CHAPTER 5

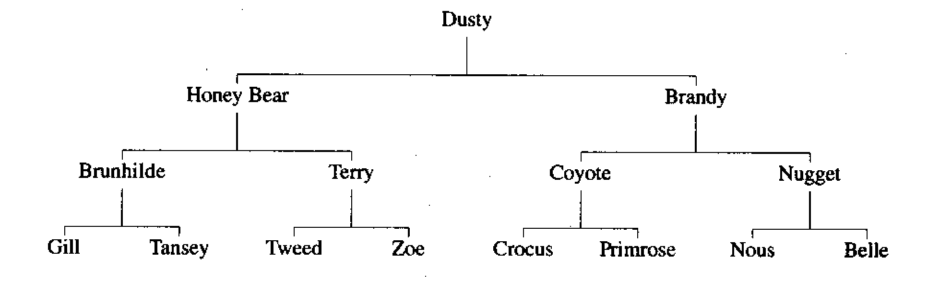
Trees

5.1 INTRODUCTION

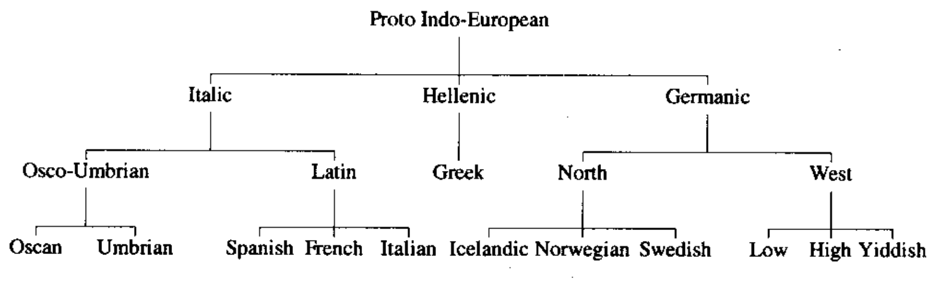
5.1.1 Terminology

In this chapter we shall study a very important data object, the tree. Intuitively, a tree structure means that the data are organized in a hierarchical manner. One very common place where such a structure arises is in the investigation of genealogies. There are two types of genealogical charts that are used to present such data: the *pedigree* and the *lineal* chart. Figure 5.1 gives an example of each.

The pedigree chart of Figure 5.1(a) shows someone's ancestors, in this case those of Dusty, whose two parents are Honey Bear and Brandy. Brandy's parents are Coyote and Nugget, who are Dusty's grandparents on her father's side. The chart continues one more generation back to the great-grandparents. By the nature of things, we know that the pedigree chart is normally two-way branching, though this does not allow for inbreeding. When inbreeding occurs, we no longer have a tree structure unless we insist that each occurrence of breeding is separately listed. Inbreeding may occur frequently when describing family histories of flowers or animals.



(a) Pedigree



(b) Lineal

Figure 5.1: Two types of genealogical charts

The lineal chart of Figure 5.1(b), though it has nothing to do with people, is still a genealogy. It describes, in somewhat abbreviated form, the ancestry of the modern European languages. Thus, this is a chart of descendants rather than ancestors, and each item can produce several others. Latin, for instance, is the forebear of Spanish, French, and Italian. Proto Indo-European is a prehistoric language presumed to have existed in the fifth millenium B.C. This tree does not have the regular structure of the pedigree chart, but it is a tree structure nevertheless.

With these two examples as motivation, let us define formally what we mean by a tree.

Definition: A *tree* is a finite set of one or more nodes such that

(1) There is a specially designated node called the *root*.

(2) The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \cdots, T_n$, where each of these sets is a tree. $T_1, \cdots, T_n$ are called the *subtrees* of the root. □

Notice that this is a recursive definition. If we return to Figure 5.1, we see that the roots of the trees are Dusty and Proto Indo-European. Tree (a) has two subtrees, whose roots are Honey Bear and Brandy; tree (b) has three subtrees, with roots Italic, Hellenic, and Germanic. The condition that $T_1, \cdots, T_n$ be disjoint sets prohibits subtrees from ever connecting together (i.e., there is no cross-breeding). It follows that every item in a tree is the root of some subtree of the whole. For instance, Osco-Umbrian is the root of a subtree of Italic, which itself has two subtrees with the roots Oscan and Umbrian. Umbrian is the root of a tree with no subtrees.

There are many terms that are often used when referring to trees. A *node* stands for the item of information plus the branches to other nodes. Consider the tree in Figure 5.2. This tree has 13 nodes, each item of data being a single letter for convenience. The root is A, and we will normally draw trees with the root at the top.

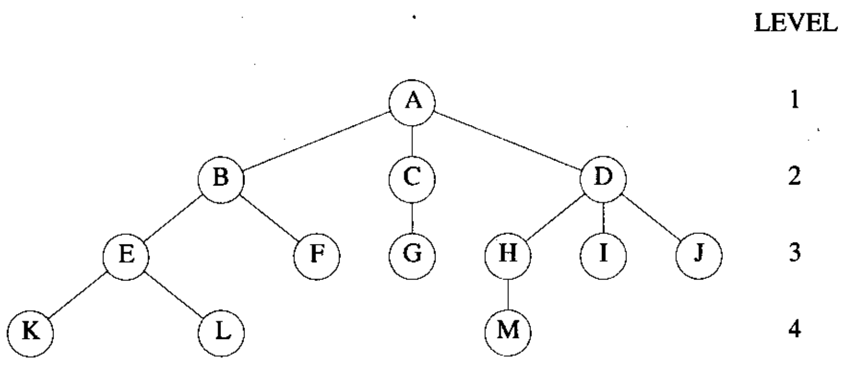


Figure 5.2: A sample tree

The number of subtrees of a node is called its *degree*. The degree of A is 3, of C is 1, and of F is zero. Nodes that have degree zero are called *leaf* or *terminal* nodes. $\{K,L,F,G,M,I,J\}$ is the set of leaf nodes. Consequently, the other nodes are referred to as *nonterminals*. The roots of the subtrees of a node X are the *children* of X. X is the *parent* of its children. Thus, the children of D are H, I, and J; the parent of D is A. Children of the same parent are said to be *siblings*. H, I, and J are siblings. We can extend this terminology if we need to so that we can ask for the grandparent of M, which is D, and so on. The *degree of a tree* is the maximum of the degree of the nodes in the tree. The tree of Figure 5.2 has degree 3. The *ancestors* of a node are all the nodes along the path from the root to that node. The ancestors of M are A, D, and H.

The *level* of a node is defined by letting the root be at level one$^+$. If a node is at level l, then its children are at level $l + 1$. Figure 5.2 shows the levels of all nodes in that tree. The *height* or *depth* of a tree is defined to be the maximum level of any node in the tree. Thus, the depth of the tree in Figure 5.2 is 4.

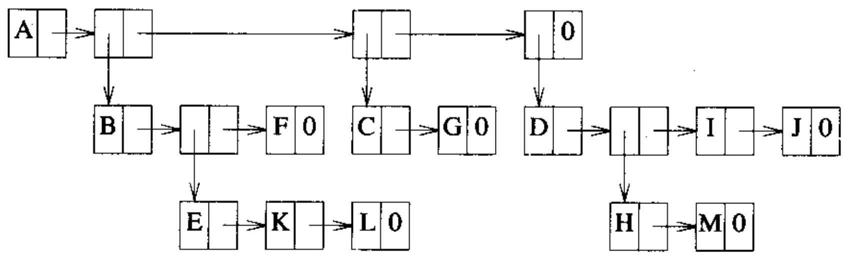
5.1.2 Representation of Trees

5.1.2.1 List Representation

There are several ways to draw a tree besides the one presented in Figure 5.2. One useful way is as a list. The tree of Figure 5.2 could be written as the list

$$(A\,(B\,(E\,(K,L),F),C\,(G),D\,(H\,(M),I,J)))$$

The information in the root node comes first, followed by a list of the subtrees of that node. Figure 5.3 shows the resulting memory representation for the tree of Figure 5.2. If we use this representation, we can make use of many of the general functions that we originally wrote for handling lists.



tag fields not shown

Figure 5.3: List representation of the tree of Figure 5.2

For several applications it is desirable to have a representation that is specialized to trees. One possibility is to represent each tree node by a memory node that has fields for the data and pointers to the tree node's children. Since the degree of each tree node may be different, we may be tempted to use memory nodes with a varying number of pointer fields. However, as it is often easier to write algorithms for a data representation when the node size is fixed, in practice one uses only nodes of a fixed size to represent tree nodes. For a tree of degree k, we could use the node structure of Figure 5.4. Each child field is used to point to a subtree. Lemma 5.1 shows that using this node structure is very wasteful of space.

DATA	CHILD 1	CHILD 2	···	CHILD k

Figure 5.4: Possible node structure for a tree of degree k

Lemma 5.1: If T is a k-ary tree (i.e., a tree of degree k) with n nodes, each having a fixed size as in Figure 5.4, then $n(k-1)+1$ of the nk child fields are 0, $n \geq 1$.

Proof: Since each non-zero child field points to a node and there is exactly one pointer to each node other than the root, the number of non-zero child fields in an n-node tree is exactly $n-1$. The total number of child fields in a k-ary tree with n nodes is nk. Hence, the number of zero fields is $nk-(n-1)=n(k-1)+1$. □

We shall develop two specialized fixed-node-size representations for trees. Both of these require exactly two link, or pointer, fields per node.

5.1.2.2 Left Child-Right Sibling Representation

Figure 5.5 shows the node structure used in the left child–right sibling representation.

data	
left child	right sibling

Figure 5.5: Left child-right sibling node structure

To convert the tree of Figure 5.2 into this representation, we first note that every node has at most one leftmost child and at most one closest right sibling. For example, in Figure 5.2, the leftmost child of A is B, and the leftmost child of D is H. The closest right sibling of B is C, and the closest right sibling of H is I. Strictly speaking, since the order of children in a tree is not important, any of the children of a node could be the leftmost child, and any of its siblings could be the closest right sibling. For the sake of definiteness, we choose the nodes based on how the tree is drawn. The *left child* field of each node points to its leftmost child (if any), and the *right sibling* field points to its closest right sibling (if any). Figure 5.6 shows the tree of Figure 5.2 redrawn using the left child-right sibling representation.

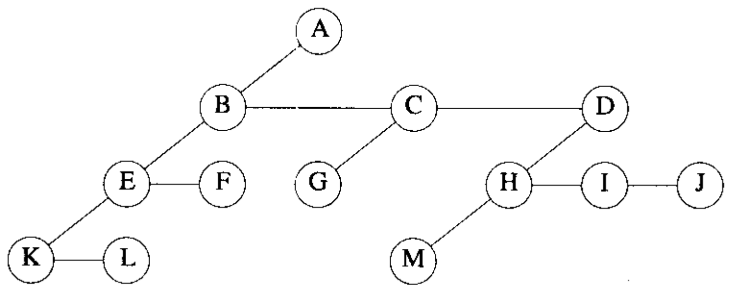


Figure 5.6: Left child-right sibling representation of tree of Figure 5.2

5.1.2.3 Representation as a Degree-Two Tree

To obtain the degree-two tree representation of a tree, we simply rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45 degrees. This gives us the degree-two tree displayed in Figure 5.7. In the degree-two representation, we refer to the two children of a node as the left and right children. Notice that the right child of the root node of the tree is empty. This is always the case since the root of the tree we are transforming can never have a sibling. Figure 5.8 shows two additional examples of trees represented as left child-right sibling trees and as left child-right child (or degree-two) trees. Left child-right child trees are also known as *binary trees*.

EXERCISES

1. Write a function to input a tree given as a generalized list (e.g., $(A(B(E(K,L),F),C(G),D(H(M),I,J)))$ and create its internal representation using nodes with three fields: *tag*, *data*, and *link*.
2. Write a function that reverses the process in Exercise 1 and takes a pointer to a tree and outputs it as a generalized list.
3. [***Programming Project***] Write the Write the following C functions.
   (a) [read]: accept a tree represented as a parenthesized list as input and create the generalized list representation of the tree (see Figure 5.3)

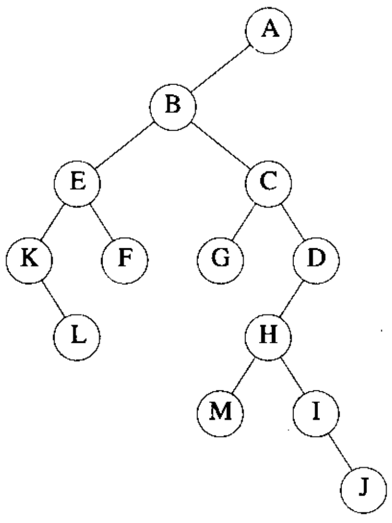


Figure 5.7: Left child-right child tree representation of tree of Figure 5.2

(b) [copy]: make a copy of a tree represented as a generalized list

(c) [isequal]: test for equality between two trees represented as generalized lists

(d) [clear]: delete a tree represented as a generalized list

(e) [write]: output a tree in its parenthesized list notation

Test the correctness of your functions using suitable test data.

5.2 BINARY TREES

5.2.1 The Abstract Data Type

We have seen that we can represent any tree as a binary tree. In fact, binary trees are an important type of tree structure that occurs very often. The chief characteristic of a binary tree is the stipulation that the degree of any given node must not exceed two. For binary trees, we also distinguish between the left subtree and the right subtree, while for trees the order of the subtrees is irrelevant. In addition, a binary tree may have zero nodes. Thus, a binary tree is really a different object than a tree.

Definition: A *binarv tree* is a finite set of nodes that is either emotv or consists of a root

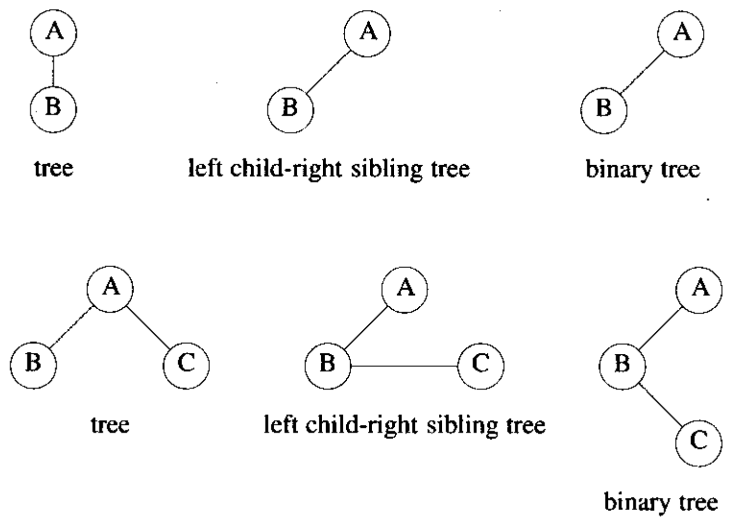


Figure 5.8: Tree representations

and two disjoint binary trees called the left subtree and the right subtree. □

ADT 5.1 contains the specification for the binary tree ADT. This structure defines only a minimal set of operations on binary trees which we use as a foundation on which to build additional operations.

Let us carefully review the distinctions between a binary tree and a tree. First, there is no tree having zero nodes, but there is an empty binary tree. Second, in a binary tree we distinguish between the order of the children while in a tree we do not. Thus, the two binary trees of Figure 5.9 are different since the first binary tree has an empty right subtree, while the second has an empty left subtree. Viewed as trees, however, they are the same, despite the fact that they are drawn slightly differently.

Figure 5.10 shows two special kinds of binary trees. The first is a *skewed* tree, skewed to the left, and there is a corresponding tree that skews to the right. The tree of Figure 5.10(b) is called a *complete* binary tree. This kind of binary tree will be defined formally later. Notice that all leaf nodes are on adjacent levels. The terms that we introduced for trees such as degree, level, height, leaf, parent, and child all apply to binary trees in the natural way.

ADT *Binary_Tree* (abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

functions:

for all $bt, bt1, bt2 \in$ *BinTree*, *item* $\in$ *element*

BinTree Create()	::=	creates an empty binary tree
Boolean IsEmpty(*bt*)	::=	**if** (*bt* == empty binary tree) **return** *TRUE* **else return** *FALSE*
BinTree MakeBT(*bt*1, *item*, *bt*2)	::=	**return** a binary tree whose left subtree is *bt*1, whose right subtree is *bt*2, and whose root node contains the data *item*.
BinTree Lchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the left subtree of *bt*.
element Data(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the data in the root node of *bt*.
BinTree Rchild(*bt*)	::=	**if** (IsEmpty(*bt*)) **return** error **else return** the right subtree of *bt*.

ADT 5.1: Abstract data type *Binary_Tree*

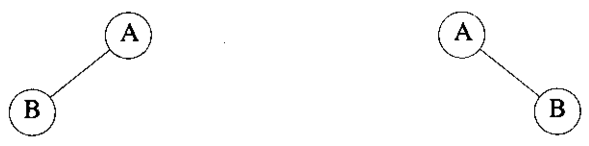


Figure 5.9: Two different binary trees

5.2.2 Properties of Binary Trees

Before examining data representations for binary trees, let us make some observations about such trees. In particular, we want to determine the maximum number of nodes in a binary tree of depth k and the relationship between the number of leaf nodes and the number of degree-two nodes in a binary tree.

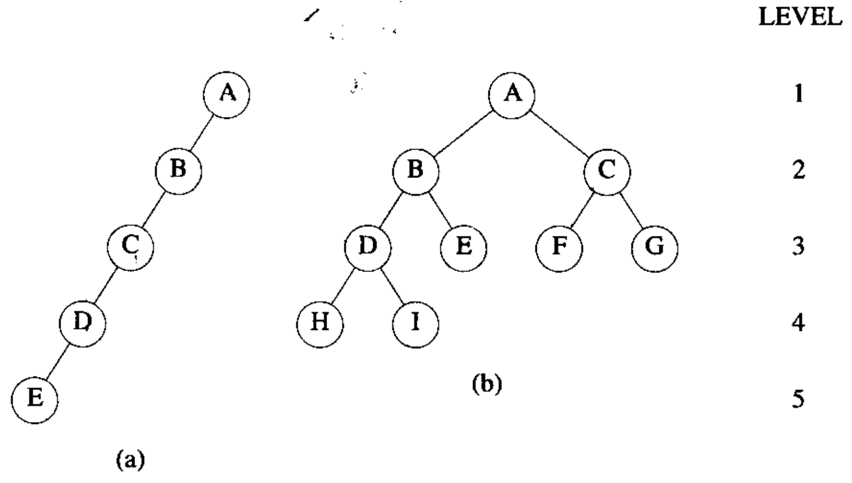


Figure 5.10: Skewed and complete binary trees

Lemma 5.2 [***Maximum number of nodes***]:

(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.

