4

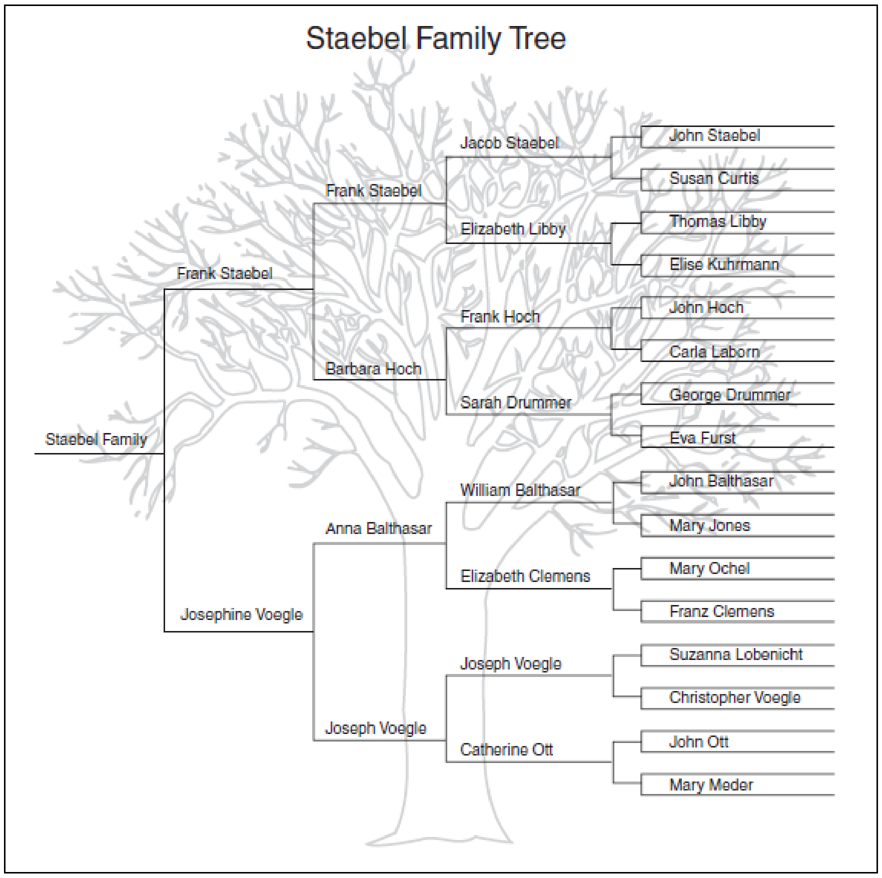
Trees

I think that I shall never see, a poem as lovely as a tree.

—Joyce Kilmer

Computer scientists, too, are enamored with trees. Trees, as defined in computer science and mathematics, sprout up everywhere in the subject of algorithms. Trees are often explicitly implemented in code for an algorithm or are implicitly present in the hierarchical logical organization of an algorithm.

You are no doubt familiar with tree-based hierarchical organizations such as family trees or organizational charts (see Figure 4.1). A table of contents in a book can be thought of as having a tree structure. The directory structure used by the operating system (UNIX, windows, Mac OS, or otherwise) on your computer for organizing files is a tree structure.



A family tree

Figure 4.1

Trees play multiple roles in the design and analysis of algorithms and in computing in general. Often it is very useful to store data in a tree-based hierarchical structure, since efficient access to the data is thereby facilitated. Imposing a tree structure on stored data will often lead to logarithmic complexity of operations versus the linear behavior that would result in viewing the data as a linear structure. The tree structure can either be explicitly or implicitly implemented. For example, we will see in section 4.6 how maintaining a priority queue using a tree-based structure known as a heap will allow addition and deletion of elements to be done in logarithmic time as opposed to the linear time that would result if the priority queue were maintained as a linear data structure. In this case the tree structure is actually overlaid implicitly on the data that is stored contiguously in a (linear) array. As another example, the binary search algorithm for searching an ordered list stored contiguously in an array is based on the implicit binary search tree associated with the data (midpoint element being the root, etc.) In other cases, the data is stored in an explicitly created tree (typically using dynamically allocated storage). For example, balanced tree structures such as red-black trees and B-trees (see Chapter 20) are often explicitly built and maintained to facilitate rapid key location in a database. Internet searches and other textual pattern matching algorithms are usually based on storing pattern strings in explicitly constructed trees, such as suffix trees or tries.

We saw in Chapter 2 that trees are also always implicitly present when modeling the resolution of recursive algorithms. Examining the tree of recursive calls implicitly generated by the complete resolution of a recursive algorithm is often the technique used to measure the complexity of the algorithm. Comparison and decision trees modeling the action of algorithms are other ways in which trees are implicitly present in algorithm analysis. For example, a comparison tree modeling the action of a comparison-based algorithm for sorting a list of size n is can be used to establish lower bounds for the complexity of the algorithm. These lower bounds follow from various mathematical properties of trees. Elementary properties of trees, such as depth, will be discussed in Section 4.6, whereas some more sophisticated properties, such as leaf path length, are the subject of Section 4.8.

In addition to all of the above ways that trees arise in the design and analysis of algorithms, their topological structure is frequently utilized as the basis for solving problems related to networks, such as the Internet, wired and wireless communication networks, computer networks, and so forth. For example, in Chapter 7 the fundamental problem of finding a minimal collection of links connecting the nodes in a network is solved by constructing a spanning tree of minimal cost. Another fundamental problem, discussed in Chapters 6 and 8, that of finding paths of shortest length from a given node in a connected network to all other nodes, is solved by constructing a shortest-path spanning tree.

4.1 Definitions

There are a thousand hacking at the leaves of evil to one who is striking at the root. —
Henry Thoreau

In this section we give the formal mathematical definition of trees, and establish some of their elementary properties. Our trees are actually *rooted* trees, but we will usually simply refer to them as trees. We begin our discussion by defining a special class of trees known as binary trees. These are simply trees where every node has at most two children.

4.1.1 Binary Trees

There are many equivalent definitions of a (rooted) binary tree, but the following recursive definition is certainly one of the most elegant.

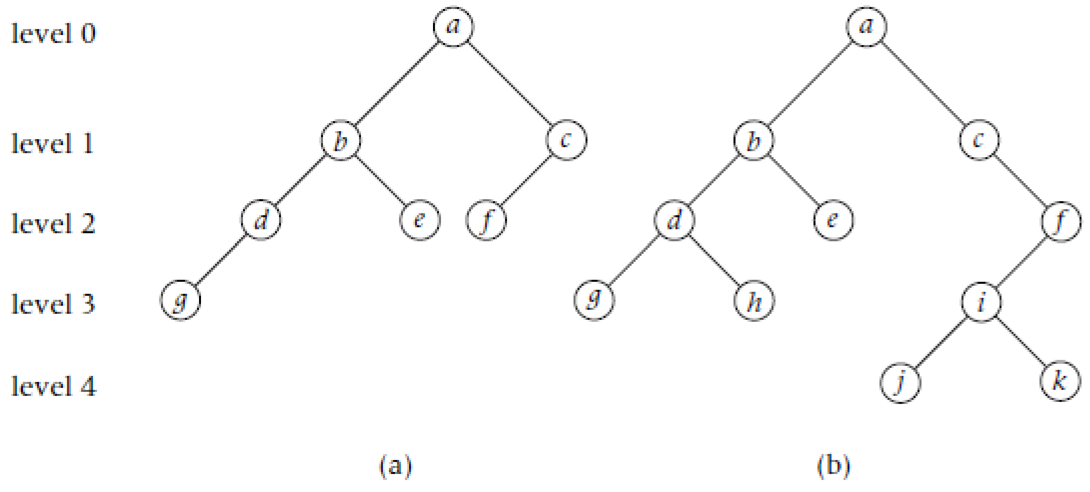
Definition 4.1.1

A binary tree T consists of a set of nodes, which is either empty or has the following properties:

1. One of the nodes, say *R*, is designated the root node.
2. The remaining nodes (if any) are partitioned into two disjoint subsets, called the left subtree and right subtree, respectively, each of which is a binary tree.

The roots of the left and right subtrees described in property (2) of the definition are called the left child and right child of *R*, respectively. Some examples of binary trees are given in Figure 4.2. A node is called a *leaf* if it has no children. Just as in family trees, we utilize genealogical notation such as *parent*, *grandchildren, great-grandchildren, sibling, descendant, ancestor*, and so forth.

The nodes of *T* can be partitioned into disjoint sets (levels) depending on their (genealogical) distance from the root *R*. In particular, the root *R* is at level 0. The *depth* (also called *height*) of *T* is the maximum distance from *R* to a leaf. The trees shown in Figure 4.2a and 4.2b have depths 3 and 4, respectively.



Sample binary trees of depths 3 and 4

Figure 4.2

4.1.1 General Trees

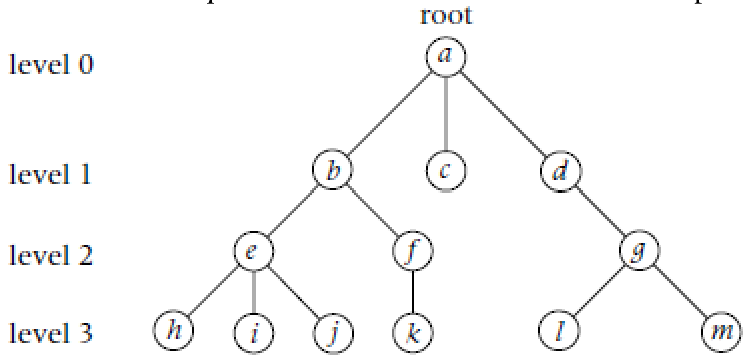
The design of many algorithms requires more general trees than binary trees. General trees are often implicitly present in the logical organization of an algorithm. The following definition extends the concept of a (rooted) binary tree given by Definition 4.1.1 to a general (rooted) tree.

