrecursive version of the tree search algorithm of Fig. 10.17.

10.17 Consider a game tree in which there are six marbles, and players 1 and 2 take turns picking from one to three marbles. The player who takes the last marble *loses* the game.

a) Draw the complete game tree for this game.

b) If the game tree were searched using the alpha-beta pruning technique, and nodes representing configurations with the smallest number of marbles are searched first, which nodes are pruned?

c) Who wins the game if both play their best?

*10.18 Develop a branch and bound algorithm for the TSP based on the idea that we shall begin a tour at vertex 1, and at each level, branch based on what node comes next in the tour (rather than on whether a particular edge is chosen as in Fig. 10.22). What is an appropriate lower bound estimator for configurations, which are lists of vertices $1, v_1, v_2, \ldots$ that begin a tour? How does your algorithm behave on Fig. 10.21, assuming a is vertex 1?

*10.19 A possible local search algorithm for the paragraphing problem is to allow local transformations that move the first word of one line to the previous line or the last word of a line to the line following. Is this algorithm locally optimal, in the sense that every locally optimal solution is a globally optimal solution?

10.20 If our local transformations consist of 2-opts only, are there any locally optimal tours in Fig. 10.21 that are not globally optimal?

Bibliographic Notes

There are many important examples of divide-and-conquer algorithms including the $O(n\log n)$ Fast Fourier Transform of Cooley and Tukey [1965], the $O(n\log n\log\log n)$ integer multiplication algorithm of Schonhage and Strassen [1971], and the $O(n^{2.81})$ matrix multiplication algorithm of Strassen [1969]. The $O(n^{1.59})$ integer multiplication algorithm is from Karatsuba and Ofman [1962]. Moenck and Borodin [1972] develop several efficient divide-and-conquer algorithms for modular arithmetic and polynomial interpolation and evaluation.

Dynamic programming was popularized by Bellman [1957]. The application of dynamic programming to triangulation is due to Fuchs, Kedem, and Uselton [1977]. Exercise 10.11 is from Knuth [1981]. Knuth [1971] contains a solution to the optimal binary search tree problem in Exercise 10.12.

Lin and Kernighan [1973] describe an effective heuristic for the traveling salesman problem.

See Aho, Hopcroft, and Ullman [1974] and Garey and Johnson [1979] for a discussion of NP-complete and other computationally difficult problems.