(2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Proof:

(1) The proof is by induction on i.

Induction Base: The root is the only node on level $i = 1$. Hence, the maximum number of nodes on level $i = 1$ is $2^{i-1} = 2^0 = 1$.

Induction Hypothesis: Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level $i - 1$ is 2^{i-2}.

Induction Step: The maximum number of nodes on level $i - 1$ is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level $i-1$, or 2^{i-1}.

(2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=1}^{k} (\text{maximum number of nodes on level } i) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1 \square$$

Lemma 5.3 [*Relation between number of leaf nodes and degree-2 nodes*]: For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof: Let n_1 be the number of nodes of degree one and n the total number of nodes. Since all nodes in T are at most of degree two, we have

$$n = n_0 + n_1 + n_2 \qquad (5.1)$$

If we count the number of branches in a binary tree, we see that every node except the root has a branch leading into it. If B is the number of branches, then $n = B+1$. All branches stem from a node of degree one or two. Thus, $B = n_1 + 2n_2$. Hence, we obtain

$$n = B + 1 = n_1 + 2n_2 + 1 \qquad (5.2)$$

Subtracting Eq. (5.2) from Eq. (5.1) and rearranging terms, we get

$$n_0 = n_2 + 1 \quad \square$$

In Figure 5.10(a), $n_0 = 1$ and $n_2 = 0$; in Figure 5.10(b), $n_0 = 5$ and $n_2 = 4$.

We are now ready to define full and complete binary trees.

Definition: A *full binary tree* of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$. $\square$

By Lemma 5.2, $2^k - 1$ is the maximum number of nodes in a binary tree of depth k. Figure 5.11 shows a full binary tree of depth 4. Suppose we number the nodes in a full binary tree starting with the root on level 1, continuing with the nodes on level 2, and so on. Nodes on any level are numbered from left to right. This numbering scheme gives us the definition of a complete binary tree.

Definition: A binary tree with n nodes and depth k is *complete* iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k. $\square$

From Lemma 5.2, it follows that the height of a complete binary tree with n nodes is $\lceil \log_2(n+1) \rceil$. (Note that $\lceil x \rceil$ is the smallest integer $\geq x$.)

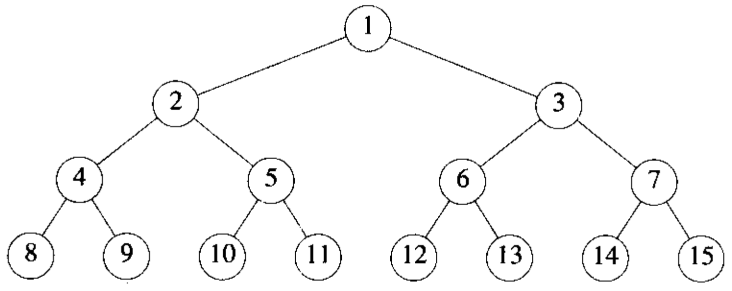


Figure 5.11: Full binary tree of depth 4 with sequential node numbers

5.2.3 Binary Tree Representations

5.2.3.1 Array Representation

The numbering scheme used in Figure 5.11 suggests our first representation of a binary tree in memory. Since the nodes are numbered from 1 to n, we can use a one-dimensional array to store the nodes. Position 0 of this array is left empty and the node numbered i in Figure Figure 5.11 is mapped to position i of the array. Using Lemma 5.4 we can easily determine the locations of the parent, left child, and right child of any node, i, in the binary tree.

Lemma 5.4: If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \leq i \leq n$, we have

(1) *parent* (i) is at $\lfloor i / 2 \rfloor$ if $i \neq 1$. If $i = 1$, i is at the root and has no parent.

(2) *leftChild* (i) is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.

(3) *rightChild* (i) is at $2i + 1$ if $2i + 1 \leq n$. If $2i + 1 > n$, then i has no right child.

Proof: We prove (2). (3) is an immediate consequence of (2) and the numbering of nodes on the same level from left to right. (1) follows from (2) and (3). We prove (2) by induction on i. For $i = 1$, clearly the left child is at 2 unless $2 > n$, in which case i has no left child. Now assume that for all j, $1 \leq j \leq i$, *leftChild* (j) is at $2j$. Then the two nodes immediately preceding *leftChild* ($i+1$) are the right and left children of i. The left child is at $2i$. Hence, the left child of $i + 1$ is at $2i + 2 = 2(i + 1)$ unless $2(i + 1) > n$, in which case $i + 1$ has no left child. □

This representation can clearly be used for all binary trees, though in most cases there will be a lot of unutilized space. Figure 5.12 shows the array representation for both trees of Figure 5.10. For complete binary trees such as the one in Figure 5.10(b), the representation is ideal, as no space is wasted. For the skewed tree of Figure 5.10(a), however, less than half the array is utilized. In the worst case a skewed tree of depth k will require 2^k-1 spaces. Of these, only k will be used.

	tree
[0]	–
[1]	A
[2]	B
[3]	–
[4]	C
[5]	–
[6]	–
[7]	–
[8]	D
[9]	–
.	.
.	.
.	.
[16]	E

(a) Tree of Figure 5.10(a)

	tree
[0]	–
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

(b) Tree of Figure 5.10(b)

Figure 5.12: Array representation of the binary trees of Figure 5.10

5.2.3.2 Linked Representation

Although the array representation is good for complete binary trees, it is wasteful for many other binary trees. In addition, the representation suffers from the general inadequacies of sequential representations. Insertion and deletion of nodes from the middle of a tree require the movement of potentially many nodes to reflect the change in level number of these nodes. These problems can be overcome easily through the use of a linked representation. Each node has three fields, *leftChild*, *data*, and *rightChild*, and is defined in C as:

```
        typedef struct node *treePointer;
        typedef struct {
                int data;
                treePointer leftChild, rightChild;
                } node;
```

We shall draw a tree node using either of the representations of Figure 5.13.

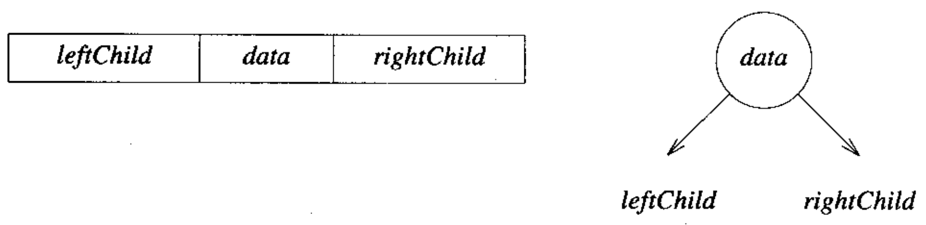


Figure 5.13: Node representations

Although with this node structure it is difficult to determine the parent of a node, we shall see that for most applications, this node structure is adequate. If it is necessary to be able to determine the parent of random nodes, then a fourth field, *parent*, may be included in the class *TreeNode*. The representation of the binary trees of Figure 5.10 using this node structure is given in Figure 5.14. The root of the tree is stored in the data member *root* of *Tree*. This data member serves as the access pointer to the tree.

EXERCISES

1. For the binary tree of Figure 5.15, list the leaf nodes, the nonleaf nodes, and the level of each node.
2. What is the maximum number of nodes in a k-ary tree of height h? Prove your answer.
3. Draw the internal memory representation of the binary tree of Figure 5.15 using (a) sequential and (b) linked representations.
4. Extend the array representation of a complete binary tree to the case of complete trees whose degree is d, $d>1$. Develop formulas for the parent and children of the node stored in position i of the array.

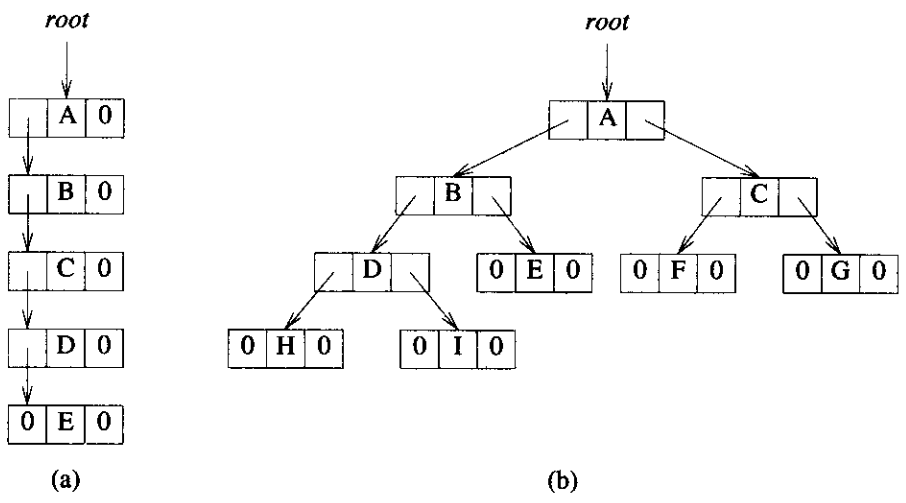


Figure 5.14: Linked representation for the binary trees of Figure 5.10

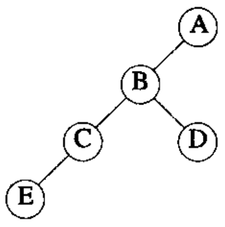


Figure 5.15: Binary tree for Exercise 1

5.3 BINARY TREE TRAVERSALS

There are many operations that we often want to perform on trees. One notion that arises frequently is the idea of traversing a tree or visiting each node in the tree exactly once. When a node is visited, some operation (such as outputting its *data* field) is performed on it. A full traversal produces a linear order for the nodes in a tree. This linear order, given by the order in which the nodes are visited, may be familiar and useful. When traversing a binary tree, we want to treat each node and its subtrees in the same fashion. If we let *L*, *V*, and *R* stand for moving left, visiting the node, and moving right when at a node, then there are six possible combinations of traversal: *LVR*, *LRV*, *VLR*, *VRL*, *RVL*, and *RLV*. If we adopt the convention that we traverse left before right, then only three traversals remain: *LVR*, *LRV*, and *VLR*. To these we assign the names *inorder*, *postorder*, and *preorder*, respectively, because of the position of the *V* with respect to the *L* and the *R*. For example, in postorder, we visit a node after we have traversed its left and right subtrees, whereas in preorder the visiting is done before the traversal of these subtrees.

There is a natural correspondence between these traversals and producing the infix, postfix, and prefix forms of an expression. Consider the binary tree of Figure 5.16. This tree contains an arithmetic expression with the binary operators add (+), multiply (*), and divide (/) and the variables *A*, *B*, *C*, *D*, and *E*. For each node that contains an operator, its left subtree gives the left operand and its right subtree the right operand. We use this tree to illustrate each of the traversals.

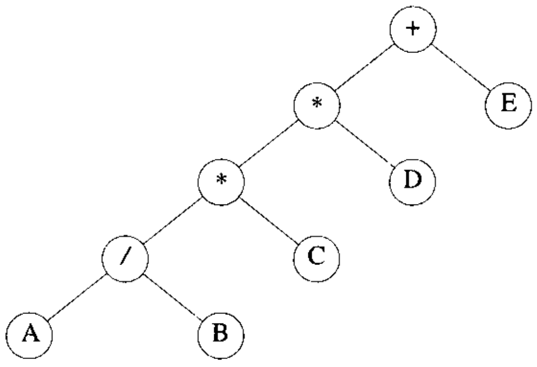


Figure 5.16: Binary tree with arithmetic expression

5.3.1 Inorder Traversal

Informally, *inorder traversal* calls for moving down the tree toward the left until you can go no farther. Then you "visit" the node, move one node to the right and continue. If you cannot move to the right, go back one more node. A precise way of describing this traversal is by using recursion as in Program 5.1.

Recursion is an elegant device for describing this traversal. Figure 5.17 is a trace

```
void inorder(treePointer ptr)
{/* inorder tree traversal */
   if (ptr) {
      inorder(ptr→leftChild);
      printf("%d",ptr→data);
      inorder(ptr→rightChild);
   }
}
```

Program 5.1: Inorder traversal of a binary tree

of *inorder* using the tree of Figure 5.16. Each step of the trace shows the call of *inorder*, the value in the root, and whether or not the **printf** function is invoked. The first three columns show the first 13 steps of the traversal. The second three columns show the remaining 14 steps. The numbers in columns 1 and 4 correspond to the node numbers displayed in Figure 5.16 and are used to show the location of the node in the tree.

Call of	Value				Value
inorder	in *root*	Action	*inorder*	in *root*	Action
1	+		11	*C*	
2	*		12	*NULL*	
3	*		11	*C*	**printf**
4	/		13	*NULL*	
5	*A*		2	*	**printf**
6	*NULL*		14	*D*	
5	*A*	**printf**	15	*NULL*	
7	*NULL*		14	*D*	**printf**
4	/	**printf**	16	*NULL*	
8	*B*		1	+	**printf**
9	*NULL*		17	*E*	
8	*B*	**printf**	18	*NULL*	
10	*NULL*		17	*E*	**printf**
3	*	**printf**	19	*NULL*	

Figure 5.17: Trace of Program 5.1

Since there are 19 nodes in the tree, *inorder* is invoked 19 times for the complete traversal. The data fields are output in the order:

$$A / B * C * D + E$$

which corresponds to the infix form of the expression.

5.3.2 Preorder Traversal

The code for the second form of traversal, *preorder*, is given in Program 5.2. In words, we would say "visit a node, traverse left, and continue. When you cannot continue, move right and begin again or move back until you can move right and resume." The nodes of Figure 5.16 would be output in *pre*order as

$$+ * * / A\ B\ C\ D\ E$$

which we recognize as the *pre*fix form of the expression.

```
void preorder(treePointer ptr)
{/* preorder tree traversal */
   if (ptr) {
      printf("%d",ptr→data);
      preorder(ptr→leftChild);
      preorder(ptr→rightChild);
   }
}
```

Program 5.2: Preorder traversal of a binary tree

5.3.3 Postorder Traversal

The code for *postorder* traversal is given in Program 5.3. On the tree of Figure 5.16, this function produces the following output:

$$A\ B / C * D * E +$$

which is the *post*fix form of our expression.

```
void postorder(treePointer ptr)
{/* postorder tree traversal */
   if (ptr) {
      postorder(ptr→leftChild);
      postorder(ptr→rightChild);
      printf("%d",ptr→data);
   }
}
```

Program 5.3: Postorder traversal of a binary tree

5.3.4 Iterative Inorder Traversal

Although we have written the inorder, preorder, and postorder traversal functions recursively, we can develop equivalent iterative functions. Let us take inorder traversal as an example. To simulate the recursion, we must create our own stack. We add nodes to and remove nodes from our stack in the same manner that the recursive version manipulates the system stack. This helps us to understand fully the operation of the recursive version. Figure 5.17 implicitly shows this stacking and unstacking. A node that has no action indicates that the node is added to the stack, while a node that has a *printf* action indicates that the node is removed from the stack. Notice that the left nodes are stacked until a null node is reached, the node is then removed from the stack, and the node's right child is stacked. The traversal then continues with the left child. The traversal is complete when the stack is empty. Function *iterInorder* (Program 5.4) stems directly from this discussion. The stack function *push* differs from that defined in Chapter 3 only in that the type of the elements in the stack is different. Similarly, the *pop* function returns a value of type *treePointer* rather than of type *element*. It returns *NULL* in case the stack is empty.

Analysis of *iterInorder*: Let n be the number of nodes in the tree. If we consider the action of *iterInorder*, we note that every node of the tree is placed on and removed from the stack exactly once. So, if the number of nodes in the tree is n, the time complexity is $O(n)$. The space requirement is equal to the depth of the tree which is $O(n)$. □

5.3.5 Level-Order Traversal

Whether written iteratively or recursively, the inorder, preorder, and postorder traversals all require a stack. We now turn to a traversal that requires a queue. This traversal, called *level-order traversal*, visits the nodes using the ordering suggested by the node

```
void iterInorder(treePointer node)
{
   int top = -1; /* initialize stack */
   treePointer stack[MAX_STACK_SIZE];
   for (;;) {
      for(; node; node = node→leftChild)
         push(node); /* add to stack */
      node = pop(); /* delete from stack */
      if (!node) break; /* empty stack */
      printf("%d", node→data);
      node = node→rightChild;
   }
}
```

Program 5.4: Iterative inorder traversal

numbering scheme of Figure 5.11. Thus, we visit the root first, then the root's left child, followed by the root's right child. We continue in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node.

The code for this traversal is contained in *levelOrder* (Program 5.5). This assumes a circular queue as in Chapter 3. Function *addq* differs from the corresponding function of Chapter 3 only in that the data type of the elements in the queue is different. Similarly, the function *deleteq* used in Program 5.5 returns a value of type *treePointer* rather than of type element. It returns *NULL* in case the queue is empty.