Definition 4.1.2

A tree T consists of a set of nodes (also called vertices), which is either empty or has the following properties:

1. One of the nodes, say R, is designated the *root* node.
2. The remaining nodes (if any) are partitioned into j disjoint subsets $T_1, T_2, \dots, T_j$, each of which is a tree (called subtrees).

Given the root R, the roots of the subtrees described in (2) are called *children* of R and R is called the *parent*. A node is called a *leaf* if it has no children. It should be noted that when applying property 2 to a subtree, the value of j will, in general, vary with the subtree. Just as for binary trees, we utilize genealogical terminology such as *parent*, *grandchildren*, *great-grandchildren*, *sibling*, *descendant*, *ancestor*, and so forth. We also have the notions of *levels* and *depth* of a tree. We illustrate a tree of depth 3 in Figure 4.3.



A sample tree of depth 3

Figure 4.3

The unordered pair consisting of a node and its parent is referred to as an edge. We leave the following proposition as an exercise.

Proposition 4.1.1 The number of edges of a tree is one less than the number of nodes.

4.2 Mathematical Properties of Binary Trees

The analysis of the complexity of many problems and algorithms discussed in this text depends on various mathematical properties of binary trees. In this section we establish a number of these properties relating to depth, internal path length, and leaf path length.

Given a binary tree T, for ease of discussion, we denote the number of nodes, the number of leaf nodes, the number of internal nodes, and the depth by $n = n(T)$, $L = L(T)$, $I = I(T)$, and $d = d(T)$, respectively, Let n_i denote the number of nodes of T at level i, $i = 0, \ldots, d$. Since T is a binary tree, each node of T has at most two children, which yields the following recurrence relation:

$$n_i \le 2n_{i-1}, \quad i = 1, \ldots, d, \qquad \textbf{init.cond.}\ n_0 = 1. \qquad (4.2.1)$$

Thus, a simple induction yields:

$$n_i \le 2^i, i = 0, \ldots, d. \qquad (4.2.2)$$

For a given level $i \in \{1, \ldots, d\}$, we say that T is full at level i, if $n_i = 2^i$.

4.2.1 Complete and Full Binary Trees

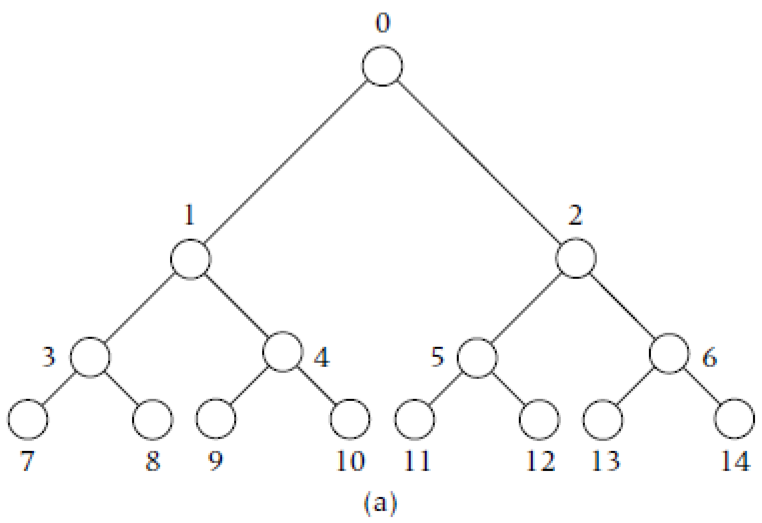
It can easily be verified that if a binary tree T is full at level i, then it is full at every level j smaller than i. A binary tree T is *full* if it is full at every level. Equivalently, a binary tree is full if it is full at the last level. Note that a full binary tree exists only for those n satisfying:

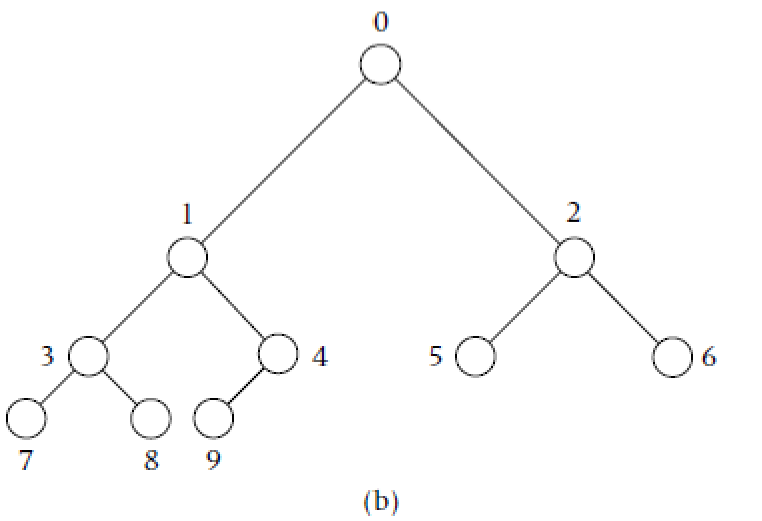
$$n = 1 + 2 + 2^2 + \cdots + 2^d = 2^{d+1} - 1. \qquad (4.2.1)$$

We denote the full binary tree on n nodes by T_n, $n = 1, 3, 7, 15, \ldots$. The full binary tree T_{15} is shown in Figure 4.4a. Solving for d in (4.2.1) yields the following formula for the depth of T_n

$$d = \log_2(n+1) - 1 = \lfloor \log_2 n \rfloor. \qquad (4.2.2)$$

Full binary trees are special cases of complete binary trees, where n is one less than a power of 2. To define the *complete* binary tree T_n for a general n, we let k be the (unique) positive integer such that $2^k - 1 < n \le 2^{k+1} - 1$. Then, for n different from $2^{k+1} - 1$, T_n is defined as a canonical subtree of the full binary tree T_n, $n = 2^{k+1} - 1$ having the same depth and differing from T_n only at the last level. The $q = n - (2^k - 1)$ nodes of T_n on the last level occupy the left-most q positions of T_n. More precisely, consider the following labeling of the nodes of T_n: label the root node 0. Inductively, label the left and right children of vertex i as $2i + 1$ and $2i + 2$, respectively. Then the complete binary tree T_n on n nodes is the subtree of T_n consisting of those nodes labeled $0, 1, \ldots, n - 1$ (see Figure 4.4b illustrating T_{10}).





a) Full binary tree T_{15}; b) complete binary tree T_{10}

Figure 4.4

Since the depth of T_n, $2^k - 1 < n \leq 2^{k+1} - 1$, is the same as the depth k of the full tree on $2^{k+1} - 1$ nodes obtained by filling out the last level, and since $k = \lfloor \log_2 n \rfloor$, we have the following proposition for the depth of an arbitrary complete tree on n nodes.

Proposition 4.2.1 The depth of the complete binary tree T_n for a general n is given by

$$d(T_n) = \lfloor \log_2 n \rfloor. \tag{4.2.3}$$

Since T_n is as full as possible for a binary tree on n nodes, it is not surprising that T_n has minimum depth for all binary trees on n nodes. The latter fact and other mathematical properties of binary trees are discussed in the next section.

Remark

The terminology for complete and full binary trees varies in the literature. For example, sometimes the name *complete* is used for what we have called *full* and the name *left complete* is used for what we have called *complete.*

4.2.2 Logarithmic Lower Bound for the Depth of Binary Trees

The following proposition establishes the useful and intuitively clear result that the depth of a binary tree having n nodes is at least the depth of the complete binary tree T_n on n nodes.

Proposition 4.2.2

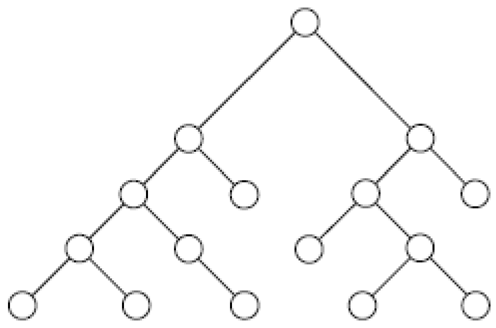
Suppose T is any binary tree having n nodes. Then T has depth at least $\lfloor \log_2 n \rfloor$, that is,

$$D(T) \geq \lfloor \log_2 n \rfloor. \tag{4.2.4}$$

The lower bound $\lfloor \log_2 n \rfloor$ is achieved for the complete binary tree T_n.

Proof The argument follows a similar pattern to the argument that led to (4.2.3), except inequalities are used in place of equalities. We leave the complete proof as an exercise.

For $n = 2^k - 1$, then T_n is the only binary tree whose depth achieves the lower bound of $\lfloor \log_2 n \rfloor$. For a general n (different from $n = 2^k - 1$), there are many binary trees on n nodes that achieve the lower bound depth of $\lfloor \log_2 n \rfloor$. In Figure 4.5, a tree having 16 nodes and depth 4 (= $\lfloor \log_2 n \rfloor$) is given, which is quite different from the complete tree T_{16}.