We begin by adding the root to the queue. The function operates by deleting the node at the front of the queue, printing out the node's data field, and adding the node's left and right children to the queue. Since a node's children are at the next lower level, and we add the left child before the right child, the function prints out the nodes using the ordering scheme found in Figure 5.11. The level order traversal of the tree in Figure 5.16 is:

$$+ * E * D / C A B$$

5.3.6 Traversal without a Stack

Before we leave the topic of tree traversal, we shall consider one final question. Is binary tree traversal possible without the use of extra space for a stack? (Note that a recursive tree traversal algorithm also implicitly uses a stack.) One simple solution is to add a *parent* field to each node. Then we can trace our way back up to any root and

```
void levelOrder(treePointer ptr)
{/* level order tree traversal */
  int front = rear = 0;
  treePointer queue[MAX_QUEUE_SIZE];
  if (!ptr) return; /* empty tree */
  addq(ptr);
  for (;;) {
    ptr = deleteq();
    if (ptr) {
      printf("%d",ptr→data);
      if(ptr→leftChild)
        addq(ptr→leftChild);
      if (ptr→rightChild)
        addq(ptr→rightChild);
    }
    else break;
  }
}
```

Program 5.5: Level-order traversal of a binary tree

down again. Another solution, which requires two bits per node, represents binary trees as threaded binary trees. We study this in Section 5.5. If the allocation of this extra space is too costly, then we can use the *leftChild* and *rightChild* fields to maintain the paths back to the root. The stack of addresses is stored in the leaf nodes.

EXERCISES

1. Write out the inorder, preorder, postorder, and level-order traversals for the binary trees of Figure 5.10.
2. Do Exercise 1 for the binary tree of Figure 5.11.
3. Do Exercise 1 for the binary tree of Figure 5.15.
4. Write a nonrecursive version of function *preorder* (Program 5.2).
5. Write a nonrecursive version of function *postorder* (Program 5.3).
6. Rework *iterInorder* (Program 5.4) so that it is as fast as possible. (Hint: Minimize the stacking and the testing within the loop.)

5.4 ADDITIONAL BINARY TREE OPERATIONS

5.4.1 Copying Binary Trees

By using the definition of a binary tree and the recursive versions of inorder, preorder, and postorder traversals, we can easily create C functions for other binary tree operations. One practical operation is copying a binary tree. The code for this operation is containted in *copy* (Program 5.6). Notice that this function is only a slightly modified version of *postorder* (Program 5.3).

```
treePointer copy(treePointer original)
{/* this function returns a treePointer to an exact copy
    of the original tree */
   treePointer temp;
   if (original) {
      MALLOC(temp, sizeof(*temp));
      temp→leftChild = copy(original→leftChild);
      temp→rightChild = copy(original→rightChild);
      temp→data = original→data;
      return temp;
   }
   return NULL;
}
```

Program 5.6: Copying a binary tree

5.4.2 Testing Equality

Another useful operation is determining the equivalence of two binary trees. Equivalent binary trees have the same structure and the same information in the corresponding nodes. By the same structure we mean that every branch in one tree corresponds to a branch in the second tree, that is, the branching of the two trees is identical. The function *equal* (Program 5.7) uses a modification of preorder traversal to test for equality. This function returns *TRUE* if the two trees are equivalent and *FALSE* if they are not.

```
int equal(treePointer first, treePointer second)
{/* function returns FALSE if the binary trees first and
    second are not equal, Otherwise it returns TRUE */
  return ((!first && !second) || (first && second &&
           (first->data == second->data) &&
           equal(first->leftChild,second->leftChild) &&
           equal(first->rightChild, second->rightChild))
}
```

Program 5.7: Testing for equality of binary trees

5.4.3 The Satisfiability Problem

Consider the set of formulas that we can construct by taking variables $x_1, x_2, \cdots, x_n$ and operators $\wedge$ (*and*), $\vee$ (*or*), and $\neg$ (*not*). The variables can hold only one of two possible values, *true* or *false*. The set of expressions that we can form using these variables and operators is defined by the following rules:

(1) A variable is an expression.

(2) If x and y are expressions, then $\neg x, x \wedge y, x \vee y$ are expressions.

(3) Parentheses can be used to alter the normal order of evaluation, which is $\neg$ before $\wedge$ before $\vee$.

These rules comprise the formulas in the propositional calculus since other operations, such as implication, can be expressed using $\neg$, $\vee$, and $\wedge$.

The expression:

$$x_1 \vee (x_2 \wedge \neg x_3)$$

is a formula (read as "x_1 *or* x_2 *and not* x_3"). If x_1 and x_3 are *false* and x_2 is *true*, then the value of the expression is:

$$\begin{aligned} & \textit{false} \vee (\textit{true} \wedge \neg \textit{false}) \\ &= \textit{false} \vee \textit{true} \\ &= \textit{true} \end{aligned}$$

The satisfiability problem for formulas of the propositional calculus asks if there is an assignment of values to the variables that causes the value of the expression to be true. This problem was originally used by Newell, Shaw, and Simon in the late 1950s to show the viability of heuristic programming (The Logic Theorist) and is still of keen interest to computer scientists.

Again, let us assume that our formula is already in a binary tree, say

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

in the tree of Figure 5.18. The inorder traversal of this tree is

$$x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_3 \vee \neg x_3$$

which is the infix form of the expression. The most obvious algorithm to determine satisfiability is to let (x_1, x_2, x_3) take on all possible combinations of *true* and *false* values and to check the formula for each combination. For n variables there are 2^n possible combinations of *true* = *t* and *false* = *f*. For example, for $n = 3$, the eight combinations are: (t,t,t), (t,t,f), (t,f,t), (t,f,f), (f,t,t), (f,t,f), (f,f,t), (f,f,f). The algorithm will take $O(g\,2^n)$, or exponential time, where g is the time to substitute values for $x_1, x_2, \cdots, x_n$ and evaluate the expression.

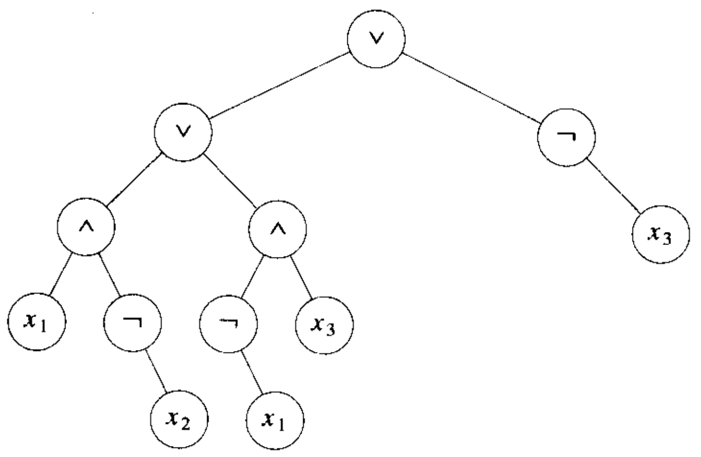


Figure 5.18: Propositional formula in a binary tree

To evaluate an expression, we traverse its tree in postorder. When visiting a node p, we compute the value of the expression represented by the subtree rooted at p. Recall that, in postorder, the left and right subtrees of a node are traversed before we visit that node. In other words, when we visit the node p, the subexpressions represented by its left and right subtrees have been computed. So, when we reach the ∨ node on level 2, the values of $x_1 \wedge \neg x_2$ and $\neg x_1 \wedge x_3$ will already be available to us, and we can apply the rule for **or**. Notice that a node containing $\neg$ has only a right branch, since $\neg$ is a unary operator.

The node structure for this problem is found in Figure 5.19. The *leftChild* and *rightChild* fields are similar to those used previously. The field *data* holds either the value of a variable or a propositional calculus operator, while *value* holds either a value of *TRUE* or *FALSE*.

leftChild	*data*	*value*	*rightChild*

Figure 5.19: Node structure for the satisfiability problem

We define this node structure in C as:

```
typedef enum {not,and,or,true,false} logical;
typedef struct node *treePointer;
typedef struct {
        treePointer leftChild;
        logical     data;
        short int   value;
        treePointer rightChild;
        } node;
```

We assume that for leaf nodes, *node* $\rightarrow$ *data* contains the current value of the variable represented at this node. For example, we assume that the tree of Figure 5.18 contains either *TRUE* or *FALSE* in the data field of x_1, x_2, and x_3. We also assume that an expression tree with n variables is pointed at by *root*. With these assumptions we can write our first version of a satisfiability algorithm (Program 5.8).

The C function that evaluates the tree is easily obtained by modifying the original, recursive postorder traversal. The function *postOrderEval* (Program 5.9) shows the C code that implements this portion of the satisfiability algorithm.

EXERCISES

1. Write a C function that counts the number of leaf nodes in a binary tree. Determine the computing time of the function.
2. Write a C function *swapTree* that takes a binary tree and swaps the left and right children of every node. An example is given in Figure 5.20.

```
for (all $2^n$ possible combinations) {
   generate the next combination;
   replace the variables by their values;
   evaluate root by traversing it in postorder;
   if (root→value) {
      printf(<combination>);
      return;
   }
}
printf("No satisfiable combination\n");
```

Program 5.8: First version of satisfiability algorithm

3. What is the computing time of *postOrderEval*?
4. Devise an external representation for the formulas in propositional calculus. Write a function that reads such a formula and creates its binary tree representation. What is the complexity of your function?
5. § [***Programming project***] Devise a representation for formulas in the propositional calculus. Write a C function that inputs such a formula and creates a binary tree representation of it. Determine the computing time of your function.

5.5 THREADED BINARY TREES

5.5.1 Threads

If we look carefully at the linked representation of any binary tree, we notice that there are more null links than actual pointers. Specifically, there are $n + 1$ null links out of $2n$ total links. A. J. Perlis and C. Thornton have devised a clever way to make use of these null links. They replace the null links by pointers, called *threads*, to other nodes in the tree. To construct the threads we use the following rules (assume that *ptr* represents a node):

(1) If *ptr* → *leftChild* is null, replace *ptr* → *leftChild* with a pointer to the node that would be visited before *ptr* in an inorder traversal. That is we replace the null link with a pointer to the *inorder predecessor* of *ptr*.

(2) If *ptr* → *rightChild* is null, replace *ptr* → *rightChild* with a pointer to the node that would be visited after *ptr* in an inorder traversal. That is we replace the null link with a pointer to the *inorder successor* of *ptr*.

```
void postOrderEval(treePointer node)
{/* modified post order traversal to evaluate a
    propositional calculus tree */
  if (node) {
    postOrderEval(node→leftChild);
    postOrderEval(node→rightChild);
    switch(node→data) {
      case not:   node→value =
            !node→rightChild→value;
            break;
      case and:   node→value =
            node→rightChild→value &&
            node→leftChild→value;
            break;
      case or:    node→value =
            node→rightChild→value ||
            node→leftChild→value;
            break;
      case true:  node→value = TRUE;
            break;
      case false: node→value = FALSE;
    }
  }
}
```

Program 5.9: Postorder evaluation function

Figure 5.21 shows the binary tree of Figure 5.10(b) with its new threads drawn in as broken lines. This tree has 9 nodes and 10 0-links, which have been replaced by threads. If we traverse the tree in inorder, the nodes will be visited in the order *H, D, I, B, E, A, F, C, G*. For example, node *E* has a predecessor thread that points to *B* and a successor thread that points to *A*.

When we represent the tree in memory, we must be able to distinguish between threads and normal pointers. This is done by adding two additional fields to the node structure, *leftThread* and *rightThread*. Assume that *ptr* is an arbitrary node in a threaded tree. If *ptr* → *leftThread* = *TRUE*, then *ptr* → *leftChild* contains a thread; otherwise it contains a pointer to the left child. Similarly, if *ptr* → *rightThread* = *TRUE*, then *ptr* → *rightChild* contains a thread; otherwise it contains a pointer to the right child.

This node structure is given by the following C declarations:

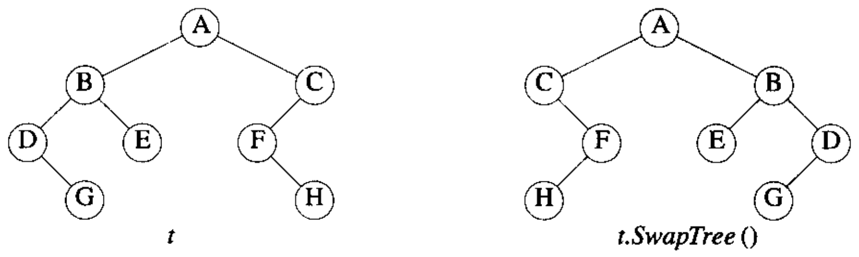


Figure 5.20: A swap tree example

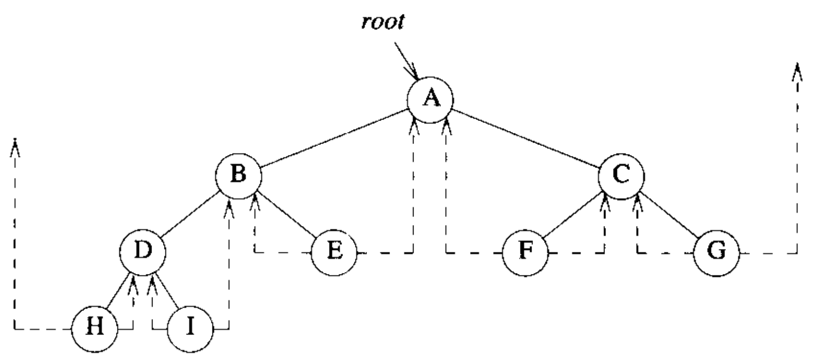


Figure 5.21: Threaded tree corresponding to Figure 5.10(b)

```
typedef struct threadedTree *threadedPointer;
typedef struct {
        short int leftThread;
        threadedPointer leftChild;
        char data;
        threadedPointer rightChild;
        short int rightThread;
        } threadedTree;
```

In Figure 5.21 two threads have been left dangling: one in the left child of H, the other in the right child of G. In order that we leave no loose threads, we will assume a header node for all threaded binary trees. The original tree is the left subtree of the header node. An empty binary tree is represented by its header node as in Figure 5.22. The complete memory representation for the tree of Figure 5.21 is shown in Figure 5.23.

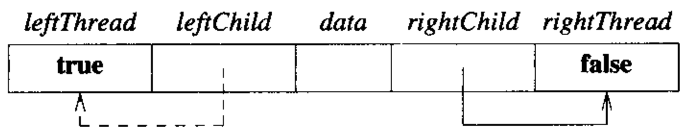
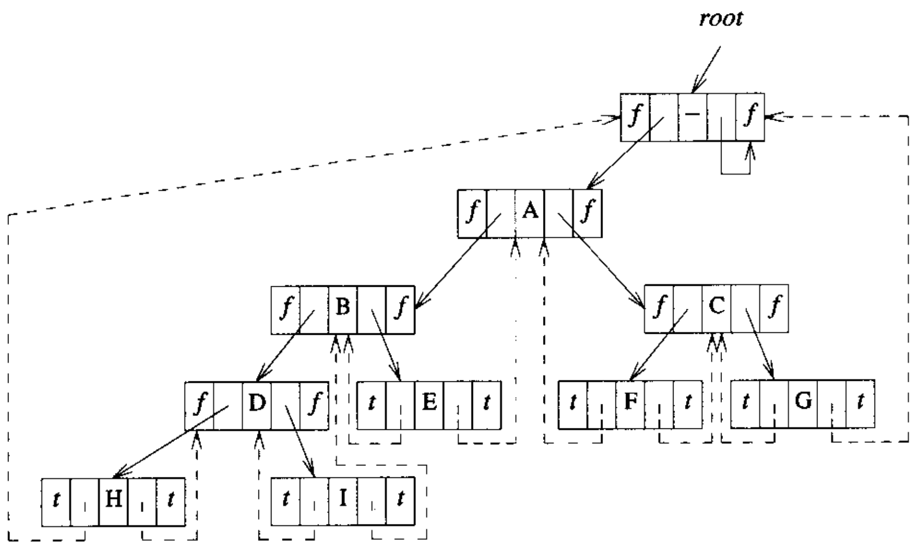


Figure 5.22: An empty threaded binary tree



f= **false**; t = **true**

Figure 5.23: Memory representation of threaded tree

The variable *root* points to the header node of the tree, while *root* $\rightarrow$ *leftChild* points to the start of the first node of the actual tree. This is true for all threaded trees. Notice that we have handled the problem of the loose threads by having them point to the head node, *root*.

5.5.2 Inorder Traversal of a Threaded Binary Tree

By using the threads, we can perform an inorder traversal without making use of a stack. Observe that for any node, *ptr*, in a threaded binary tree, if *ptr* → *rightThread* = *TRUE*, the inorder successor of *ptr* is *ptr* → *rightChild* by definition of the threads. Otherwise we obtain the inorder successor of *ptr* by following a path of left-child links from the right-child of *ptr* until we reach a node with *leftThread* = *TRUE*. The function *insucc* (Program 5.10) finds the inorder successor of any node in a threaded tree without using a stack.

```
threadedPointer insucc(threadedPointer tree)
{/* find the inorder sucessor of tree in a threaded binary
    tree */
  threadedPointer temp;
  temp = tree→rightChild;
  if (!tree→rightThread)
    while (!temp→leftThread)
      temp = temp→leftChild;
  return temp;
}
```

Program 5.10: Finding the inorder successor of a node

To perform an inorder traversal we make repeated calls to *insucc*. The operation is implemented in *tinorder* (Program 5.11). This function assumes that the tree is pointed to by the header node's left child and that the header node's right thread is *FALSE*. The computing time for *tinorder* is still O(*n*) for a threaded binary tree with *n* nodes, although the constant factor is smaller than that of *iterInorder*.

5.5.3 Inserting a Node into a Threaded Binary Tree

We now examine how to make insertions into a threaded tree. This will give us a function for growing threaded trees. We shall study only the case of inserting *r* as the right child of a node *s*. The case of insertion of a left child is given as an exercise. The cases

```
void tinorder(threadedPointer tree)
{/* traverse the threaded binary tree inorder */
   threadedPointer temp = tree;
   for (;;) {
      temp = insucc(temp);
      if (temp == tree) break;
      printf("%3c", temp→data);
   }
}
```

Program 5.11: Inorder traversal of a threaded binary tree

for insertion are

(1) If s has an empty right subtree, then the insertion is simple and diagrammed in Figure 5.24(a).

(2) If the right subtree of s is not empty, then this right subtree is made the right subtree of r after insertion. When this is done, r becomes the inorder predecessor of a node that has a *leftThread* == **true** field, and consequently there is a thread which has to be updated to point to r. The node containing this thread was previously the inorder successor of s. Figure 5.24(b) illustrates the insertion for this case. The function *insertRight* (Program 5.12) contains the C code which handles both cases.

EXERCISES

1. Draw the binary tree of Figure 5.15, showing its threaded representation.
2. Write a function, *insertLeft*, that inserts a new node, *child*, as the left child of node *parent* in a threaded binary tree. The left child pointer of *parent* becomes the left child pointer of *child*.
3. Write a function that traverses a threaded binary tree in postorder. What are the time and space requirements of your method?
4. Write a function that traverses a threaded binary tree in preorder. What are the time and space requirements of your method?

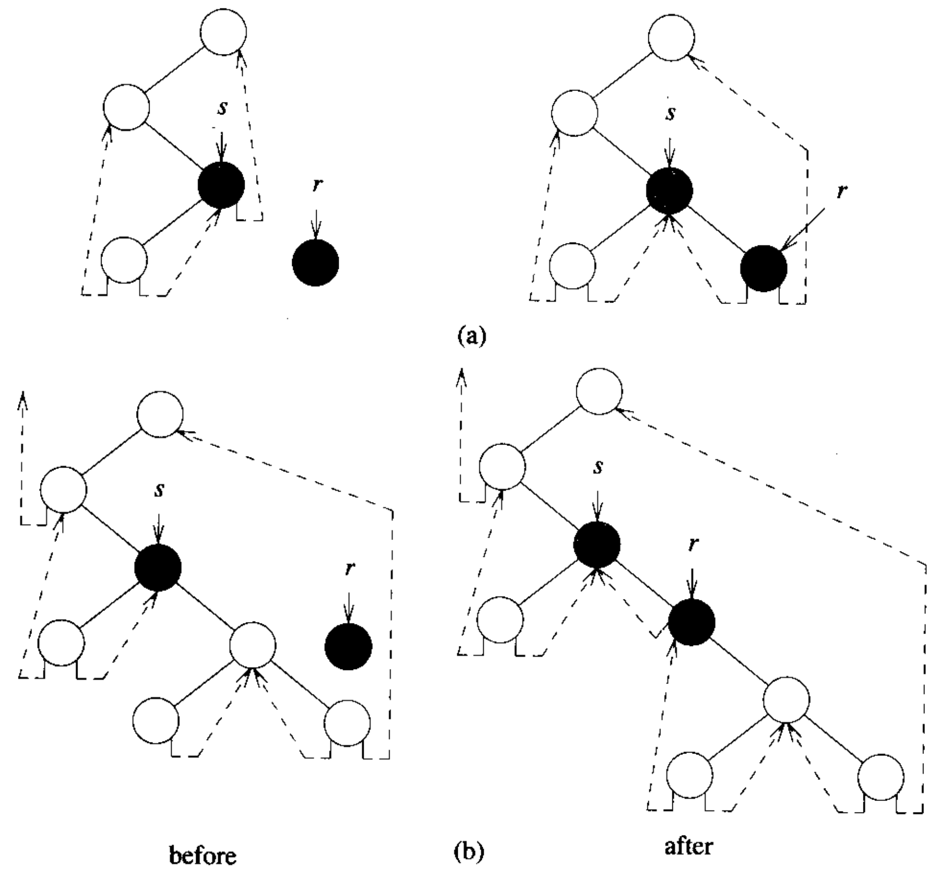


Figure 5.24: Insertion of *r* as a right child of *s* in a threaded binary tree

5.6 HEAPS

5.6.1 Priority Queues

Heaps are frequently used to implement *priority queues*. In this kind of queue, the element to be deleted is the one with highest (or lowest) priority. At any time, an element with arbitrary priority can be inserted into the queue. ADT 5.2 specifies a max priority queue.

```
void insertRight(threadedPointer s, threadedPointer r)
{/* insert r as the right child of s */
   threadedPointer temp;
   r→rightChild = parent→rightChild;
   r→rightThread = parent→rightThread;
   r→leftChild = parent;
   r→leftThread = TRUE;
   s→rightChild = child;
   s→rightThread = FALSE;
   if (!r→rightThread) {
      temp = insucc(r);
      temp→leftChild = r;
   }
}
```

Program 5.12: Right insertion in a threaded binary tree

ADT *MaxPriorityQueue* is
  objects: a collection of $n > 0$ elements, each element has a key
  functions:
    for all $q \in$ *MaxPriorityQueue*, *item* $\in$ *Element*, $n \in$ integer

MaxPriorityQueue create(*max_size*)	::=	create an empty priority queue.
Boolean isEmpty(q, n)	::=	**if** ($n > 0$) **return** *TRUE* **else return** *FALSE*
Element top(q, n)	::=	**if** (!isEmpty(q, n)) **return** an instance of the largest element in q **else return** error.
Element pop(q, n)	::=	**if** (!isEmpty(q, n)) **return** an instance of the largest element in q and remove it from the heap **else return** error.
MaxPriorityQueue push(q, *item*, n)	::=	insert *item* into *pq* and return the resulting priority queue.

ADT 5.2: Abstract data type *MaxPriorityQueue*

Example 5.1: Suppose that we are selling the services of a machine. Each user pays a fixed amount per use. However, the time needed by each user is different. We wish to maximize the returns from this machine under the assumption that the machine is not to be kept idle unless no user is available. This can be done by maintaining a priority queue of all persons waiting to use the machine. Whenever the machine becomes available, the user with the smallest time requirement is selected. Hence, a min priority queue is required. When a new user requests the machine, his/her request is put into the priority queue.

If each user needs the same amount of time on the machine but people are willing to pay different amounts for the service, then a priority queue based on the amount of payment can be maintained. Whenever the machine becomes available, the user paying the most is selected. This requires a max priority queue. □

Example 5.2: Suppose that we are simulating a large factory. This factory has many machines and many jobs that require processing on some of the machines. An *event* is said to occur whenever a machine completes the processing of a job. When an event occurs, the job has to be moved to the queue for the next machine (if any) that it needs. If this queue is empty, the job can be assigned to the machine immediately. Also, a new job can be scheduled on the machine that has become idle (provided that its queue is not empty).

To determine the occurrence of events, a priority queue is used. This queue contains the finish time of all jobs that are presently being worked on. The next event occurs at the least time in the priority queue. So, a min priority queue can be used in this application. □

The simplest way to represent a priority queue is as an unordered linear list. Regardless of whether this list is represented sequentially or as a chain, the *isEmpty* function takes $O(1)$ time; the *top* () function takes $\Theta(n)$ time, where n is the number of elements in the priority queue; a push can be done in $O(1)$ time as it doesn't matter where in the list the new element is inserted; and a *pop* takes $\Theta(n)$ time as me must first find the element with max priority and then delete it. As we shall see shortly, when a max heap is used, the complexity of *isEmpty* and *top* is $O(1)$ and that of *push* and *pop* is $O(\log n)$.

5.6.2 Definition of a Max Heap

In Section 5.2.2, we defined a complete binary tree. In this section we present a special form of a complete binary tree that is useful in many applications.

Definition: A *max (min) tree* is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any). A *max heap* is a complete binary tree that is also a max tree. A *min heap* is a complete binary tree that is also a min tree. □

Some examples of max heaps and min heaps are shown in Figures 5.25 and 5.26, respectively.

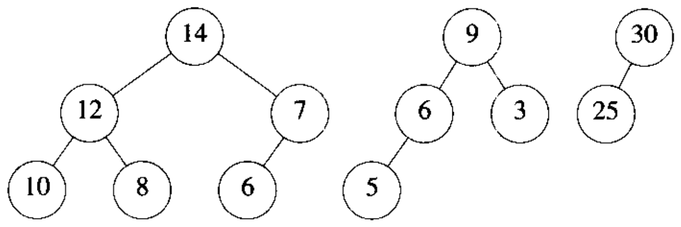


Figure 5.25: Max heaps

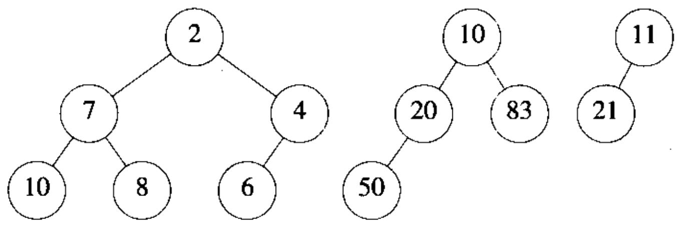


Figure 5.26: Min heaps

From the definitions, it follows that the key in the root of a min tree is the smallest key in the tree, whereas that in the root of a max tree is the largest. When viewed as an ADT, a max heap is very simple. The basic operations are the same as those for a max priority queue (ADT 5.2). Since a max heap is a complete binary tree, we represent it using an array *heap*.

5.6.3 Insertion into a Max Heap

A max heap with five elements is shown in Figure 5.27(a). When an element is added to this heap, the resulting six-element heap must have the structure shown in Figure 5.27(b), because a heap is a complete binary tree. To determine the correct place for the element that is being inserted, we use a *bubbling up* process that begins at the new node of the tree and moves toward the root. The element to be inserted bubbles up as far as is necessary to ensure a max heap following the insertion. If the element to be inserted has key value 1, it may be inserted as the left child of 2 (i.e., in the new node). If instead, the key value of the new element is 5, then this cannot be inserted as the left child of 2 (as otherwise, we will not have a max heap following the insertion). So, the 2 is moved down to its left child (Figure 5.27(c)), and we determine if placing the 5 at the old position of 2 results in a max heap. Since the parent element (20) is at least as large as the element being inserted (5), it is all right to insert the new element at the position shown in the figure. Next, suppose that the new element has value 21 rather than 5. In this case, the 2 moves down to its left child as in Figure 5.27(c). The 21 cannot be inserted into the old position occupied by the 2, as the parent of this position is smaller than 21. Hence, the 20 is moved down to its right child and the 21 inserted into the root of the heap (Figure 5.27(d)).

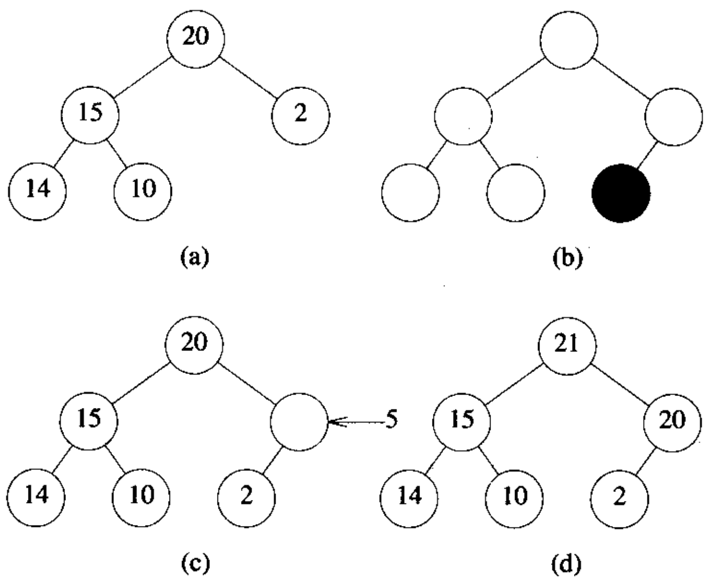


Figure 5.27: Insertion into a max heap

To implement the insertion strategy just described, we need to go from an element to its parent. Lemma 5.4 enables us to locate the parent of any element easily. Program 5.13 performs an insertion into a max heap. We assume that the heap is created using the following C declarations:

```
#define MAX_ELEMENTS 200 /* maximum heap size+1 */
#define HEAP_FULL(n) (n == MAX_ELEMENTS-1)
```

```
#define HEAP_EMPTY(n) (!n)
typedef struct {
        int key;
        /* other fields */
        } element;
element heap[MAX_ELEMENTS];
int n = 0;
```

An alternative representation using a dynamically allocated array whose initial capacity is 1 and doubling array capacity whenever we wish to insert into a full heap is considered in the exercises.

```
void push(element item, int *n)
{/* insert item into a max heap of current size *n */
  int i;
  if (HEAP_FULL(*n)){
    fprintf(stderr, "The heap is full. \n");
    exit(EXIT_FAILURE);
  }
  i = ++(*n);
  while ((i != 1) && (item.key > heap[i/2].key)) {
    heap[i] = heap[i/2];
    i /= 2;
  }
  heap[i] = item;
}
```

Program 5.13: Insertion into a max heap

Analysis of *push*: The function *push* first checks for a full heap. If the heap is not full, we set *i* to the size of the new heap ($n + 1$). We must now determine the correct position of *item* in the heap. We use the **while** loop to accomplish this task. This follows a path from the new leaf of the max heap to the root until it either reaches the root or reaches a position *i* such that the value in the parent position $i/2$ is at least as large as the value to be inserted. Since a heap is a complete binary tree with *n* elements, it has a height of $\lceil \log_2(n + 1) \rceil$. This means that the **while** loop is iterated $O(\log_2 n)$ times. Hence, the complexity of the insertion function is $O(\log_2 n)$. □

5.6.4 Deletion from a Max Heap

When an element is to be deleted from a max heap, it is taken from the root of the heap. For instance, a deletion from the heap of Figure 5.27(d) results in the removal of the element 21. Since the resulting heap has only five elements in it, the binary tree of Figure 5.27(d) needs to be restructured to correspond to a complete binary tree with five elements. To do this, we remove the element in position 6 (i.e., the element 2). Now we have the right structure (Figure 5.28(a)), but the root is vacant and the element 2 is not in the heap. If the 2 is inserted into the root, the resulting binary tree is not a max heap. The element at the root should be the largest from among the 2 and the elements in the left and right children of the root. This element is 20. It is moved into the root, thereby creating a vacancy in position 3. Since this position has no children, the 2 may be inserted here. The resulting heap is shown in Figure 5.27(a).

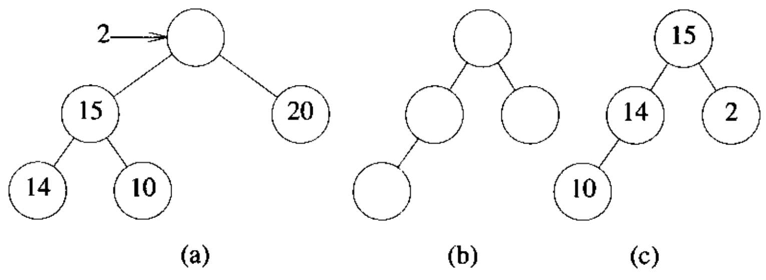


Figure 5.28: Deletion from a heap

Now, suppose we wish to perform another deletion. The 20 is to be deleted. Following the deletion, the heap has the binary tree structure shown in Figure 5.28(b). To get this structure, the 10 is removed from position 5. It cannot be inserted into the root, as it is not large enough. The 15 moves to the root, and we attempt to insert the 10 into position 2. This is, however, smaller than the 14 below it. So, the 14 is moved up and the 10 inserted into position 4. The resulting heap is shown in Figure 5.28(c).

Program 5.14 implements this *trickle down* strategy to delete from a heap.

Analysis of *pop*: The function *pop* operates by moving down the heap, comparing and exchanging parent and child nodes until the heap definition is re-established. Since the height of a heap with n elements is $\lceil \log_2(n+1) \rceil$, the **while** loop of *pop* is iterated $O(\log_2 n)$ times. Hence, the complexity of a deletion is $O(\log_2 n)$. □

```
element pop(int *n)
{/* delete element with the highest key from the heap */
   int parent, child;
   element item, temp;
   if (HEAP_EMPTY(*n)) {
      fprintf(stderr, "The heap is empty\n");
      exit(EXIT_FAILURE);
   }
   /* save value of the element with the highest key */
   item = heap[1];
   /* use last element in heap to adjust heap */
   temp = heap[(*n)--];
   parent = 1;
   child = 2;
   while (child <= *n) {
      /* find the larger child of the current parent */
      if    (child    <    *n)    &&    (heap[child].key    <
     heap[child+1].key)
         child++;
      if (temp.key >= heap[child].key) break;
      /* move to the next lower level */
      heap[parent] = heap[child];
      parent = child;
      child *= 2;
   }
   heap[parent] = temp;
   return item;
}
```

Program 5.14: Deletion from a max heap

EXERCISES

1. Suppose that we have the following key values: 7, 16, 49, 82, 5, 31, 6, 2, 44.
   (a) Write out the max heap after each value is inserted into the heap.
   (b) Write out the min heap after each value is inserted into the heap.
2. Write a structure specification similar to ADT 5.2 for the ADT *MinPQ*, which defines a min priority queue.

3. Compare the run-time performance of max heaps with that of unordered and ordered linear lists as a representation for priority queues. For this comparison, program the max heap push and pop algorithms, as well as algorithms to perform these tasks on unordered and ordered linear lists that are maintained as sequential lists in a one-dimensional array. Generate a random sequence of n values and insert these into the priority queue. Next, perform a random sequence of m inserts and deletes starting with the initial queue of n values. This sequence is to be generated so that the next operation in the sequence has an equal chance of being either an insert or a delete. Care should be taken so that the sequence does not cause the priority queue to become empty at any time. Measure the time taken for the sequence of m operations using both a max heap and an unordered list. Divide the total time by m and plot the times as a function of n. Make some qualitative statements about the relative performance of the two representations for a max priority queue.

4. The worst-case number of comparisons performed during an insertion into a max heap can be reduced to O(loglog n) by performing a binary search on the path from the new leaf to the root. This does not affect the number of data moves though. Write an insertion algorithm that uses this strategy. Redo Exercise 1 using this insertion algorithm. Based on your experiments, what can you say about the value of this strategy over the one used in Program 5.13?

5. Write a C function that changes the priority of an arbitrary element in a max heap. The resulting heap must satisfy the max heap definition. What is the computing time of your function?

6. Write a C function that deletes an arbitrary element from a max heap (the deleted element may be anywhere in the heap). The resulting heap must satisfy the max heap definition. What is the computing time of your function? (Hint: Change the priority of the element to one greater than that of the root, use the change priority function of Exercise 3, and then *pop*.)