A noncomplete tree T with n = 16 having depth = $\log_2 16$ = 4

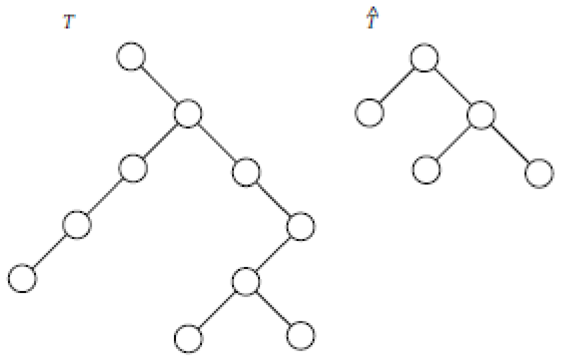
Figure 4.5

To establish various other mathematical properties of binary trees, we need to introduce the notion of a 2-tree.

4.2.3 2-Trees

A *2-tree* is a binary tree such that every node that is not a leaf has exactly two children. Any binary tree T can be canonically mapped to an associated 2-tree $\hat{T}$ having the same

number of leaf nodes and no larger depth. The 2-tree $\hat{T}$ is obtained from T as follows: If a node of T has one child, then we contract the edge joining this node with its child (so that the node is identified with its child). This process is repeated until we end up at the 2-tree $\hat{T}$ (see Figure 4.6). Clearly, $L(T) = L(\hat{T})$, and $d(T) \geq d(\hat{T})$.
The following proposition establishes a simple relationship between the number of internal nodes and the number of leaf nodes of a 2-tree.



A binary tree T and its associated 2-tree $\hat{T}$

Figure 4.6

Proposition 4.2.3 Suppose T is any 2-tree. Then the number of leaf nodes is one greater than the number of internal nodes of T; that is,

$$I(T) = L(T) - 1. \tag{4.2.5}$$

Equivalently, we have

$$n(T) = 2L(T) - 1. \tag{4.2.6} \square$$

Proposition 4.2.3 is easily proved by induction. Proposition 4.2.3 fails dramatically for binary trees in general, since an entirely skewed binary tree has only one leaf node.

Proposition 4.2.4

Suppose T is any binary tree. Then the depth of T satisfies

$$d(T) \geq \lceil \log_2 L(T) \rceil. \tag{4.2.7}$$

Proof Consider the 2-tree $\hat{T}$ associated with T. By Proposition 4.2.3, $n(\hat{T}) = 2L(\hat{T}) - 1$. Using Proposition 4.2.2 and the fact that $\lfloor \log_2(2n-1) \rfloor = \lceil \log_2 n \rceil$ for any integer $n > 1$, we have

$$d(\hat{T}) \geq \lfloor \log_2(2L(\hat{T}) - 1) \rfloor = \left| \log_2 L(\hat{T}) \right| .$$

Since $L(T) = L(\hat{T})$ and $d(T) \geq d(\hat{T})$, it follows that

$$d(T) \geq d(\hat{T}) \geq \left| \log_2 L(\hat{T}) \right| = \lceil \log_2 L(T) \rceil. \quad \blacksquare$$

Recall that a binary tree *T* is called full at level *i* if *T* contains the maximum number 2^i nodes at that level. A full binary tee *T* is full at every level *i*, $i = 0, \ldots, d(T)$. Proofs of the following two propositions are left to the exercises.

Proposition 4.2.5

Suppose *T* is a 2-tree. Then *T* is full at the second-deepest level if, and only if, all the leaf nodes are contained in two levels ($d - 1$ and d). □

Proposition 4.2.6

If a 2-tree *T* is full at the second-deepest level, then the depth $d(T)$ and the number of leaf nodes $L(T)$ are related by

$$d(T) = \lceil \log_2 L(T) \rceil. \qquad (4.2.8)$$ □

4.2.4 Internal and Leaf Path Lengths of Binary Trees

Inputs to search algorithms that are based implicitly (such as *BinarySearch*) or explicitly on binary trees usually follow paths from the root to an internal or leaf node. Moreover, the number of basic operations performed by the algorithm is usually proportional to the length of such a path. In particular, determining the average complexity of such algorithms requires computing the sum of the lengths of all paths from the root to nodes in the tree, which motivates the following definitions. The *internal path length IPL*(*T*) of a binary tree *T* is defined as the sum of the lengths of the paths from the root to the internal nodes as the internal nodes vary over the entire tree. The *leaf path length LPL*(*T*) is defined similarly. Note that the length of a path from the root to a node at level *i* is *i*.

When the binary tree *T* is a 2-tree, then *LPL*(*T*) is precisely 2*I* more than *IPL*(*T*). This simple relationship between *IPL*(*T*) and *LPL*(*T*) for 2-trees is stated as Proposition 4.2.7; its proof is left as an exercise.

Proposition 4.2.7

Given any 2-tree *T* having *I* internal nodes,

$$IPL(T) = LPL(T) - 2I. \qquad (4.2.9)$$ □

The following theorem establishes a useful lower bound for the leaf path length of a binary tree in terms of the number of leaf nodes.

Theorem 4.2.8

Given any binary tree *T* with *L* leaf nodes

$$LPL(T) \geq L\lfloor \log_2 L \rfloor + 2(L - 2^{\lfloor \log_2 L \rfloor}). \qquad (4.2.10)$$

Moreover, inequality (4.2.10) is an equality if, and only if, *T* is a 2-tree that is full at the second-deepest level.

Proof We first verify that inequality (4.2.10) is an equality if T is a 2-tree that is full at the second-deepest level. Suppose that $L = 2^k$, so that $\lfloor \log_2 L \rfloor = \log_2 L = k$. Then T is a full binary tree, and (4.2.10) correctly yields the equality $LPL(T) = L\log_2 L$. Now suppose $2^{k-1} < L < 2^k$ for a suitable $k > 1$. By Proposition 4.2.4, T has depth $d = \lceil \log_2 L \rceil$. Since T is full at level $d - 1 = \lfloor \log_2 L \rfloor$, each of the L paths in T from the root to a leaf reach level $d - 1$. Hence, the total contribution to $LPL(T)$ from reaching level $d - 1$ is $L\lfloor \log_2 L \rfloor$, so that

$$LPL(T) = L\lfloor \log L \rfloor + \text{the number of nodes at level } d\,.$$

If x denotes the number of nodes at level $d - 1$ having two children (non-leaf nodes), then $2x$ equals the number of nodes at level d. The remaining $2^{d-1} - x$ nodes at level $d - 1$ are leaf nodes, so that $L = (2^{d-1} - x) + 2x$. Thus, we have $x = L - 2^{d-1}$, and $LPL(T) = L\lfloor \log_2 L \rfloor + 2(L - 2^{\lfloor \log_2 L \rfloor})$.

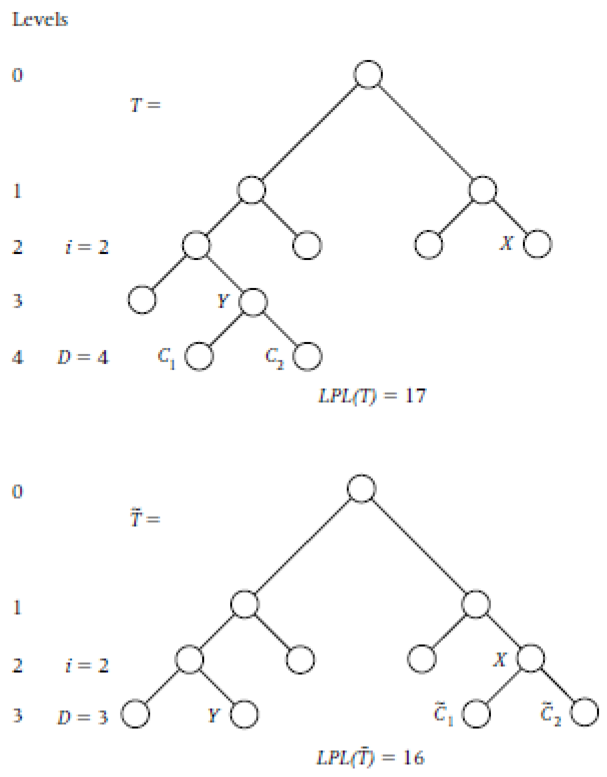
We now verify that inequality (4.2.10) holds for any binary tree. Clearly, if T is any binary tree with L leaf nodes, then its associated 2-tree $\hat{T}$ has L leaf nodes and $LPL(\hat{T}) \leq LPL(T)$. Thus, to complete the proof of Theorem 4.2.8, it is sufficient to show that $LPL(T)$ is not minimized for 2-trees T having L leaf nodes that are not full at the second-deepest level.