7. Write a C function that searches for an arbitrary element in a max heap. What is the computing time of your function?

8. Write insertion and deletion functions for a max heap represented as a linked binary tree. Assume that each node has a parent field as well as the usual left child, right child, and data fields.

9. § [***Programming project***] Write a user-friendly, menu-driven program that allows the user to perform the following operations on min heaps.

   (a) create a min heap

   (b) remove the key with the lowest value

   (c) change the priority of an arbitrary element

   (d) insert an element into the heap.

10. Develop C functions to insert and delete into/from a max heap under the assumptions that a dynamically allocate array is used, the initil capacity of this array is 1, and array doubling is done whenever we are to insert into a max heap that is full. Test your functions.

5.7 BINARY SEARCH TREES

5.7.1 Definition

A *dictionary* is a collection of pairs, each pair has a key and an associated item. Although naturally occurring dictionaries have several pairs that have the same key, we make the assumption here that no two pairs have the same key. The data structure, binary search tree, that we study in this section is easily extended to accommodate dictionaries in which several pairs have the same key. ADT 5.3 gives the specification of a dictionary.

ADT *Dictionary* is

objects: a collection of $n > 0$ pairs, each pair has a key and an associated item

functions:

for all $d \in$ *Dictionary*, *item* $\in$ *Item*, $k \in$ *Key*, $n \in$ integer

Dictionary Create(*max_size*)	::=	create an empty dictionary.
Boolean IsEmpty(*d*, *n*)	::=	**if** ($n > 0$) **return** *TRUE* **else return** *FALSE*
Element Search(*d*, *k*)	::=	**return** item with key *k*, **return** NULL if no such element.
Element Delete(*d*, *k*)	::=	delete and return item (if any) with key *k*;
void Insert(*d*, *item*, *k*)	::=	insert *item* with key *k* into *d*.

ADT 5.3: Abstract data type *dictionary*

A binary search tree has a better performance than any of the data structures studied so far when the functions to be performed are search, insert, and delete. In fact, with a binary search tree, these functions can be performed both by key value and by rank (i.e., find the item with key *k*; find the fifth smallest item; delete the item with key *k*; delete the fifth smallest item; insert an item and determine its rank; and so on).

Definition: A *binary search tree* is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

(1) Each node has exactly one key and the keys in the tree are distinct.

(2) The keys (if any) in the left subtree are smaller than the key in the root.

(3) The keys (if any) in the right subtree are larger than the key in the root.

(4) The left and right subtrees are also binary search trees. □

There is some redundancy in this definition. Properties (2), (3), and (4) together imply that the keys must be distinct. So, property (1) can be replaced by the property: The root has a key.

Some examples of binary trees in which the nodes have distinct keys are shown in Figure 5.29. In this figure, only the key component of each dictionary pair is shown. The tree of Figure 5.29(a) is not a binary search tree, despite the fact that it satisfies properties (1), (2), and (3). The right subtree fails to satisfy property (4). This subtree is not a binary search tree, as its right subtree has a key value (22) that is smaller than that in the subtree's root (25). The binary trees of Figures 5.29(b) and (c) are binary search trees.

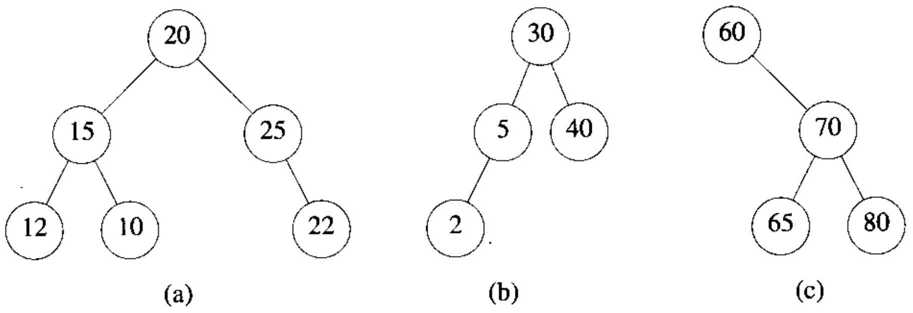


Figure 5.29: Binary trees

5.7.2 Searching a Binary Search Tree

Since the definition of a binary search tree is recursive, it is easiest to describe a recursive search method. Suppose we wish to search for a node whose key is k. We begin at the root of the binary search tree. If the root is *NULL*, the search tree contains no nodes and the search is unsuccessful. Otherwise, we compare k with the key in root. If k equals the root's key, then the search terminates successfully. If k is less than root's key, then no element in the right subtree can have a key value equal to k. Therefore, we search the left subtree of the root. If *k* is larger than root's key value, we search the right subtree of the root. The function *search* (Program 5.15) recursively searches the subtrees. We assume that the *data* field of a node is of type *element* and that the type *element* has two components *key* and *item* whose types are **int** and *iType*, respectively.

```
element* search(treePointer root, int key)
{/* return a pointer to the element whose key is k, if
    there is no such element, return NULL. */
   if (!root) return NULL;
   if (k == root→data.key) return &(root→data);
   if (k < root→data.key)
      return search(root→leftChild, k);
   return search(root→rightChild, k);
}
```

Program 5.15: Recursive search of a binary search tree

We can easily replace the recursive search function with a comparable iterative one. The function *iterSearch* (Program 5.16) accomplishes this by replacing the recursion with a **while** loop.

```
element* iterSearch(treePointer tree, int k)
{/* return a pointer to the element whose key is k,  if
    there is no such element, return NULL. */
   while (tree) {
      if (k == tree→data.key) return &(tree→data);
      if (k < tree→data.key)
         tree = tree→leftChild;
      else
         tree = tree→rightChild;
      }
   return NULL;
}
```

Program 5.16: Iterative search of a binary search tree

Analysis of *search* and *iterSearch*: If *h* is the height of the binary search tree, then we can perform the search using either *search* or *iterSearch* in O(*h*). However, *search* has an additional stack space requirement which is O(h). □

5.7.3 Inserting into a Binary Search Tree

To insert a dictionary pair whose key is k, we must first verify that the key is different from those of existing pairs. To do this we search the tree. If the search is unsuccessful, then we insert the pair at the point the search terminated. For instance, to insert a pair with key 80 into the tree of Figure 5.29(b) (only keys are shown), we first search the tree for 80. This search terminates unsuccessfully, and the last node examined has key 40. We insert the new pair as the right child of this node. The resulting search tree is shown in Figure 5.29(a). Figure 5.30(b) shows the result of inserting the key 35 into the search tree of Figure 5.30(a). This strategy is implemented by *insert* (Program 5.17). This function uses the function *modifiedSearch* which is a slightly modified version of function *iterSearch* (Program 5.16), which searches the binary search tree **node* for the key k. If the tree is empty or if k is present, it returns *NULL*. Otherwise, it returns a pointer to the last node of the tree that was encountered during the search. The new pair is to be inserted as a child of this node.

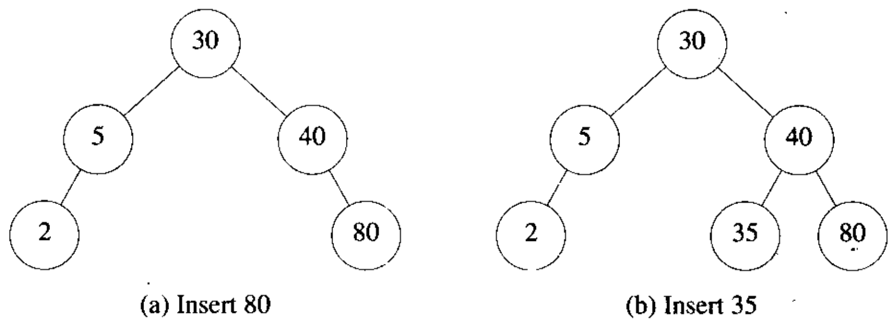


Figure 5.30: Inserting into a binary search tree

Analysis of *insert*: The time required to search the tree for k is O(h) where h is its height. The remainder of the algorithm takes $\Theta(1)$ time. So, the overall time needed by *insert* is O(h). □

```
void insert(treePointer *node, int k, iType theItem)
{/* if k is in the tree pointed at by node do nothing;
    otherwise add a new node with data = (k, theItem) */
  treePointer ptr, temp = modifiedSearch(*node, k);
  if (temp || !(*node)) {
    /* k is not in the tree */
    MALLOC(ptr, sizeof(*ptr));
    ptr→data.key = k;
    ptr→data.item = theItem;
    ptr→leftChild = ptr→rightChild = NULL;
    if (*node) /* insert as child of temp */
      if (k < temp→data.key) temp→leftChild = ptr;
      else temp→rightChild = ptr;
    else *node = ptr;
  }
}
```

Program 5.17: Inserting a dictionary pair into a binary search tree

5.7.4 Deletion from a Binary Search Tree

Deletion of a leaf is quite easy. For example, to delete 35 from the tree of Figure 5.30(b), the left-child field of its parent is set to 0 (*NULL*) and the node freed. This gives us the tree of Figure 5.30(a). To delete the 80 from this tree, the right-child field of 40 is set to 0, obtaining the tree of Figure 5.29(b), and the node containing 80 is freed.

The deletion of a nonleaf that has only one child is also easy. The node containing the dictionary pair to be deleted is freed, and its single-child takes the place of the freed node. So, to delete the 5 from the tree of Figure 5.30(a), we simply change the pointer from the parent node (i.e., the node containing 30) to the single-child node (i.e., the node containing 2).

When the pair to be deleted is in a nonleaf node that has two children, the pair to be deleted is replaced by either the largest pair in its left subtree or the smallest one in its right subtree. Then we proceed to delete this replacing pair from the subtree from which it was taken. For instance, if we wish to delete the pair with key 30 from the tree of Figure 5.30(a), then we replace it by either the largest pair, the one with key 5, in its left subtree or the smallest pair, the one with key 40, in its right subtree. Suppose we opt for the largest pair in the left subtree. The pair with key 5 is moved into the root, and the tree of Figure 5.31(a) is obtained. Now we must delete the second 5. Since the node with the second 5 has only one child, the pointer from its parent is changed to point to this child. The tree of Figure 5.31(b) is obtained. One may verify that regardless of whether the replacing pair is the largest in the left subtree or the smallest in the right subtree, it is originally in a node with a degree of at most one. So, deleting it from this node is quite easy. We leave the writing of the deletion function as an exercise. It should be evident that a deletion can be performed in $O(h)$ time if the search tree has a height of h.

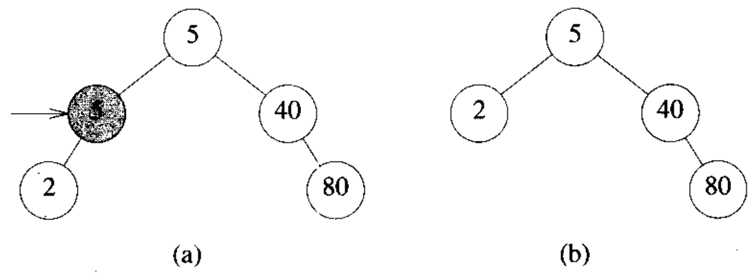


Figure 5.31: Deletion from a binary search tree

5.7.5 Joining and Splitting Binary Trees

Although search, insert, and delete are the operations most frequently performed on a binary search tree, the following additional operations are useful in certain applications:

(a) *threeWayJoin (small,mid,big)*: This creates a binary search tree consisting of the pairs initially in the binary search trees *small* and *big*, as well as the pair *mid*. It is assumed that each key in *small* is smaller than *mid . key* and that each key in *big* is greater than *mid . key*. Following the join, both *small* and *big* are empty.

(b) *twoWayJoin (small,big)*: This joins the two binary search trees *small* and *big* to obtain a single binary search tree that contains all the pairs originally in *small* and *big*. It is assumed that all keys of *small* are smaller than all keys of *big* and that following the join both *small* and *big* are empty.

(c) *split (theTree,k,small,mid,big)*: The binary search tree *theTree* is split into three parts: *small* is a binary search tree that contains all pairs of *theTree* that have key less than *k*; *mid* is the pair (if any) in *theTree* whose key is *k*; and *big* is a binary search tree that contains all pairs of *theTree* that have key larger than *k*. Following the split operation *theTree* is empty. When *theTree* has no pair whose key is *k*, *mid.key* is set to -1 (this assumes that -1 is not a valid key for a dictionary pair).

A three-way join operation is particularly easy to perform. We simply obtain a new node and set its data field to *mid*, its left-child pointer to *small*, and its right-child pointer to *big*. This new node is the root of the binary search tree that was to be created. Finally, *small* and *big* are set to NULL. The time taken for this operation is O(1), and the height of the new tree is max{*height* (*small*), *height* (*big*)} + 1.

Consider the two-way join operation. If either *small* or *big* is empty, the result is the other tree. When neither is empty, we may first delete from *small* the pair *mid* with the largest key. Let the resulting binary search tree be *small*′. To complete the operation, we perform the three-way join operation *threeWayJoin* (*small*′,*mid*,*big*). The overall time required to perform the two-way join operation is O(*height* (*small*)), and the height of the resulting tree is max{*height* (*small*′), *height* (*big*)} + 1. The run time can be made O(min{*height* (*small*), *height* (*big*)}) if we retain with each tree its height. Then we delete the pair with the largest key from *small* if the height of *small* is no more than that of *big*; otherwise, we delete from *big* the pair with the smallest key. This is followed by a three-way join operation.

To perform a split, we first make the following observation about splitting at the root (i.e., when k = *theTree* →*data* . *key*). In this case, *small* is the left subtree of *theTree*, *mid* is the pair in the root, and *big* is the right subtree of *theTree*. If k is smaller than the key at the root, then the root together with its right subtree is to be in *big*. When k is larger than the key at the root, the root together with its left subtree is to be in *small*. Using these observations, we can perform a split by moving down the search tree *theTree* searching for a pair with key k. As we move down, we construct the two search trees *small* and *big*. The function to split *theTree* given in Program 5.18. To simplify the code, we begin with two header nodes *sHead* and *bHead* for *small* and *big*, respectively. *small* is grown as the right subtree of *sHead*; *big* is grown as the left subtree of *bHead*. *s* (*b*) points to the node of *sHead* (*bHead*) at which further subtrees of *theTree* that are to be part of *small* (*big*) may be attached. Attaching a subtree to *small* (*big*) is done as the right (left) child of *s* (*b*).

Analysis of *split*: The **while** loop maintains the invariant that all keys in the subtree with root *currentNode* are larger than those in the tree rooted at *sHead* and smaller than those in the tree rooted at *bHead*. The correctness of the function is easy to establish, and its complexity is seen to be O(*height* (*theTree*)). One may verify that neither *small* nor *big* has a height larger than that of *theTree*. □

5.7.6 Height of a Binary Search Tree

Unless care is taken, the height of a binary search tree with n elements can become as large as n. This is the case, for instance, when Program 5.17 is used to insert the keys [1, 2, 3, . . ., n], in this order, into an initially empty binary search tree. It can, however, be shown that when insertions and deletions are made at random using the functions given here, the height of the binary search tree is O(log n) on the average.

```
void split(nodePointer *theTree, int k, nodePointer *small,
           element *mid, nodePointer *big)
{/* split the binary search tree with respect to key k */
   if (!theTree) {*small = *big = 0;
                  (*mid).key = -1; return;} /* empty tree */
   nodePointer sHead, bHead, s, b, currentNode;
   /* create header nodes for small and big */
   MALLOC(sHead, sizeof(*sHead));
   MALLOC(bHead, sizeof(*bHead));
   s  = sHead; b = bHead;

   /* do the split */
   currentNode  = *theTree;
   while (currentNode)
      if (k < currentNode→data.key) {/* add to big */
         b→leftChild = currentNode;
         b = currentNode; currentNode = currentNode→leftChild;
      }
      else if (k > currentNode→data.key) {/* add to small */
         s→rightChild = currentNode;
         s = currentNode; currentNode = currentNode→rightChild,
      }
      else {/* split at currentNode */
         s→rightChild = currentNode→leftChild;
         b→leftChild = currentNode→rightChild;
         *small = sHead→rightChild; free(sHead);
         *big = bHead→leftChild; free(bHead);
         (*mid).item = currentNode→data.item;
         (*mid).key = currentNode→data.key;
         free(currentNode);
         return;
      }
   /* no pair with key k */
   s→rightChild = b→leftChild = 0;
   *small = sHead→rightChild; free(sHead);
   *big = bHead→leftChild; free(bHead);
   (*mid).key  =  -1;
   return;
}
```

Program 5.18 Splitting a binary search tree

Search trees with a worst-case height of O(log n) are called *balanced search trees*. Balanced search trees that permit searches, inserts, and deletes to be performed in O(h) time exist. Most notable among these are AVL, red/black, 2-3, 2-3-4, B, and B^+ trees. These are discussed in Chapters 10 and 11.

EXERCISES

1. Write a C function to delete the element with key k from a binary search tree. What is the time complexity of your function?
2. Write a program to start with an initially empty binary search tree and make n random insertions. Use a uniform random number generator to obtain the values to be inserted. Measure the height of the resulting binary search tree and divide this height by $\log_2 n$. Do this for n = 100, 500, 1000, 2000, 3000, · · · , 10,000. Plot the ratio *height*/$\log_2 n$ as a function of n. The ratio should be approximately constant (around 2). Verify that this is so.
3. Suppose that each node in a binary search tree also has the field *leftSize* as described in the text. Write a function to insert a pair into such a binary search tree. The complexity of your function should be O(h), where h is the height of the search tree. Show that this is the case.
4. Do Exercise 3, but this time write a function to delete the pair with the kth smallest key in the binary search tree.
5. Write a C function that implements the three-way join operation in O(1) time. log
6. Write a C function that implements the two-way join operation in O(h) time, where h is the height of one of the two trees being joined.
7. Any algorithm that merges together two sorted lists of size n and m, respectively, must make at least $n + m - 1$ comparisons in the worst case. What implications does this result have on the time complexity of any comparison-based algorithm that combines two binary search trees that have n and m pairs, respectively?
8. In Chapter 7, we shall see that every comparison-based algorithm to sort n elements must make O($n \log n$) comparisons in the worst case. What implications does this result have on the complexity of initializing a binary search tree with n pairs?
9. Notice that a binary search tree can be used to implement a priority queue.
   (a) Write a C functions for a max priority queue that represents the priority queue as a binary search tree. Your codes for *top*, *pop* and *push* should have complexity O(h), where h is the height of the search tree. Since h is O(log n) on average, we can perform each these priority queue operations in average time O(log n).
   (b) Compare the actual performance of heaps and binary search trees as data structures for priority queues. For this comparison, generate random

sequences of delete max and insert operations and measure the total time taken for each sequence by each of these data structures.