Suppose T is a 2-tree of depth d having L leaf nodes that is not full at level $d - 1$. We show that there then exists a 2-tree T having L leaf nodes such that $LPL(\tilde{T}) < LPL(T)$. Since level $d - 1$ is not full, there exists a leaf node, say X, at some level $i \leq d - 2$. Let Y be any node at level $d - 1$ that is not a leaf node, and let C_1 and C_2 denote the children of Y. We construct $\tilde{T}$ from T by removing the leaf nodes C_1 and C_2 and adding two new leaf nodes $\tilde{C}_1$ and $\tilde{C}_2$ as the children of X (see Figure 4.7). Let $L(T)$ and $L(\tilde{T})$ denote the set of all leaf nodes of T and $\tilde{T}$, respectively. Then,

$$L(\tilde{T}) = (L(TL) - \{C_1, C_2 X\}) \cup \{\tilde{C}_1, \tilde{C}_2, Y\}.$$

Since C_1, C_2, and X are at levels d, d, and i in T, respectively, and $\tilde{C}_1$, $\tilde{C}_2$, and Y are at levels $i + 1$, $i + 1$, and $d - 1$ in $\tilde{T}$, respectively, it follows that

$$LPL(\tilde{T}) = LPL(T) - (2d + i) + (2(i+1) + d - 1) = LPL(T) - d + i + 1 < LPL(T). \quad \blacksquare$$



Construction of $\tilde{T}$ from T used in the proof of Theorem 4.2.8

Figure 4.7

Corollary 4.2.9
If T is any binary tree having L leaf nodes, then

$$LPL(T) \geq \lceil L\log_2 L \rceil. \qquad (4.2.11)$$

Further, if T is a full binary tree, then inequality (4.2.11) is an equality.

Proof Inequality (4.2.11) follows immediately from inequality (4.2.10) of Theorem 4.2.8 by using the inequality

$$x\lfloor \log_2 x \rfloor + 2\left(x - 2^{\lfloor \log_2 x \rfloor}\right) \geq \lceil x\log_2 x \rceil, \text{ for any integer } x \geq 1.$$

Moreover, we have already observed in the proof of Theorem 4.2.8 that if T is a full binary tree, then $LPL(T) = L \log_2 L = \lceil L \log_2 L \rceil$. ■

4.2.5 Number of Binary Trees

For any nonnegative integer n, let b_n denote the number of different binary trees on n nodes. The following theorem, which is proved in Appendix D, gives a simple formula b_n.

Theorem 4.2.10
Let b_n denote the number of different binary trees on n nodes. Then,

$$b_n = \frac{1}{n+1}\binom{2n}{n}, \quad n \geq 1 \qquad (4.2.12)$$

The value $b_n = \frac{1}{n+1}\binom{2n}{n}, \quad n \geq 1$, occurs frequently in enumeration and is also known as the *n*th Catalan number. Formula (4.2.12), together with the lower bound estimate

$$\binom{2n}{n} \geq \frac{4^n}{(2n+1)}, \quad n \geq 1$$

(see Exercise 4.14), so that the number of binary trees on *n* nodes grows exponentially with *n*.

4.3 Implementation of Trees and Forests

4.3.1 Array Implementation of Complete Trees

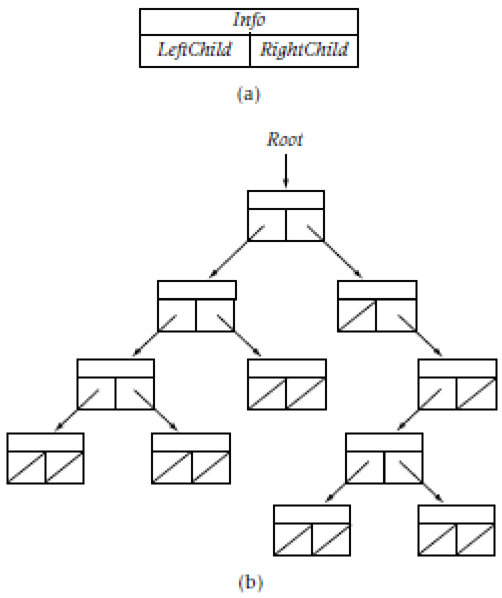
The labeling of the nodes of the complete binary tree T_n illustrated in Figure 4.4 is particularly useful when the tree is implemented using an array $T[0:n - 1]$, where $T[i]$ corresponds to the *i*th node in the labeling, $i = 0, \ldots, n - 1$. For example, in Section 4.6 we will see how the heap ADT is implemented very effectively using the array implementation of T_n. Writing code for algorithms implementing the creation of a heap, and deletion and insertion operations, is greatly facilitated using the fact that the children of the i^{th} node $T[i]$ are $T[2i + 1]$ and $T[2i + 2]$, and the parent of the *i*th node is $T[\lfloor(i - 1)/2\rfloor]$.

4.3.2 Implementing General Binary Trees Using Dynamic Variables

Binary trees can be implemented using pointer and dynamic variables as follows. The nodes of the tree are represented by a dynamic variable having the following structure.

```
BinaryTreeNode = record
    Info: InfoType
    LeftChild: → BinaryTreeNode
    RightChild: → BinaryTreeNode
end BinaryTreeNode
```

In addition to the dynamic variables representing the nodes of the binary tree, we have a pointer variable Root pointing to the root node. The implementation of the binary tree from Figure 4.2b using the pointer and dynamic variables is given in Figure 4.8.



(a) Representation of a node of a binary tree; (b) implementation of a binary tree using pointers and dynamic variables

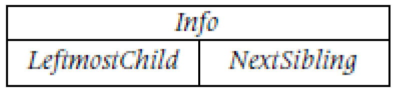
Figure 4.8

4.3.3 Child-Sibling Representation of a General Tree

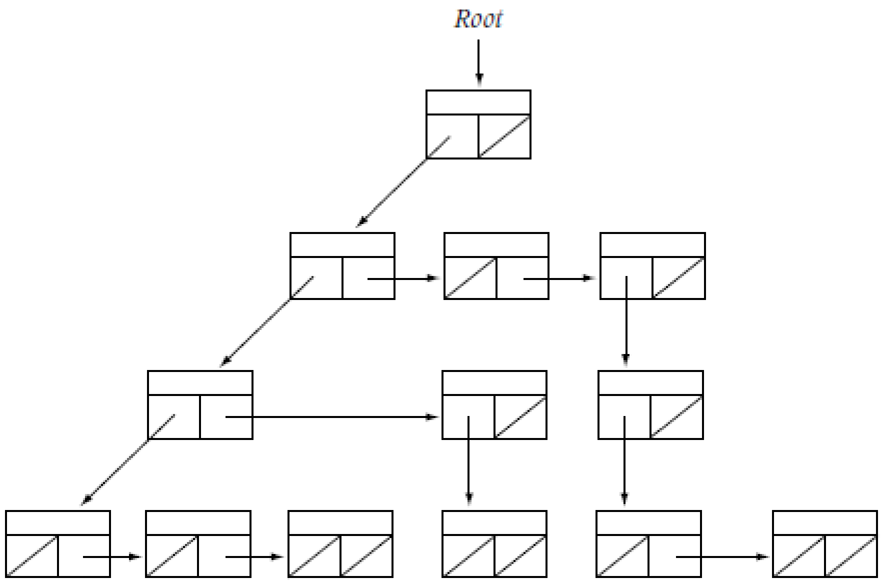
A child-sibling representation of a tree *T* can be implemented using pointer and dynamic variables as follows. The nodes of the tree are represented by the following structure.

```
TreeNode = record
    Info: InfoType
    LeftmostChild: → TreeNode
    NextSibling: → TreeNode
end TreeNode
```

As with binary trees, we keep a pointer variable Root that points to the root node of the tree. For example, the tree from Figure 4.3 is redrawn in Figure 4.9 using pointers and dynamic variables.



(a)



(a) Representation of a node of a general tree; (b) child-sibling implementation of a tree using pointers and dynamic variables

Figure 4.9

Remark

This implementation of a general tree leads naturally to a transformation (known as the Knuth transformation) from a general tree to a binary tree. We simply think of the *LeftmostChild* as the left child of the node and the *NextSibling* as the right child. Indeed, Figure 4.9, showing the general tree implementation of the tree from Figure 4.3, is actually a binary tree.

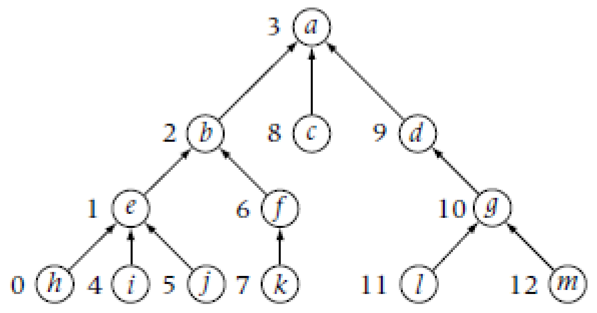
4.3.4 Parent Representation of Trees and Forests

Another representation of a general rooted tree, called the *parent representation*, keeps track of the parent of a node instead of its children. The parent representation has the advantage of efficiently generating the path in the tree from any given node of the tree to the root. However, the parent representation does not facilitate traversals of the tree.

The parent representation can actually be used to implement a slightly more general ADT known as a rooted forest. A *rooted forest* is a union F of (pairwise disjoint) rooted trees. A natural way to implement the parent representation of a (rooted) forest F uses an array $Parent[0{:}n-1]$. For convenience, we assume that F has n nodes labeled 0,1, . . . , n – 1. When referring to these nodes, we do not distinguish between a node and its label. For each $i \in \{0, 1, \ldots, n-1\}$, $Parent[i]$ is –1, if i is a root of one of the trees in F, and $Parent[i]$ is the parent of node i, otherwise. Information associated with the node i, $i = 0$, .

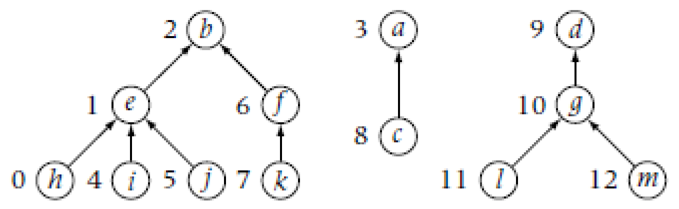
. . , $n - 1$, can be stored in a second array *Info*[0:n – 1]. In Figure 4.10*a* the values of the two arrays *Parent* and *Info* are given for the tree from Figure 4.3. The number beside each node of the tree in the diagram is the label of that node, and the character inside each node is the information stored in that node. In Figure 4.10*b* the parent array representation of a sample forest is given.

7,*Parent*[7] = 6,*Parent*[6] = 2,*Parent*[2] = 3



i	0	1	2	3	4	5	6	7	8	9	10	11	12
Info[*i*]	*h*	*e*	*b*	*a*	*i*	*j*	*f*	*k*	*c*	*d*	*g*	*l*	*m*
Parent[*i*]	1	2	3	–1	1	1	2	6	3	3	9	10	10

(a)



i	0	1	2	3	4	5	6	7	8	9	10	11	12
Info[*i*]	*h*	*e*	*b*	*a*	*i*	*j*	*f*	*k*	*c*	*d*	*g*	*l*	*m*
Parent[*i*]	1	2	–1	–1	1	1	2	6	3	–1	9	10	10

(b)

(a) Parent array implementation of a tree; (b) parent array implementation of a forest

Figure 4.10

The parent representation of a forest can also be implemented using pointers and dynamic variables. The nodes of the forest in this representation are given by the following structure.

```
ForestNode = record
   Info: Infotype
   Parent: → ForestNode
end ForestNode
```

The parent field of a node contains a pointer to the parent of that node if it has a parent. If the node corresponds to a root of one of the trees in the forest, then it contains the value **null**. The parent implementation of trees and forests has many important applications. Many of these applications exploit the fact that the parent implementation of a tree T allows efficient generation of the *path* P in T from any given node u of T to the root r. (The path in T from u to r is the sequence of nodes $u_0, u_1, \ldots, u_p$, where $u_0 = u$ and $u_p = r$, such that u_i is the parent of u_{i-1}, $i = 1, 2, \ldots, p$.) For example, in Figure 4.10a we generate the path in the indicated tree T from vertex 8 to the root 4 as follows:

$$8, Parent[8] = 7, Parent[7] = 3, Parent[3] = 4$$

4.4 Tree Traversal

When a tree is implemented as a data structure in an algorithm, the nodes of the tree contain information (data) germane to the application in question. In many applications it is necessary to access the fields in every node of the tree. During the execution of an algorithm, the current value of a variable (pointer variable, index variable, and so forth) that refers to a node in the tree thereby makes the fields in a particular node accessible. A node is visited when the node is accessed and some operation is performed on the information field(s) contained in the node. For example, visiting a node might consist in simply printing an information field. A tree is traversed if all its nodes are visited. The number of node accesses performed by a tree-traversal algorithm is used to measure the efficiency of the algorithm. (Normally, a node is visited only once in a traversal, but may be accessed several times.)

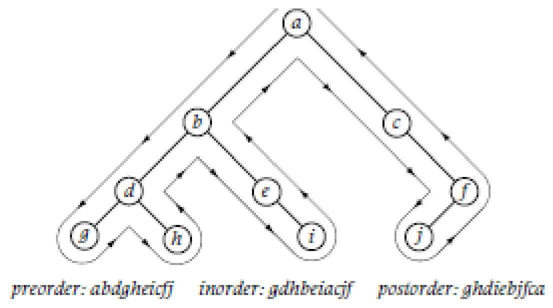
4.4.1 Traversing Binary Trees

We discuss three standard traversals of a binary tree T, namely, the preorder, inorder, and postorder traversals of T. Each traversal can be defined recursively, as illustrated by the following definition of preorder traversal.

Definition 4.4.1

The preorder traversal of T is defined recursively as follows:
If T is not empty, then
1. visit the root node R,
2. perform a preorder traversal of the left subtree LT of the root,
3. perform a preorder traversal of the right subtree RT of the root.

Note that no action is performed in the preorder traversal of an empty tree T. The inorder and postorder traversals of T differ from the preorder traversal only in the order in which the root is visited. In the inorder traversal the root is visited after the left subtree tree has been traversed. In the postorder traversal the root is visited last (see Figure 4.11).



Here visiting a node consists of printing the single character information field. All three traversals have the same *path around the tree* as indicated.

Figure 4.11

In Figure 4.11, the information in each node of the binary tree is a single alphabetical character. Assuming that visiting a node consists of simply printing the information stored in the node, then the resulting character strings printed out when the tree is traversed in preorder, inorder, and postorder are *abdgheicfj*, *gdhbeiacjf*, and *ghdiebjfca*, respectively.
All three traversals have the same "path around the tree," as shown in Figure 4.11. The difference between them is the order in which visits occur. Figure 4.12 illustrates the sequence of accesses and visits for the tree given in Figure 4.12 for each of the three traversals.

preorder traversal		inorder traversal		postorder traversal	
access		access		access	
a	visit	*a*		*a*	
b	visit	*b*		*b*	
d	visit	*d*		*d*	
g	visit	*g*	visit	*g*	visit
d		*d*	visit	*d*	
h	visit	*h*	visit	*h*	visit
d		*d*		*d*	visit
b		*b*	visit	*b*	
e	visit	*e*	visit	*e*	
i	visit	*i*	visit	*i*	visit
e		*e*		*e*	visit
b		*b*		*b*	visit
a		*a*	visit	*a*	
c	visit	*c*	visit	*c*	
f	visit	*f*		*f*	
j	visit	*j*	visit	*j*	visit
f		*f*	visit	*f*	visit
c		*c*		*c*	visit
a		*a*		*a*	visit

Accessing versus visiting for the binary tree given in Figure 4.11 for preorder, inorder and postorder

Figure 4.12

The traversals preorder, inorder, and postorder have many applications. For example, preorder and postorder traversals of algebraic expression trees generate *prefix* and *postfix* expressions, respectively (see Exercise 4.19). A postorder traversal can be used to free all the nodes in a dynamically allocated tree in order to delete the tree, whereas a preorder traversal in such a tree can be used to copy the tree. A preorder traversal of an appropriate binary tree can be used to generate a binary coding for a given alphabet for text compression purposes (see Chapter 7). Inorder traversals are particularly useful in connection with binary search trees. In particular, an inorder traversal of a binary search tree visits the nodes in sorted order, which is the basis for the sorting algorithm *Treesort* given in Section 4.5.6.

We now give a recursive procedure for a preorder traversal of the binary tree *T* implemented using pointers and dynamic variables, where the pointer variable *Root* points to the root of *T*. The procedure *Visit* performs some action with the data in *Info*, such as printing it out.

```
procedure Preorder(Root) recursive
Input: Root (pointer to the root node of a binary tree T)
Output:    the preorder traversal of T
    if Root ≠ null then
        Visit(Root→Info)
        Preorder(Root→LeftChild)
        Preorder(Root→RightChild)
    endif
end Preorder
```

The procedures *Inorder* and *Postorder* have pseudocode identical to *Preorder* except for the order of the call to *Visit*.

We now analyze the performance of *Inorder* (a similar analysis holds for *Preorder* and *Postorder*). The two main operations performed by *Inorder* are visiting a node (calls to *Visit*) and accessing a node. Since *Inorder* visits each node of *T* exactly once, the total number of node visits is n. During the performance of *Inorder*, each edge (line joining two nodes) in the three *T* is traversed exactly twice, in opposite directions. For each edge traversal, the endpoint node (in the direction of the traversal) of the edge is accessed. Since the number of edges of a tree is one less than the number of nodes, *T* has $n - 1$ edges. It follows that *Inorder* performs $2(n - 1)$ node accesses, plus one additional initial access of the root. Therefore, the total number of node accesses performed by *Inorder* (or by *Preorder* or *Postorder*) is

$$1 + 2(n - 1) = 2n - 1.$$

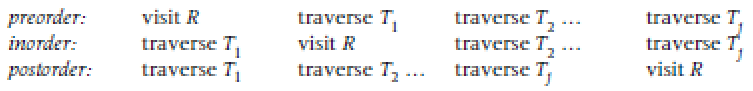
4.4.4 Traversing General Trees

The three standard traversals of a binary tree can be generalized to any tree as follows. In fact, if the tree is implemented using the leftmostchild-nextsibling representation, then these traversals simply reduce to the corresponding traversals of a binary tree. We can also define the preorder, inorder, and postorder traversals of a general *T* directly in terms of its recursive Definition 4.1.2.

Definition 4.4.2

The preorder traversal of *T* is defined recursively as follows:
If *T* is not empty, then
1. visit the root node *R*,
2. perform successively a preorder traversal of T_1, T_2, . . . , T_j.