10. Assume that we change the definition of a binary search tree so that equal keys are permitted and that we add a count field to the node structure.
    (a) Rewrite *insertNode* so that it increments the count field when a plural key is found. Otherwise, a new node is created.
    (b) Rewrite *delete* so that it decrements the count field when the key is found. The node is eliminated only if its count is 0.
11. Write the C code for the function *modifiedSearch* that is used in Program 5.17.
12. Obtain a recursive version of *insertNode*. Which of the two versions is more efficient? Why?
13. Write a recursive C function to delete a key from a binary search tree. What is the time and space complexity of your function?
14. Obtain an iterative C function to delete a key from a binary search tree. The space complexity of your function should be O(1). Show that this is the case. What is the time complexity of your function?
15. Assume that a binary search tree is represented as a threaded binary search tree. Write functions to search, insert, and delete.

5.8 SELECTION TREES

5.8.1 Introduction

Suppose we have k ordered sequences, called *runs*, that are to be merged into a single ordered sequence. Each run consists of some records and is in nondecreasing order of a designated field called the *key*. Let n be the number of records in all k runs together. The merging task can be accomplished by repeatedly outputting the record with the smallest key. The smallest has to be found from k possibilities, and it could be the leading record in any of the k runs. The most direct way to merge k runs is to make $k - 1$ comparisons to determine the next record to output. For $k > 2$, we can achieve a reduction in the number of comparisons needed to find the next smallest element by using the *selection tree* data structure. There are two kinds of selection trees: *winner trees* and *loser trees*.

5.8.2 Winner Trees

A *winner tree* is a complete binary tree in which each node represents the smaller of its two children. Thus, the root node represents the smallest node in the tree. Figure 5.32 illustrates a winner tree for the case $k = 8$.

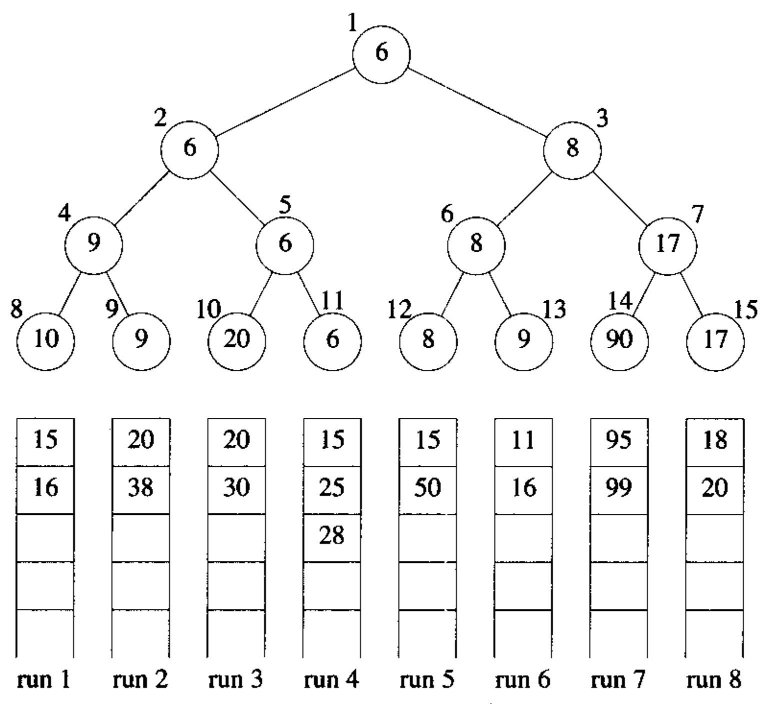


Figure 5.32: Winner tree for $k = 8$, showing the first three keys in each of the eight runs

The construction of this winner tree may be compared to the playing of a tournament in which the winner is the record with the smaller key. Then, each nonleaf node in the tree represents the winner of a tournament, and the root node represents the overall winner, or the smallest key. Each leaf node represents the first record in the corresponding run. Since the records being merged are generally large, each node will contain only a pointer to the record it represents. Thus, the root node contains a pointer to the first record in run 4.

A winner tree may be represented using the sequential allocation scheme for binary trees that results from Lemma 5.4. The number above each node in Figure 5.32 is the address of the node in this sequential representation. The record pointed to by the root has the smallest key and so may be output. Now, the next record from run 4 enters the winner tree. It has a key value of 15. To restructure the tree, the tournament has to be replayed only along the path from node 11 to the root. Thus, the winner from nodes 10 and 11 is again node 11 (15 < 20). The winner from nodes 4 and 5 is node 4 (9 < 15). The winner from 2 and 3 is node 3 (8 < 9). The new tree is shown in Figure 5.33. The tournament is played between sibling nodes and the result put in the parent node. Lemma 5.4 may be used to compute the address of sibling and parent nodes efficiently. Each new comparison takes place at the next higher level in the tree.

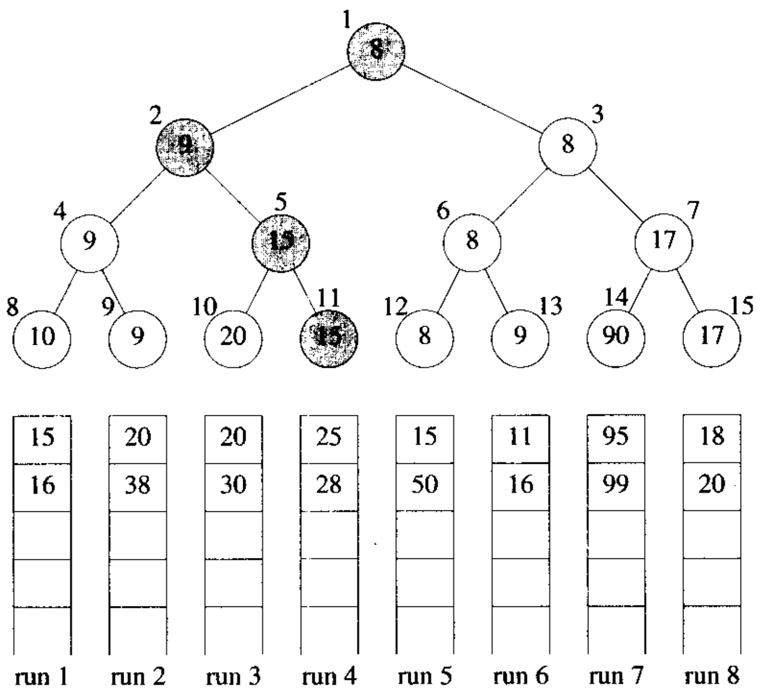


Figure 5.33: Winner tree of Figure 5.32 after one record has been output and the tree restructured (nodes that were changed are shaded)

Analysis of merging runs using winner trees: The number of levels in the tree is $\lceil \log_2(k+1) \rceil$. So, the time to restructure the tree is $O(\log_2 k)$. The tree has to be restructured each time a record is merged into the output file. Hence, the time required to merge all n records is $O(n \log_2 k)$. The time required to set up the selection tree the first time is $O(k)$. Thus, the total time needed to merge the k runs is $O(n \log_2 k)$. □

5.8.3 Loser Trees

After the record with the smallest key value is output, the winner tree of Figure 5.32 is to be restructured. Since the record with the smallest key value is in run 4, this restructuring involves inserting the next record from this run into the tree. The next record has key value 15. Tournaments are played between sibling nodes along the path from node 11 to the root. Since these sibling nodes represent the losers of tournaments played earlier, we can simplify the restructuring process by placing in each nonleaf node a pointer to the record that loses the tournament rather than to the winner of the tournament. A selection tree in which each nonleaf node retains a pointer to the loser is called a *loser tree*. Figure 5.34 shows the loser tree that corresponds to the winner tree of Figure 5.32. For convenience, each node contains the key value of a record rather than a pointer to the record represented. The leaf nodes represent the first record in each run. An additional node, node 0, has been added to represent the overall winner of the tournament. Following the output of the overall winner, the tree is restructured by playing tournaments along the path from node 11 to node 1. The records with which these tournaments are to be played are readily available from the parent nodes. As a result, sibling nodes along the path from 11 to 1 are not accessed.

EXERCISES

1. Write abstract data type specifications for winner and loser trees.
2. Write a function to construct a winner tree for k records. Assume that k is a power of 2. Each node at which a tournament is played should store only a pointer to the winner. Show that this construction can be carried out in time $O(k)$.
3. Do Exercise 2 for the case when k is not restricted to being a power of 2.
4. Write a function to construct a loser tree for k records. Use position 0 of your loser-tree array to store a pointer to the overall winner. Show that this construction can be carried out in time $O(k)$. Assume that k is a power of 2.
5. Do Exercise 4 for the case when k is not restricted to being a power of 2.
6. Write a function, using a tree of losers, to carry out a k-way merge of k runs, $k \geq 2$. Assume the existence of a function to initialize a loser tree in linear time. Show that if there are $n>k$ records in all k runs together, then the computing time is $O(n \log_2 k)$.

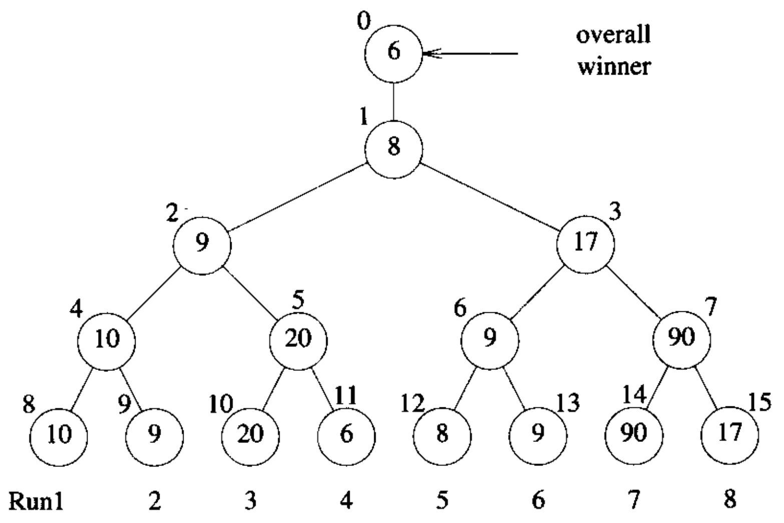


Figure 5.34: Loser tree corresponding to winner tree of Figure 5.32

7. Do the previous exercise for the case in which a tree of winners is used. Assume the existence of a function to initialize a winner tree in linear time.
8. Compare the performance of your functions for the preceding two exercises for the case $k = 8$. Generate eight runs of data, each having 100 records. Use a random number generator for this (the keys obtained from the random number generator will need to be sorted before the merge can begin). Measure and compare the time taken to merge the eight runs using the two strategies.

5.9 FORESTS

Definition: A *forest* is a set of $n \geq 0$ disjoint trees. □

A three-tree forest is shown in Figure 5.35. The concept of a forest is very close to that of a tree because if we remove the root of a tree, we obtain a forest. For example, removing the root of any binary tree produces a forest of two trees. In this section, we briefly consider several forest operations, including transforming a forest into a binary tree and forest traversals. In the next section, we use forests to represent disjoint sets.

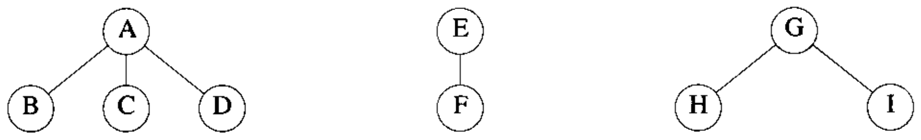


Figure 5.35: Three-tree forest

5.9.1 Transforming a Forest into a Binary Tree

To transform a forest into a single binary tree, we first obtain the binary tree representation of each of the trees in the forest and then link these binary trees together through the *rightChild* field of the root nodes. Using this transformation, the forest of Figure 5.35 becomes the binary tree of Figure 5.36.

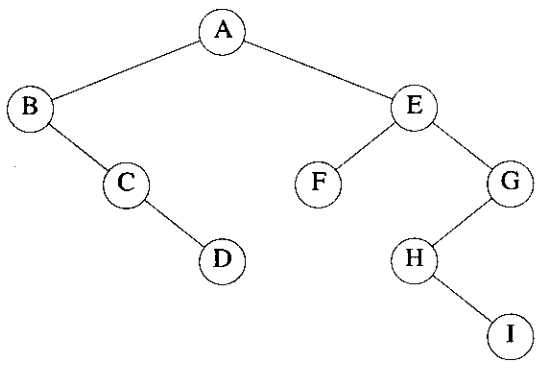


Figure 5.36: Binary tree representation of forest of Figure 5.35

We can define this transformation in a formal way as follows:

Definition: If $T_1, \cdots, T_n$ is a forest of trees, then the binary tree corresponding to this forest, denoted by $B(T_1, \cdots, T_n)$,

(1) is empty if $n = 0$

(2) has root equal to root (T_1); has left subtree equal to $B(T_{11}, T_{12}, \cdots, T_{1m})$, where

$T_{11}, \cdots, T_{1m}$ are the subtrees of root(T_1); and has right subtree $B(T_2, \cdots, T_n)$. □

5.9.2 Forest Traversals

Preorder and inorder traversals of the corresponding binary tree T of a forest F have a natural correspondence to traversals on F. Preorder traversal of T is equivalent to visiting the nodes of F in *forest preorder*, which is defined as follows:

(1) If F is empty then return.

(2) Visit the root of the first tree of F.

(3) Traverse the subtrees of the first tree in forest preorder.

(4) Traverse the remaining trees of F in forest preorder.

Inorder traversal of T is equivalent to visiting the nodes of F in *forest inorder*, which is defined as follows:

(1) If F is empty then return.

(2) Traverse the subtrees of the first tree in forest inorder.

(3) Visit the root of the first tree.

(4) Traverse the remaining trees in forest inorder.

The proofs that preorder and inorder traversals on the corresponding binary tree are the same as preorder and inorder traversals on the forest are left as exercises. There is no natural analog for postorder traversal of the corresponding binary tree of a forest. Nevertheless, we can define the *postorder traversal of a forest* as follows:

(1) If F is empty then return.

(2) Traverse the subtrees of the first tree of F in forest postorder.

(3) Traverse the remaining trees of F in forest postorder.

(4) Visit the root of the first tree of F.

In a *level-order traversal of a forest*, nodes are visited by level, beginning with the roots of each tree in the forest. Within each level, nodes are visited from left to right. One may verify that the level-order traversal of a forest and that of its associated binary tree do not necessarily yield the same result.

EXERCISES

1. Define the inverse transformation of the one that creates the associated binary tree from a forest. Are these transformations unique?
2. Prove that the preorder traversal of a forest and the preorder traversal of its associated binary tree give the same result.
3. Prove that the inorder traversal of a forest and the inorder traversal of its associated binary tree give the same result.
4. Prove that the postorder traversal of a forest and that of its corresponding binary tree do not necessarily yield the same result.
5. Prove that the level-order traversal of a forest and that of its corresponding binary tree do not necessarily yield the same result.
6. Write a nonrecursive function to traverse the associated binary tree of a forest in forest postorder. What are the time and space complexities of your function?
7. Do the preceding exercise for the case of forest level-order traversal.

5.10 REPRESENTATION OF DISJOINT SETS

5.10.1 Introduction

In this section, we study the use of trees in the representation of sets. For simplicity, we assume that the elements of the sets are the numbers 0, 1, 2, $\cdots$, n–1. In practice, these numbers might be indices into a symbol table that stores the actual names of the elements. We also assume that the sets being represented are pairwise disjoint, that is, if S_i and S_j are two sets and $i \neq j$, then there is no element that is in both S_i and S_j. For example, if we have 10 elements numbered 0 through 9, we may partition them into three disjoint sets, $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$, and $S_3 = \{2, 3, 5\}$. Figure 5.37 shows one possible representation for these sets. Notice that for each set we have linked the nodes from the children to the parent, rather than our usual method of linking from the parent to the children. The reason for this change in linkage will become apparent when we discuss the implementation of set operations.

The minimal operations that we wish to perform on these sets are:

(1) *Disjoint set union.* If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = {all elements, x, such that x is in S_i or S_j}. Thus, $S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$. Since we have assumed that all sets are disjoint, following the union of S_i and S_j we can assume that the sets S_i and S_j no longer exist independently. That is, we replace them by $S_i \cup S_j$.

(2) *Find(i).* Find the set containing the element, i. For example, 3 is in set S_3 and 8 is in set S_1.

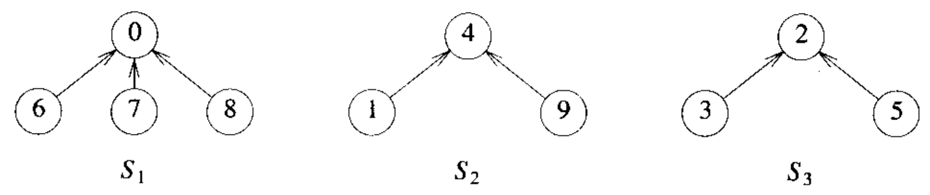


Figure 5.37: Possible tree representation of sets

5.10.2 Union And Find Operations

Let us consider the union operation first. Suppose that we wish to obtain the union of S_1 and S_2. Since we have linked the nodes from children to parent, we simply make one of the trees a subtree of the other. $S_1 \cup S_2$ could have either of the representations of Figure 5.38.