Again, the postorder traversal is identical to the preorder traversal except that steps 1 and 2 are interchanged. The definition of the inorder traversal of a general *T* is not as natural as it was for binary trees, since there are many intermediate orders in which to perform a visit. In Figure 4.13 we formulate an inorder traversal that visits a node immediately after traversing the leftmost subtree of the node.



preorder:	visit R	traverse T_1	traverse T_2 …	traverse T_j
inorder:	traverse T_1	visit R	traverse T_2 …	traverse T_j
postorder:	traverse T_1	traverse T_2 …	traverse T_j	visit R

Preorder, inorder and postorder traversals of general tree *T*

Figure 4.13

Pseudocode for implementing preorder, inorder, and postorder traversals of general trees, as well as their generalizations to traversals of rooted forests, is developed in the exercises. As with binary trees, each of the standard traversals of general trees has its particular uses. For example, what amounts to a postorder traversal of a forest arises in an algorithm for determining the strongly connected components of the digraph (see Chapter 13). It turns out that a multivisit version of an inorder traversal is very useful in multiway search trees.

Traversals with Multivisits

For general trees, visiting a node might occur more than once in the course of a traversal. For example, an inorder traversal might actually take the following form:

inorder: traverse T_1 visit$_1R$ traverse T_2 visit$_2R$ . . . visit$_{j-1}R$ traverse T_j

where each visit of the node *R*, visit$_1R$, visit$_2R$, . . . , visit$_{j-1}R$ results in a (possibly) different operation performed on the information fields in *R*. We call this type of traversal a multivisit inorder traversal. Multivisit inorder traversals can be used to visit the keys in a multiway search trees as defined in Section 4.4.4.

4.5 Binary Search Trees

Given a totally ordered set *S* (elements of *S* will be called keys or identifiers), the *dictionary problem* is the well-known problem of designing an ADT to maintain a collection of items drawn from *S*. The classical dictionary problem restricts attention to the dictionary operations of inserting a key, deleting a key, and searching for a key, but we also might consider performing additional operations, such as finding the maximum or minimum elements or accessing the keys in sorted order.

In this section we introduce the binary search tree ADT and show how it supports the dictionary operations. In what follows we assume that the entire binary search tree is kept in internal memory, which is a reasonable assumption if the number of keys to be maintained is not too large. For larger sets of keys (such as typically encountered in databases stored in external memory) other ADTs such as B-trees are more appropriate (see Chapter 20). However, internal memory is rapidly becoming larger and cheaper, so that binary trees held in internal memory and containing keys from database records held in external memory are becoming more realistic.

Binary search trees are explicit generalizations of the implicit tree underlying the algorithm *BinarySearch*. The algorithm for inserting an element in a binary search tree is a variation of the searching strategy. Deletion is slightly more complicated, requiring in some cases a replacement of the deleted node by its inorder successor. We could also use the inorder predecessor for deletion, but for definiteness we always use the inorder successor. A useful property of a binary search tree is that an inorder traversal visits the nodes in sorted order.

4.5.1 Definition and Examples of Binary Search Trees

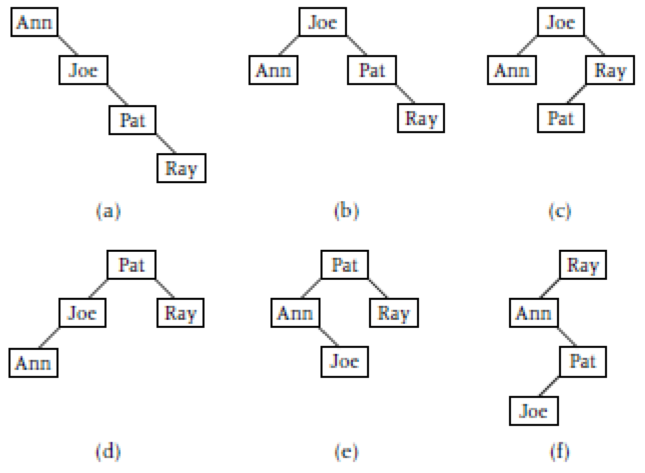
A binary search tree *T* with respect to keys in the nodes of *T* can be defined recursively as follows.

Definition 4.5.1

A binary tree *T* is called a binary search tree (with respect to a key field in each node) if *T* is empty, or if *T* has root *R*, and its right and left subtrees *RT*, *LT* satisfy the following.

1. Each key in *LT* is not larger than the key in *R*.
2. Each key in *RT* is not smaller than the key in *R*.
3. Both *LT* and *RT* are binary search trees.

Figure 4.14 gives 6 of the 24 different binary search trees for the set of four keys: ‘Ann’, ‘Joe’, ‘Pat’, ‘Ray’.



Sample binary search trees for the keys Ann, Joe, Pat and Ray

Figure 4.14

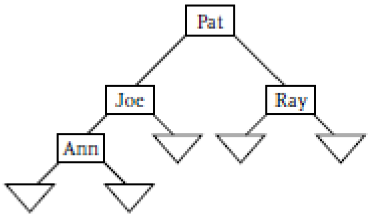
Given any set of *n* distinct keys (identifiers) and an arbitrary binary tree *T* on n nodes, there is a unique assignment of the keys to the nodes in *T* such that *T* becomes a binary search tree for the keys (see Exercise 4.29). Hence, the number of binary search trees containing a given set of *n* distinct keys equals the number $b_n = \frac{1}{n+1}\binom{2n}{n}$ of binary trees on *n* nodes.

4.5.2 Searching a Binary Search Tree

A simple and efficient algorithm (*BinSrchTreeSearch*) exists for locating information stored in binary search trees. For example, if we wish to determine which node in the tree (if any) has its key equal to a search element *X*, then we first compare *X* to the key contained in the root. If *X* equals this key, we are done. Otherwise, if *X* is less than the key in the root, then we search the left subtree, else we search the right subtree. The following algorithm *BinSrchTreeSearch* based on this simple process returns a pointer *Location* to a node where a search element *X* is found. If *X* is not found in the tree, then *BinSrchTreeSearch* sets *Location* to a node in the tree where *X* can be inserted as a child and still maintain a search tree. In the pseudocode for *BinSrchTreeSearch*, we assume that the nodes in the tree have the structure as described in Section 4.3.2.

```
procedure BinSrchTreeSearch(Root,S,Location,Found)
Input: Root (→BinaryTreeNode)//points at root of a binary search tree
          X (KeyType)
Output:   Location (→BinaryTreeNode) //points at occurrence of X, if any
          Found(Boolean)    //.false. if X not in tree, .true. otherwise
    Found ← .false.
    Location ← null
    Current ← Root
    while $Current \neq$ null .and. .not. Found do
        Location ← Current
        if $X = Current \rightarrow Key$ then
            Found ← .true.
        else
            if $X < Current \rightarrow Key$ then
                Current ← Current→LeftChild
            else
                Current ← Current→RightChild
            endif
        endif
    endwhile
end BinSrchTreeSearch
```

In Figure 4.15 we have not drawn the (implicit) leaf nodes corresponding to unsuccessful searches. The tree in Figure 4.14d is redrawn in Figure 4.15 with leaf nodes corresponding to unsuccessful searches added. For example, a search for 'Mary' would end up at the right child of 'Joe'. The latter node is where the key 'Mary' can be inserted and still maintain the binary search tree property.



Search tree with leaf nodes drawn representing unsuccessful searches

Figure 4.15

Inserting a node in a binary tree is similar to searching. In the exercises we develop algorithms for inserting and deleting from a binary search tree.

4.5.5 Multiway Search Trees

Multiway search trees generalize binary search trees by allowing more than one key to be stored in a given node. Thus, multiway search trees allow multiway branching to occur at a given node, instead of the two-way branching allowed by binary trees. The keys in a given node of a multiway search tree are maintained as an ordered list. During a search of a multiway search tree, when a search element reaches a given node, a search of the keys in the node is performed. (The latter search is usually a linear search, but if the keys are maintained in an array, then binary search might be used if the node contains a sufficiently large number of keys.) Either the search element equals one of the keys, or a branch is made to an appropriate child's subtree (which contains the key if it is in the tree at all), where the search continues. The following recursive definition places the fairly obvious restriction on the keys in a multiway search tree that must hold to support the natural generalization of the search strategy for binary search trees. In the definition, the value of j varies with the particular node in T.

Definition 4.5.2

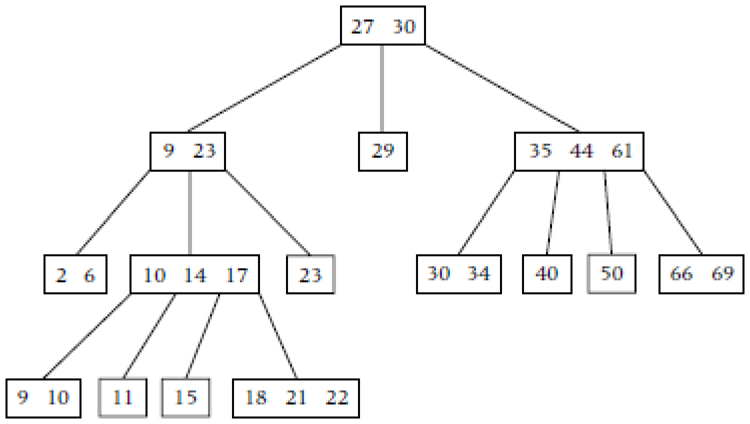
A multiway search tree T is either empty or consists of a root R containing j keys, $key_1 \leq key_2 \leq \ldots \leq key_j$, $j \geq 1$, and a collection of subtrees $T_1, T_2, \ldots, T_{j+1}$ having the following two properties:

1. for any $k_i \in T_i$, $1 \leq i \leq j + 1$, we have

$$k_1 \leq key_1 \leq k_2 \leq key_2 \leq \cdots \leq k_j \leq key_j \leq k_{j+1},$$

2. every T_i is a multiway search tree.

For a given fixed constant m, multiway search trees for which the maximum number of keys in a given node is bounded above by $m - 1$ are called *m-way* search trees. For example, a four-way search tree on 15 nodes is shown in Figure 4.16. Note that a binary search tree is a two-way search tree.



A four-way search tree on 15 nodes

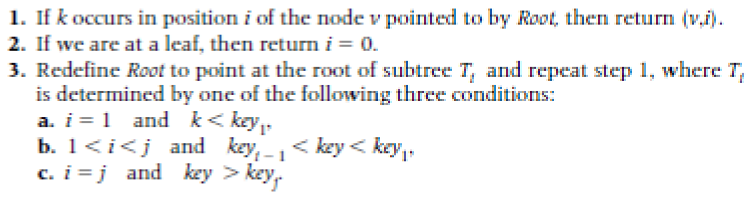
Figure 4.16

Remarks

A commonly used multiway search tree is the B-tree. B-trees are subject to several additional restrictions, including bounds on the minimum and maximum number of keys allowed in each node. B-trees also satisfy a more restrictive condition than (2), namely:
2.′ Either *every* T_i is empty, or *every* T_i is a nonempty multiway search tree.

We adopt the more general condition (2) in our definition of multiway search trees so that multiway search trees generalize the notion of binary search trees.

Figure 4.17 contains a high-level procedure determining a location in the multiway search tree T where the key k is found. In our high-level description, we do not specify how the tree T is implemented. The location of k consists of a pair (v,i), where v is a node of *T* in which *k* occurs, and *i* is a position of the occurrence of *k* in the ordered list of keys of *v*. If *k* does not occur in the tree, we set $i = 0$ and regard *v* as undefined. We initially perform step 1 in Figure 4.17 with a variable *Root* pointing at the root of *T*. Before repeating step 1, *Root* is redefined to point to the appropriate subtree being searched.



1. If *k* occurs in position *i* of the node *v* pointed to by *Root*, then return (v,i).
2. If we are at a leaf, then return $i = 0$.
3. Redefine *Root* to point at the root of subtree T_i and repeat step 1, where T_i is determined by one of the following three conditions:
- **a.** $i = 1$ and $k < key_1$,
- **b.** $1 < i < j$ and $key_{i-1} < key < key_1$,
- **c.** $i = j$ and $key > key_j$.

High-level description of procedure for search a multiway search tree

Figure 4.17

4.5.6 Treesort

The algorithm tree sort is based on the following interesting key fact.

Key Fact An inorder traversal of a binary search tree visits the nodes in nondecreasing order of the keys.

You should convince yourself of the validity of this key fact (a formal proof uses mathematical induction, see Exercise 4.35). Thus, given a list $L[0:n-1]$, we can sort the list by creating a binary search tree *T* whose keys are the elements of *L*, and then performing an inorder traversal of *T*. The algorithm *CreateBinSrchTree* for creating a binary tree simply performs *n* successive insertions of $L[0], L[1], \ldots, L[n-1]$, respectively. In the case where the list is already sorted, *CreateBinSrchTree* creates a completely skewed tree and has worst-case complexity given by:

$$W(n) = 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2).$$

However, in spite of the quadratic worst-case complexity of *CreateBinSrchTree*, it turns out that its average complexity belongs to $\Theta(n\log n)$. Since an inorder traversal of a tree is linear, tree sort also has quadratic worst-case complexity, and $\Theta(n\log n)$ average complexity.

The following proposition extends to multiway search trees the useful property that an inorder traversal of a binary search tree visits the nodes in sorted order. In our multivisit inorder traversal, if node *v* contains *q* keys, then *Visit*(v,i) processes the *i*th key in the sequential ordering of the keys in the node, $i = 1, \ldots, q$.

Proposition 4.4.1

A multivisit inorder traversal of a multiway search tree visits the nodes in sorted order. $\square$

A formal proof of Proposition 4.4.1 is left to the exercises.

4.6 Priority Queues and Heaps

A *priority queue* is a collection of elements each having a priority value, and such that when an element is deleted from the collection, an element of highest priority is always chosen for deletion. (Note that a priority queue is *not* a queue; that is, is not a FIFO list, but this ambiguity has become standard terminology.) One way to maintain a priority queue would be as a list ordered by the priority values, so that the highest priority element would be at the end of the list. Then deletion of the element would consist of a single list operation, and have $\Theta(1)$ complexity. However, insertion of an element would have $\Theta(n)$ complexity in the worst case. Similarly, a priority queue could simply be maintained as an unordered list, in which case deletion would require a linear scan to find an element of highest priority, and deletion would now have $\Theta(n)$ complexity in the worst case. Insertion, however, would now have $\Theta(1)$ complexity. It would be nice to find a data structure that would have, say, logarithmic complexity for both operations of deletion and insertion. The data structure known as a heap does the job.

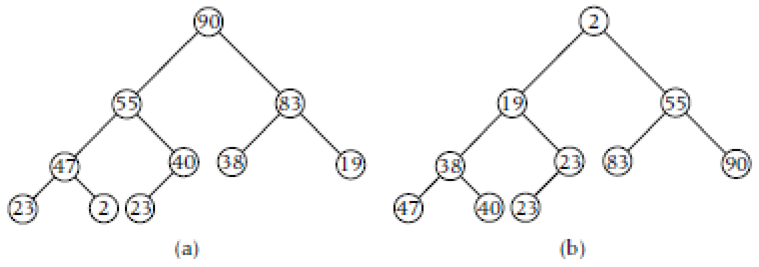
4.6.1 Heap Implementation of a Priority Queue

The underlying structure of a heap is a complete binary tree whose root contains the highest priority element.

Definition 4.6.1

A *max-heap* (*min-heap*) *H* is a complete binary tree whose information fields contain values (keys) having the following *max-heap* (*min-heap*) *property*: Given any node *v* in *H*, the value of *v* is not smaller (not larger) than the value of any node in the subtree having *v* as a root.

For convenience of notation, throughout the remainder of this section we discuss *max-heaps* and sometimes simply use the word *heap* for *max-heap*. A completely analogous discussion can be given for *min-heaps*. (See Figure 4.18.)



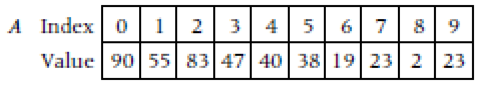
a) Max-heap; b) min-heap

Figure 4.18

Key Fact

Clearly, the max-heap property can be equivalently expressed by requiring that the value of each node *v* is not smaller than the value of its left child or right child (if they exist). This allows us to localize the heap property to each node. Thus, we can talk about the heap property holding or not holding at a given node.

A very convenient implementation of a heap results from storing the *n* key values in an array $A[0{:}m - 1]$, where $m \geq n$. The convenience of the array implementation results from the ease in which we can reference the children of a node, as well as its parent. Indeed, given a node of index *i*, its left child has index $2i + 1$ and its right child has index $2i + 2$ (assuming that these children exist). Also, if $i > 0$, the parent of a node of index *i* is given by $\lfloor (i - 1) /2 \rfloor$.
Figure 4.19 shows the max-heap given in Figure 4.18 stored in an array $A[0{:}m - 1]$, $m \geq 10$.



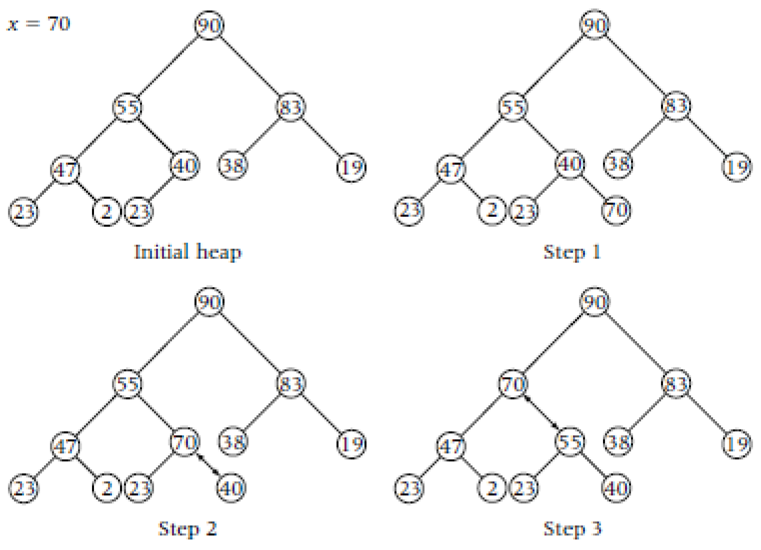
A	Index	0	1	2	3	4	5	6	7	8	9
	Value	90	55	83	47	40	38	19	23	2	23

Max-heap stored in array $A[0{:}m - 1]$, $m \geq 10$.

Figure 4.19

4.6.2 Inserting into a Heap

We now describe an algorithm for inserting a new element into an existing heap. We suppose that the existing heap occupies positions $A[0], \ldots, A[\mathit{HeapSize} - 1]$ within the array $A[0{:}m - 1]$. The idea is to initially place the new element in position $A[\mathit{HeapSize}]$ and let it "rise" along the unique path from $A[\mathit{HeapSize}]$ to the root until it finds a proper position. Letting $i = \mathit{Heapsize}$, we begin by comparing *x* to the value of the parent node $A[\lfloor (i - 1)/2 \rfloor]$. If *x* has a larger value than its parent, we move the parent down to position *i* and move *x* up to the (old) position of its parent. The argument is then repeated until *x* has been moved up the path to a proper position. We illustrate this movement in Figure 4.20, where we insert the value $x = 70$ into the max-heap given in Figure 4.18.