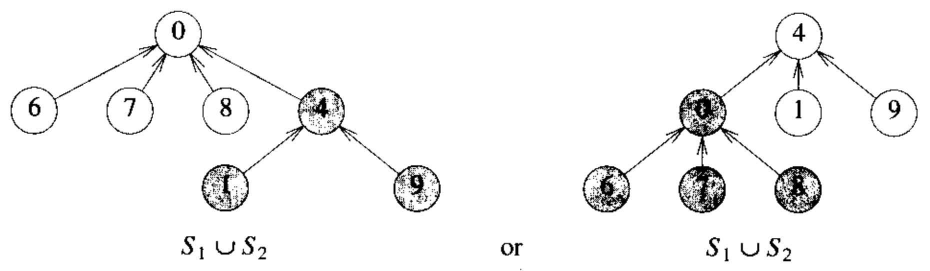


Figure 5.38: Possible representation of $S_1 \cup S_2$

To implement the set union operation, we simply set the parent field of one of the roots to the other root. We can accomplish this easily if, with each set name, we keep a pointer to the root of the tree representing that set. If, in addition, each root has a pointer to the set name, we can find which set an element is in by following the parent links to the root of its tree and then returning the pointer to the set name. Figure 5.39 shows this representation of S_1, S_2, and S_3.

To simplify the discussion of the union and find algorithms, we will ignore the set names and identify the sets by the roots of the trees representing them. For example, rather than using the set name S_1 we refer to this set as 0. The transition to set names is easy. We assume that a table, *name* [ ], holds the set names. If i is an element in a tree

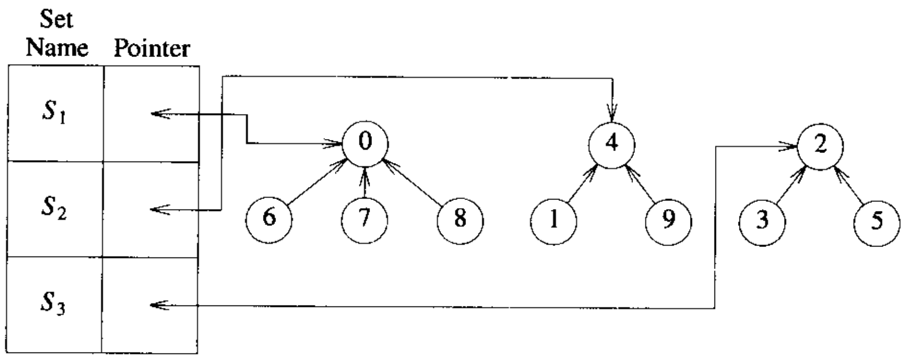


Figure 5.39: Data representation of S_1, S_2, and S_3

with root j, and j has a pointer to entry k in the set name table, then the set name is just *name*[k].

Since the nodes in the trees are numbered 0 through $n - 1$ we can use the node's number as an index. This means that each node needs only one field, the index of its parent, to link to its parent. Thus, the only data structure that we need is an array, *int parent*[*MAX–ELEMENTS*], where *MAX–ELEMENTS* is the maximum number of elements. Figure 5.40 shows this representation of the sets, S_1, S_2, and S_3. Notice that root nodes have a parent of −1.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	−1	4	−1	2	−1	2	0	0	0	4

Figure 5.40: Array representation of S_1, S_2, and S_3

We can now find element i by simply following the parent values starting at i and continuing until we reach a negative parent parent value. For example, to find 5, we start at 5, and then move to 5's parent, 2. Since this node has a negative parent value we have reached the root. The operation *union*(i,j) is equally simple. We pass in two trees with roots i and j. Assuming that we adopt the convention that the first tree becomes a subtree of the second, the statement *parent* [i] = j accomplishes the union. Program 5.19 implements the simple union and find operations as just discussed.

```
int simpleFind(int i)
{
   for(; parent[i] >= 0; i = parent[i])
      ;
   return i;
}
void simpleUnion(int i, int j)
{
   parent[i] = j;
}
```

Program 5.19: Initial attempt at union-find functions

Analysis of *simpleUnion*1 and *simpleFind*1: Although *simpleUnion* and *simpleFind* are easy to implement, their performance characteristics are not very good. For instance, if we start with p elements, each in a set of its own, that is, $S_i = \{i\}$, $0 \le i < p$, then the initial configuration is a forest with p nodes and $parent[i] = -1$, $0 \le i < p$. Now let us process the following sequence of union-find operations:

$$\begin{gathered} union(0, 1), find(0) \\ union(1, 2), find(0) \\ \vdots \\ union(n-2, n-1), find(0) \end{gathered}$$

This sequence produces the degenerate tree of Figure 5.41. Since the time taken for a union is constant, we can process all the $n - 1$ unions in time $O(n)$. However, for each *find*, we must follow a chain of parent links from 0 to the root. If the element is at level i, then the time required to find its root is $O(i)$. Hence, the total time needed to process the $n - 1$ finds is:

$$\sum_{i=2}^{n} i = O(n^2) \square$$

By avoiding the creation of degenerate trees, we can attain far more efficient implementations of the union and find operations. We accomplish this by adopting the following *Weighting rule* for *union(i, j)*.

Definition: *Weighting rule for union(i, j).* If the number of nodes in tree i is less than the number in tree j then make j the parent of i; otherwise make i the parent of j. □

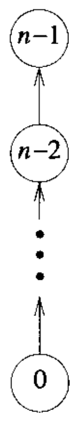


Figure 5.41: Degenerate tree

When we use this rule on the sequence of set unions described above, we obtain the trees of Figure 5.42. To implement the weighting rule, we need to know how many nodes there are in every tree. To do this easily, we maintain a count field in the root of every tree. If i is a root node, then $count[i]$ equals the number of nodes in that tree. Since all nodes but the roots of trees have a nonnegative number in the parent field, we can maintain the count in the parent field of the roots as a negative number. When we incorporate the weighting rule, the union operation takes the form given in *weightedUnion* (Program 5.20). Remember that the arguments passed into *weightedUnion* must be roots of trees.

Lemma 5.5: Let T be a tree with n nodes created as a result of *weightedUnion*. No node in T has level greater than $\lfloor \log_2 n \rfloor + 1$.

Proof: The lemma is clearly true for $n = 1$. Assume that it is true for all trees with i nodes, $i \leq n - 1$. We show that it is also true for $i = n$. Let T be a tree with n nodes created by *weightedUnion*. Consider the last union operation performed, *union*(k,j). Let m be the number of nodes in tree j and $n-m$, the number of nodes in k. Without loss of generality, we may assume that $1 \leq m \leq n / 2$. Then the maximum level of any node in T is either the same as k or is one more than in j. If the former is the case, then the maximum level in T is $\leq \lfloor \log_2(n-m) \rfloor + 1 \leq \lfloor \log_2 n \rfloor + 1$. If the latter is the case, then the maximum level is $\leq \lfloor \log_2 m \rfloor + 2 \leq \lfloor \log_2 n/2 \rfloor + 2 \leq \lfloor \log_2 n \rfloor + 1$. □

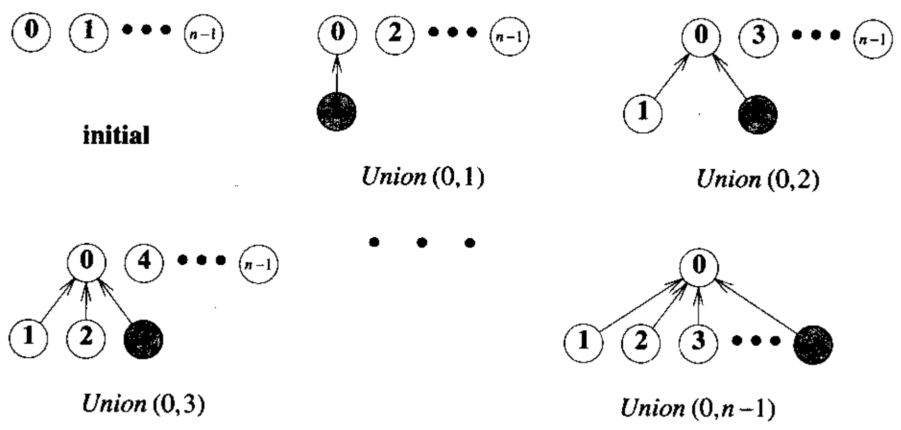


Figure 5.42: Trees obtained using the weighting rule

```
void weightedUnion(int i, int j)
{/* union the sets with roots i and j, i != j, using
    the weighting rule. parent[i] = -count[i] and
    parent[j] = -count[j] */
   int temp = parent[i] + parent[j];
   if (parent[i] > parent[j]) {
     parent[i] = j; /* make j the new root */
     parent[j] = temp;
   }
   else {
     parent[j] = i; /*make i the new root */
     parent[i] = temp;
   }
}
```

Program 5.20: Union function using weighting rule

Example 5.3 shows that the bound of Lemma 5.5 is achievable for some sequence of unions.

Example 5.3: Consider the behavior of *weightedUnion* on the following sequence of unions starting from the initial configuration of *parent* $[i] = -count[i] = -1$, $0 \le i < n = 2^3$:

union(0, 1) *union*(2, 3) *union*(4, 5) *union*(6, 7)
union(0, 2) *union*(4, 6) *union*(0, 4)

When the sequence of unions is performed by columns (i.e., top to bottom within a column with column 1 first, column 2 next, and so on), the trees of Figure 5.43 are obtained. As is evident from this example, in the general case, the maximum level can be $\lfloor \log_2 m \rfloor + 1$ if the tree has m nodes. □

From Lemma 5.5, it follows that the time to process a find is $O(\log m)$ if there are m elements in a tree. If an intermixed sequence of $u - 1$ union and f find operations is to be processed, the time becomes $O(u + f \log u)$, as no tree has more than u nodes in it. Of course, we need $O(n)$ additional time to initialize the n-tree forest.

Surprisingly, further improvement is possible. This time the modification will be made in the find algorithm using the *collapsing rule*.

Definition [*Collapsing rule*]: If j is a node on the path from i to its root and *parent* $[i] \ne$ *root* (i), then set *parent* $[j]$ to *root* (i). □

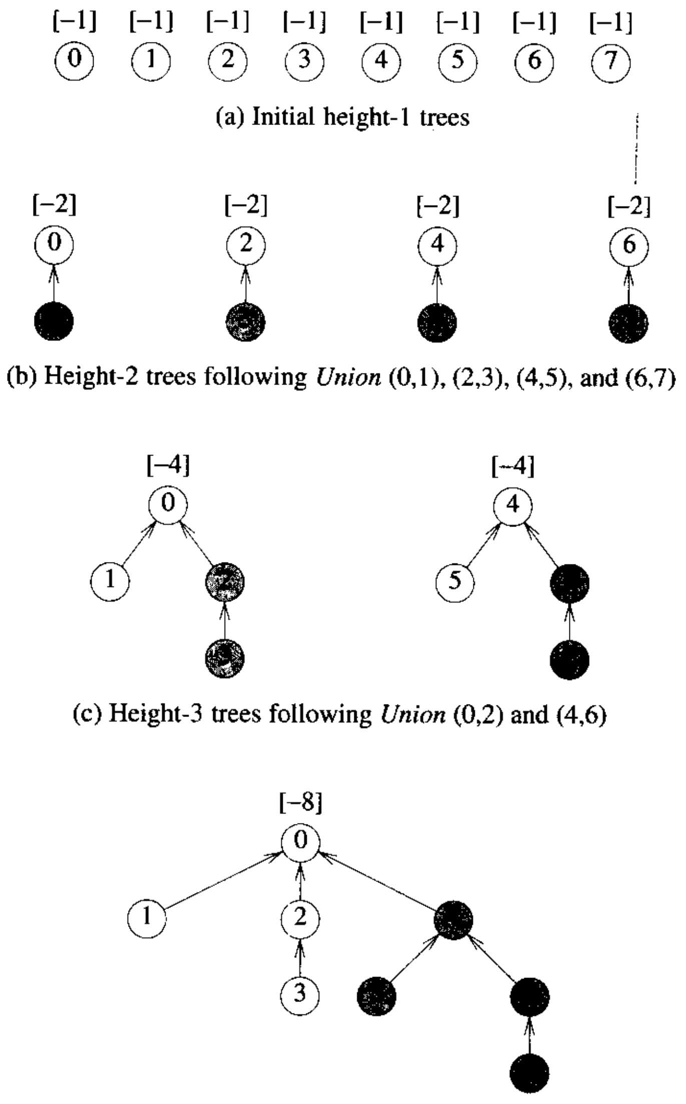
Function *collapsingFind* (Program 5.21) incorporates the collapsing rule.

Example 5.4: Consider the tree created by function *weightedUnion* on the sequence of unions of Example 5.5. Now process the following eight finds:

$$find(7), find(7), \cdots, find(7)$$

If *simpleFind* is used, each *find*(7) requires going up three parent link fields for a total of 24 moves to process all eight finds. When *collapsingFind* is used, the first *find*(7) requires going up three links and then resetting two links. Note that even though only two parent links need to be reset, function *collapsingFind* will actually reset three (the parent of 4 is reset to 0). Each of the remaining seven finds requires going up only one link field. The total cost is now only 13 moves. □

Analysis of *weightedUnion* and *collapsingFind*: Use of the collapsing rule roughly doubles the time for an individual find. However, it reduces the worst-case time over a sequence of finds. The worst-case complexity of processing a sequence of unions and finds using *weightedUnion* and *collapsingFind* is stated in Lemma 5.6. This lemma



(a) Initial height-1 trees

(b) Height-2 trees following *Union* (0,1), (2,3), (4,5), and (6,7)

(c) Height-3 trees following *Union* (0,2) and (4,6)

(d) Height-4 tree following *Union* (0,4)

Figure 5.43: Trees achieving worst case bound

```
int collapsingFind(int i)
{/* find the root of the tree containing element i. Use the
    collapsing rule to collapse all nodes from i to root */
  int root, trail, lead;
  for (root = i; parent[root] >= 0; root = parent[root])
    ;
  for (trail = i; trail != root; trail = lead) {
    lead = parent[trail];
    parent[trail] = root;
  }
  return root;
}
```

Program 5.21: Collapsing rule

makes use of a function $\alpha(p,q)$ that is related to a functional inverse of Ackermann's function $A(i,j)$. These functions are defined as follows:

$$A(1,j) = 2^j, \quad \text{for } j \geq 1$$
$$A(i,1) = A(i-1,2) \quad \text{for } i \geq 2$$
$$A(i,j) = A(i-1, A(i,j-1)) \quad \text{for } i,j \geq 2$$

$$\alpha(p,q) = \min\{z \geq 1 \mid A(z, \lfloor p/q \rfloor) > \log_2 q\}, \quad p \geq q \geq 1$$

The function $A(i,j)$ is a very rapidly growing function. Consequently, α grows very slowly as p and q are increased. In fact, since $A(3,1) = 16$, $\alpha(p,q) \leq 3$ for $q < 2^{16} = 65{,}536$ and $p \geq q$. Since $A(4,1)$ is a very large number and in our application q will be the number, n, of set elements and p will be $n + f$ (f is the number of finds), $\alpha(p,q) \leq 4$ for all practical purposes. □

Lemma 5.6 [***Tarjan and Van Leeuwen***]: Assume that we start with a forest of trees, each having one node. Let $T(f,u)$ be the maximum time required to process any intermixed sequence of f finds and u unions. Assume that $u \geq n/2$. Then

$$k_1(n + f\,\alpha(f+n,n)) \leq T(f,u) \leq k_2(n + f\,\alpha(f+n,n))$$

for some positive constants k_1 and k_2. □

The requirement that $u \geq n/2$ in Lemma 5.6, is really not significant, as when $u < n/2$, some elements are involved in no union operation. These elements remain in singleton sets throughout the sequence of union and find operations and can be eliminated from consideration, as find operations that involve these can be done in $O(1)$ time each. Even though the function $\alpha(f,u)$ is a very slowly growing function, the complexity of our solution to the set representation problem is not linear in the number of unions and finds. The space requirements are one node for each element.

In the exercises, we explore alternatives to the weight rule and the collapsing rule that preserve the time bounds of Lemma 5.6.

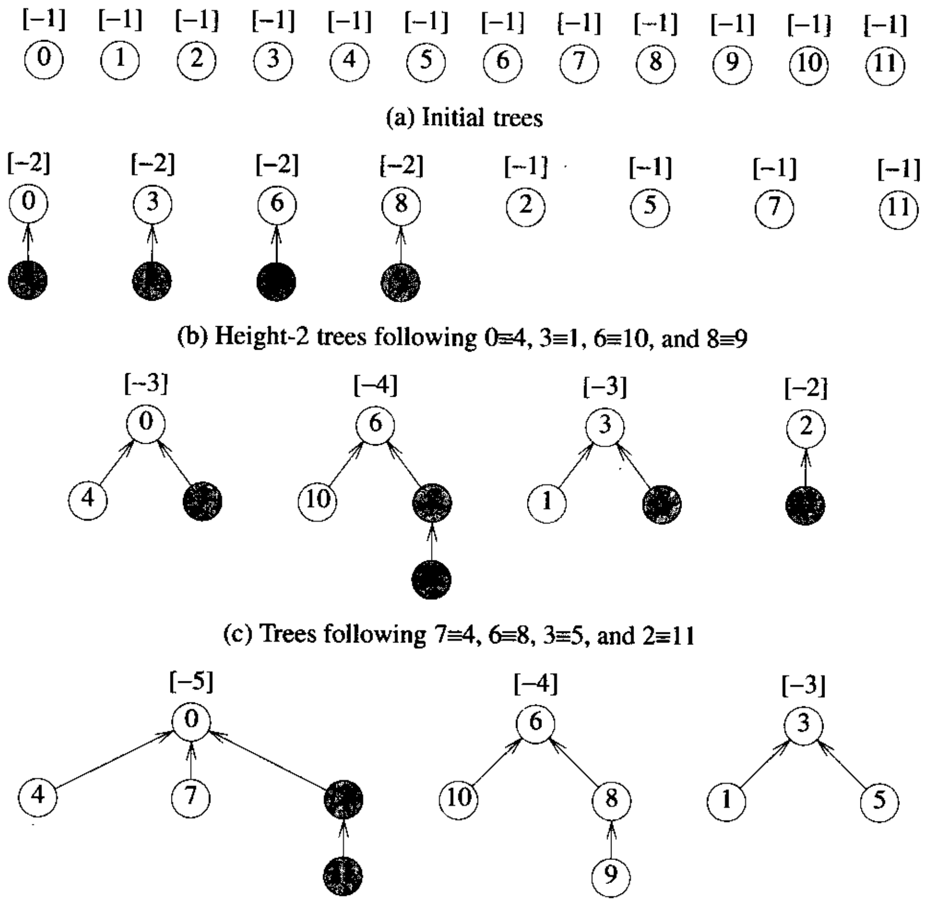
5.10.3 Application to Equivalence Classes

Consider the equivalence pairs processing problem of Section 4.6. The equivalence classes to be generated may be regarded as sets. These sets are disjoint, as no polygon can be in two equivalence classes. Initially, all n polygons are in an equivalence class of their own; thus $parent[i] = -1, 0 \le i < n$. If an equivalence pair, $i \equiv j$, is to be processed, we must first determine the sets containing i and j. If these are different, then the two sets are to be replaced by their union. If the two sets are the same, then nothing is to be done, as the relation $i \equiv j$ is redundant; i and j are already in the same equivalence class. To process each equivalence pair we need to perform two finds and at most one union. Thus, if we have n polygons and m equivalence pairs, we need to spend $O(n)$ time to set up the initial n-tree forest, and then we need to process $2m$ finds and at most $\min\{n-1, m\}$ unions. (Note that after $n-1$ unions, all n polygons will be in the same equivalence class and no more unions can be performed.) If we use *weightedUnion* and *collapsingFind*, the total time to process the equivalence relations is $O(n + m\alpha(2m, \min\{n-1, m\}))$. Although this is slightly worse than the algorithm of Section 4.9, it needs less space and is on line. By "on line," we mean that as each equivalence is processed, we can tell which equivalence class each polygon is in.

Example 5.6: Consider the equivalence pairs example of Section 4.6. Initially, there are 12 trees, one for each variable. $parent[i] = -1, 0 \le i < 12$. The tree configuration following the processing of each equivalence pair is shown in Figure 5.44. Each tree represents an equivalence class. It is possible to determine if two elements are currently in the same equivalence class at each stage of the processing simply by making two finds. □

EXERCISES

1. Suppose we start with n sets, each containing a distinct element.
   (a) Show that if u unions are performed, then no set contains more than $u + 1$ elements.
   (b) Show that at most $n - 1$ unions can be performed before the number of sets becomes 1.
   (c) Show that if fewer than $\lceil n/2 \rceil$ unions are performed, then at least one set



(a) Initial trees

(b) Height-2 trees following 0≡4, 3≡1, 6≡10, and 8≡9

(c) Trees following 7≡4, 6≡8, 3≡5, and 2≡11

(d) Trees following 11≡0

Figure 5.44: Trees for Example 5.6

with a single element in it remains.

(d) Show that if u unions are performed, then at least $\max\{n - 2u, 0\}$ singleton sets remain.

2. Using the result of Example 5.6, draw the trees after processing the instruction *union*(11,9).

3. Experimentally compare the performance of *simpleUnion* and *simpleFind* (Program 5.19) with *weightedUnion* (Program 5.20) and *collapsingFind* (Program 5.21). For this, generate a random sequence of union and find operations.

4. (a) Write a function *heightUnion* that uses the *height rule* for union operations instead of the weighting rule. This rule is defined below:

   Definition [*Height Rule*]: If the height of tree *i* is less than that of tree *j*, then make *j* the parent of *i*, otherwise make *i* the parent of *j*. □

   Your function must run in O(1) time and should maintain the height of each tree as a negative number in the *parent* field of the root.

   (b) Show that the height bound of Lemma 5.5 applies to trees constructed using the height rule.

   (c) Give an example of a sequence of unions that start with singleton sets and create trees whose height equals the upper bound given in Lemma 5.5. Assume that each union is performed using the height rule.

   (d) Experiment with functions *weightedUnion* (Program 5.20) and *heightUnion* to determine which one produces better results when used in conjunction with function *collapsingFind* (Program 5.21).

5. (a) Write a function *splittingFind* that uses *path splitting* for the find operations instead of path collapsing. This is defined below:

   Definition [*Path Splitting*]: In path splitting, the parent pointer in each node (except the root and its child) on the path from *i* to the root is changed to point to the node's grandparent. □

   Note that when path splitting is used, a single pass from *i* to the root suffices. Tarjan and Van Leeuwen have shown that Lemma 5.6 holds when path splitting is used in conjunction with either the weight or height rule for unions.

   (b) Experiment with functions *collapsingFind* (Program 5.21) and *splittingFind* to determine which produces better results when used in conjunction with function *weightedUnion* (Program 5.20).

6. (a) Write a function *halvingFind* that uses *path halving* for the find operations instead of path collapsing. This is defined below:

   Definition [*Path Halving*]: In path halving, the parent pointer of every other node (except the root and its child) on the path from *i* to the root is changed to point to the node's grandparent. □

   Note that path halving, like path splitting (Exercise 5) can be implemented with a single pass from *i* to the root. However, in path halving, only half as many pointers are changed as in path splitting. Tarjan and Van Leeuwen have shown that Lemma 5.6 holds when path halving is used in conjunction with either the weight or height rule for unions.

(b) Experiment with functions *collapsingFind* and *halvingFind* to determine which one produces better results when used in conjunction with function *weightedUnion*.

5.11 COUNTING BINARY TREES

As a conclusion to our chapter on trees, we consider three disparate problems that amazingly have the same solution. We wish to determine the number of distinct binary trees having n nodes, the number of distinct permutations of the numbers from 1 through n obtainable by a stack, and the number of distinct ways of multiplying $n + 1$ matrices. Let us begin with a quick look at these problems.

5.11.1 Distinct Binary Trees

We know that if $n = 0$ or $n = 1$, there is only one binary tree. If $n = 2$, then there are two distinct trees (Figure 5.45), and if $n = 3$, there are five such trees (Figure 5.46). How many distinct trees are there with n nodes? Before deriving a solution, we will examine the two remaining problems. You might attempt to sketch out a solution of your own before reading further.

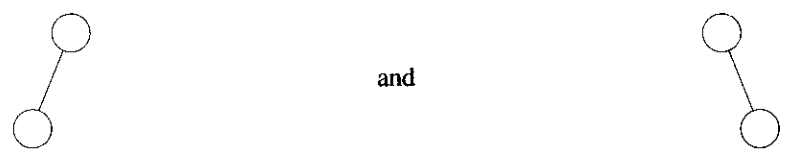


Figure 5.45: Distinct binary trees with $n = 2$

5.11.2 Stack Permutations

In Section 5.3, we introduced preorder, inorder, and postorder traversals and indicated that each traversal requires a stack. Suppose we have the preorder sequence *A B C D E F G H I* and the inorder sequence *B C A E D G H F I* of the same binary tree. Does such a pair of sequences uniquely define a binary tree? Put another way, can this pair of sequences come from more than one binary tree?

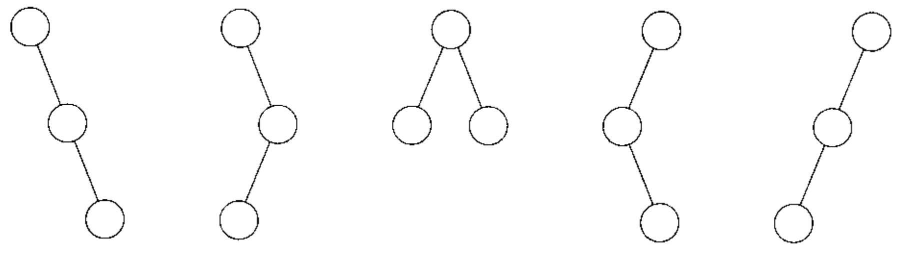


Figure 5.46: Distinct binary trees with $n = 3$

To construct the binary tree from these sequences, we look at the first letter in the preorder sequence, A. This letter must be the root of the tree by definition of the preorder traversal (VLR). We also know by definition of the inorder traversal (LVR) that all nodes preceding A in the inorder sequence (B C) are in the left subtree, and the remaining nodes (E D G H F I) are in the right subtree. Figure 5.47(a) is our first approximation to the correct tree.

Moving right in the preorder sequence, we find B as the next root. Since no node precedes B in the inorder sequence, B has an empty left subtree, which means that C is in its right subtree. Figure 5.47(b) is the next approximation. Continuing in this way, we arrive at the binary tree of Figure 5.48(a). By formalizing this argument (see the exercises), we can verify that every binary tree has a unique pair of preorder/inorder sequences.

Let the nodes of an n-node binary tree be numbered from 1 through n. The inorder permutation defined by such a binary tree is the order in which its nodes are visited during an inorder traversal of the tree. A preorder permutation is similarly defined.

As an example, consider the binary tree of Figure 5.48(a) with the node numbering of Figure 5.48(b). Its preorder permutation is $1, 2, \cdots, 9$, and its inorder permutation is $2, 3, 1, 5, 4, 7, 8, 6, 9$.

If the nodes of the tree are numbered such that its preorder permutation is $1, 2, \cdots, n$, then from our earlier discussion it follows that distinct binary trees define distinct inorder permutations. Thus, the number of distinct binary trees is equal to the number of distinct inorder permutations obtainable from binary trees having the preorder permutation, $1, 2, \cdots, n$.

Using the concept of an inorder permutation, we can show that the number of distinct permutations obtainable by passing the numbers 1 through n through a stack and deleting in all possible ways is equal to the number of distinct binary trees with n nodes (see the exercises). If we start with the numbers 1, 2, and 3, then the possible permutations obtainable by a stack are

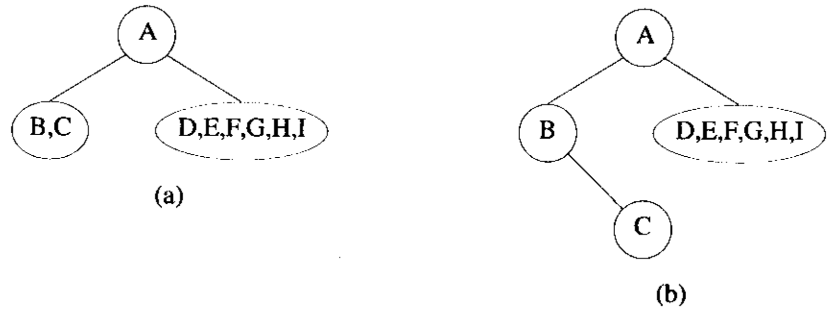


Figure 5.47: Constructing a binary tree from its inorder and preorder sequences

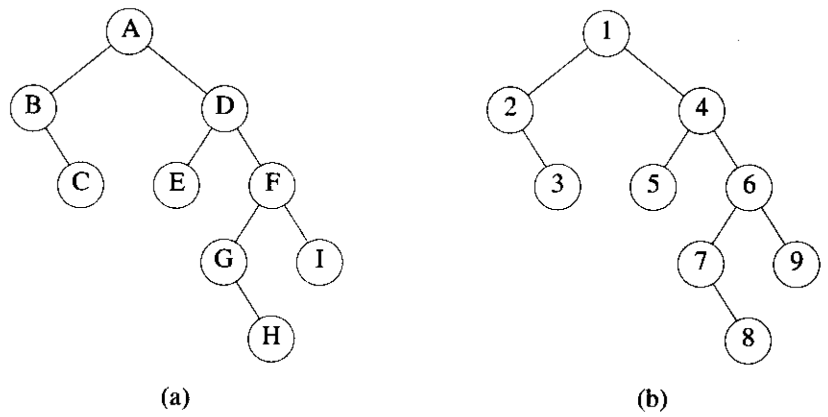


Figure 5.48: Binary tree constructed from its inorder and preorder sequences

$$(1, 2, 3)\ (1, 3, 2)\ (2, 1, 3)\ (2, 3, 1)\ (3, 2, 1)$$

Obtaining (3, 1, 2) is impossible. Each of these five permutations corresponds to one of the five distinct binary trees with three nodes (Figure 5.49).

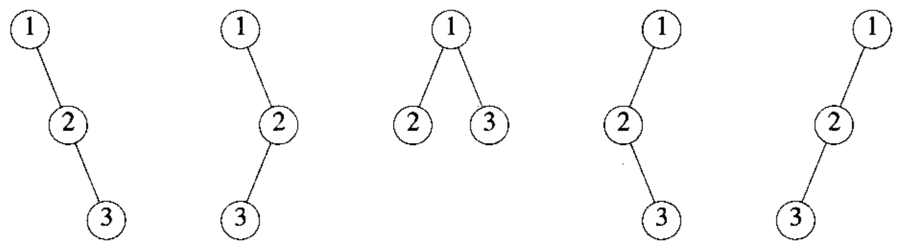


Figure 5.49: Binary trees corresponding to five permutations

5.11.3 Matrix Multiplication

Another problem that surprisingly has a connection with the previous two involves the product of n matrices. Suppose that we wish to compute the product of n matrices:

$$M_1 * M_2 * \cdots * M_n$$

Since matrix multiplication is associative, we can perform these multiplications in any order. We would like to know how many different ways we can perform these multiplications. For example, if $n = 3$, there are two possibilities:

$$(M_1 * M_2) * M_3$$
$$M_1 * (M_2 * M_3)$$

and if $n = 4$, there are five:

$$((M_1 * M_2) * M_3) * M_4$$
$$(M_1 * (M_2 * M_3)) * M_4$$
$$M_1 * ((M_2 * M_3) * M_4)$$
$$(M_1 * (M_2 * (M_3 * M_4)))$$
$$((M_1 * M_2) * (M_3 * M_4))$$

Let b_n be the number of different ways to compute the product of n matrices. Then $b_2 = 1$, $b_3 = 2$, and $b_4 = 5$. Let M_{ij}, $i \leq j$, be the product $M_i * M_{i+1} * \cdots * M_j$. The product we wish to compute is M_{1n}. We may compute M_{1n} by computing any one of the products $M_{1i} * M_{i+1,n}$, $1 \leq i \leq n$. The number of distinct ways to obtain M_{1i} and $M_{i+1,n}$ are b_i and b_{n-i}, respectively. Therefore, letting $b_1 = 1$, we have

$$b_n = \sum_{i=1}^{n-1} b_i \, b_{n-i}, \, n > 1$$

If we can determine the expression for b_n only in terms of n, then we have a solution to our problem.

Now instead let b_n be the number of distinct binary trees with n nodes. Again an expression for b_n in terms of n is what we want. Then we see that b_n is the sum of all the possible binary trees formed in the following way: a root and two subtrees with b_i and b_{n-i-1} nodes, for $0 \le i < n$ (Figure 5.50). This explanation says that

$$b_n = \sum_{i=0}^{n-1} b_i\, b_{n-i-1}\ ,\ n \ge 1\ ,\ \text{and } b_0 = 1 \tag{5.3}$$

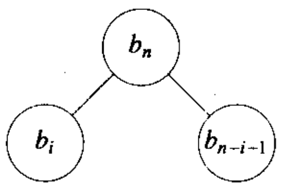


Figure 5.50: Decomposing b_n

This formula and the previous one are essentially the same. Therefore, the number of binary trees with n nodes, the number of permutations of 1 to n obtainable with a stack, and the number of ways to multiply $n + 1$ matrices are all equal.

5.11.4 Number of Distinct Binary Trees

To obtain the number of distinct binary trees with n nodes, we must solve the recurrence of Eq. (5.3). To begin we let

$$B(x) = \sum_{i \ge 0} b_i\, x^i \tag{5.4}$$

which is the generating function for the number of binary trees. Next observe that by the recurrence relation we get the identity

$$xB^2(x) = B(x) - 1$$

Using the formula to solve quadratics and the fact that $B(0) = b_0 = 1$ (Eq.(5.3)), we get

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

We can use the binomial theorem to expand $(1-4x)^{1/2}$ to obtain

$$B(x) = \frac{1}{2x}\left[1 - \sum_{n\geq 0} \begin{bmatrix}1/2\\ n\end{bmatrix} (-4x)^n\right] = \sum_{m\geq 0} \begin{bmatrix}1/2\\ m+1\end{bmatrix} (-1)^m\, 2^{2m+1}\, x^m \qquad (5.5)$$

Comparing Eqs. (5.4) and (5.5), we see that b_n, which is the coefficient of x^n in $B(x)$, is

$$\begin{bmatrix}1/2\\ n+1\end{bmatrix} (-1)^n\, 2^{2n+1}$$

Some simplification yields the more compact form

$$b_n = \frac{1}{n+1} \begin{bmatrix}2n\\ n\end{bmatrix} \sim O(4^n/n^{3/2})$$

EXERCISES

1. Prove that every binary tree is uniquely defined by its preorder and inorder sequences.
2. Do the inorder and postorder sequences of a binary tree uniquely define the binary tree? Prove your answer.
3. Do the inorder and preorder sequences of a binary tree uniquely define the binary tree? Prove your answer.
4. Do the inorder and level-order sequences of a binary tree uniquely define the binary tree? Prove your answer.
5. Write an algorithm to construct the binary tree with given preorder and inorder sequences.
6. Repeat Exercise 5 with the inorder and postorder sequences.
7. Prove that the number of distinct permutations of $1, 2, \cdots, n$ obtainable by a stack is equal to the number of distinct binary trees with n nodes. (Hint: Use the concept of an inorder permutation of a tree with preorder permutation $1, 2, \cdots, n$).

5.12 REFERENCES AND SELECTED READINGS

For more on trees, see *The Art of Computer Programming: Fundamental Algorithms*, Third Edition, by D. Knuth, Addison-Wesley, Reading, MA, 1998 and "Handbook of data structures and applications," edited by D. Mehta and S. Sahni, Chapman & Hall/CRC, Boca Raton, 2005.