Inserting the value $x = 70$ into the existing heap $A[0:9]$

Figure 4.20

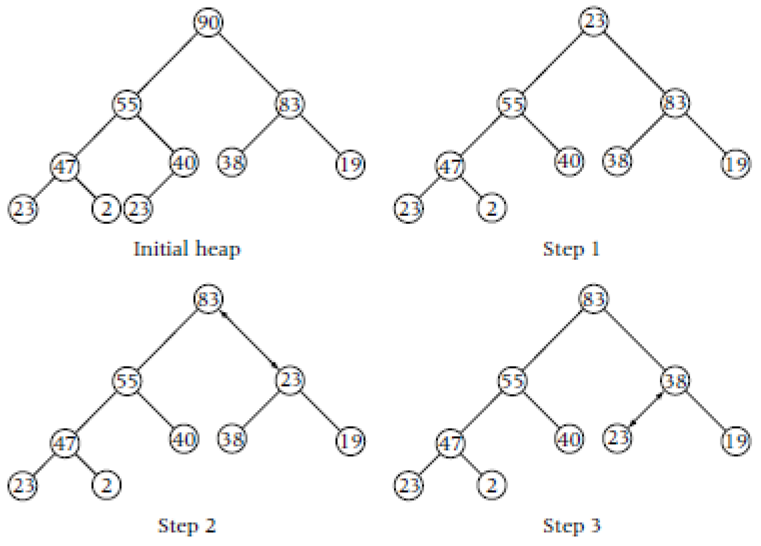
The pseudocode for the insertion operation follows. Note that we do not need to make an assignment of x until a proper place for its insertion into the heap is found.

```
procedure InsertMaxHeap(A[0:m – 1],HeapSize,x)
Input: A[0:m – 1] (an array containing a heap in positions 0, . . . , HeapSize – 1)
            HeapSize (the number of elements in the heap)
            x (a value to be inserted into heap)
Output:  A[0:m – 1] (array altered by addition of x and heap maintained)
            HeapSize (input HeapSize + 1)
    i ← HeapSize
    j ← ⌊(i – 1)/2⌋
    //traverse path to root until proper position for inserting x is found
    while j ≥ 0 .and. A[j] < x do
        A[i] ← A[j]              //move parent down one position in path
        i ← j                    //move x up one position in path
        j ← ⌊(j – 1)/2⌋
    endwhile
    //i is now a proper position for x
    A[i] ← x
    HeapSize ← HeapSize + 1
end InsertMaxHeap
```

Using comparisons to an element in the heap as our basic operation, the worst-case complexity for *InsertMaxHeap* occurs when the inserted element must rise all the way to the root. Thus, the worst-case complexity of *InsertMaxHeap* belongs to $\Theta(\log n)$.

4.6.3 Deleting from a Heap

The other basic operation for a priority queue that we need to implement is to remove the highest priority element. For our heap implementation, this element is at the root. The removal of the root is accomplished by elevating the bottom heap element $x = A[HeapSize - 1]$ to the root and then letting x "fall" down an appropriate path in the complete binary tree $A[0{:}HeapSize - 2]$ until it finds a proper position. Initially we compare x to the larger of the two children $A[1]$ and $A[2]$. If x is smaller, the larger child is elevated to the root (made "king of the heap") and x moves down to the position formerly occupied by the elevated child. The argument is then repeated until a proper position for x is found. We illustrate this in Figure 4.21 by removing the root from the max-heap given in Figure 4.20.



Removing the root from a heap

Figure 4.21

The adjustment operation illustrated in steps 1 through 3 in Figure 4.21 can be applied to the subtree rooted at any node having index i in a complete binary tree $A[0{:}n - 1]$, where the subtrees of $A[0{:}n - 1]$ with roots at the two children $2i + 1$ and $2i + 2$ are both heaps. The algorithm *AdjustMaxHeap* shows how to let $A[i]$ follow an appropriate path in the subtree with i as a root so as to make a heap out of this subtree.

```
procedure AdjustMaxHeap($A[0{:}m - 1],n,i$)
Input: $A[0{:}m - 1]$ (an array)
            $n$ (consider $A[0{:}n - 1]$ as complete binary tree, $n \le m$)
            $i$ (index where subtrees of $A[0{:}n - 1]$ rooted at $2i + 1$and $2i + 2$ are heaps)
Output:     $A[0{:}m - 1]$ (subtree of $A[0{:}n - 1]$ rooted at $A[i]$, adjusted so that it becomes a
            heap)
    $Temp \leftarrow A[i]$
    //traverse down a path until a proper position for Temp = A[i] is found
    Found ← .false.         //Found signals when a proper position is found
    j ← 2*i + 1                   //j is the path finder. At completion of loop,
                                  //⌊(j –1)/2⌋ is a proper position for Temp = A[i]
    while j ≤ n .and. .not. Found do
        if j < n – 1.and. A[j] < A[j + 1] then //then move to right child j + 1
            j ← j + 1         //path finder updated to right child
        endif
        if Temp ≥ A[j] then
            Found ← .true.
        else
            A[⌊(j – 1)/2⌋] ← A[j]   //move larger child up one position in path
            j ← 2*j + 1     //move path finder to next possible position for Temp
        endif
    endwhile
    A[⌊(j – 1)/2⌋] ← Temp
end AdjustMaxHeap
```

Clearly, the worst-case complexity of *AdjustMaxHeap* occurs when the element *A*[*i*] must fall all the way down to the bottom of the heap. Thus, when calling *AdjustMaxHeap* with the parameter *i*, in the worst case we make $\log_2 n - \log_2 i$ comparisons.
The operation of removing the root from a heap is accomplished by the following pseudocode.

```
procedure RemoveMaxHeap(A[0:m – 1],HeapSize,x)
Input: A[0:m – 1] (an array containing a heap in positions 0, . . . , HeapSize – 1)
            HeapSize (the number of elements in the heap)
Output:   A[0:m – 1] (array altered by removal of the root A[0] and heap maintained)
            x (assigned the value of the (original) root A[0])
            HeapSize (input HeapSize – 1)
    x ← A[0]
    HeapSize ← HeapSize – 1
    A[0] ← A[HeapSize]
    AdjustMaxHeap(A[0:m – 1],HeapSize,0)
end RemoveMaxHeap
```

The worst-case complexity of *RemoveMaxHeap* belongs to Θ(log *n*), so that we have implemented both the insertion of an element and the removal of the highest priority element with worst-case complexity Θ(log *n*).

4.6.4 Creating a Heap

Given *n* values in an array *A*[0:*m* – 1], the question arises as to how to make *A*[0:*n* – 1] into a heap. The most obvious way is to simply sequentially invoke *InsertMaxHeap*(*A*[0:*m* – 1],*HeapSize*,*x*) for *x* = *A*[0], . . . , *A*[*n* – 1]. In fact, this method is well suited to applications where the heap is maintained dynamically, with new elements being added to an already existing heap.

```
procedure MakeMaxHeap1(A[0:m – 1],n)
Input: A[0:m – 1] (an array)
          n (values in A[0:n – 1] considered as complete binary tree, n ≤ m)
Output:   A[0:m – 1] (A[0:n – 1] made into a heap)
   HeapSize ← 0
   for j ← 0 to n – 1
      InsertMaxHeap(A[0:m – 1],HeapSize,A[j])
   endfor
end MakeMaxHeap1
```

Figure 4.22 shows the action of *MakeMaxHeap1* on the set of elements (10, 5, 20, 30, 25, 7, 40). We show the result after each iteration of the **for** loop.

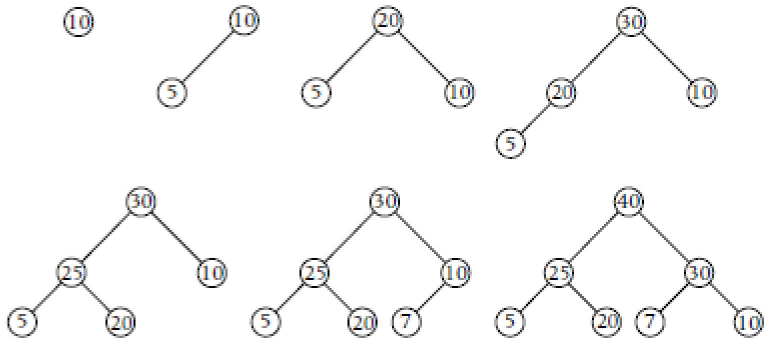

Action of *MakeMaxHeap1* on the set of elements (10, 5, 20, 30, 25, 7, 40)

Figure 4.22

Since the worst-case complexity of *InsertMaxHeap* belongs to O(log n), the worst-case complexity W(n) of *MakeMaxHeap1* belongs to O($n$log n). The worst case occurs when the elements are in increasing order, so that each element must rise to the root when it is inserted. We now show that W(n) belongs to Ω($n$log n). Let $k = \lfloor \log_2 n \rfloor$ and recall that a complete binary tree on n elements has 2^i nodes at level i for $i = 0, \ldots, k - 1$. Since we make i comparisons for each node at level i, and applying identity (1.2.11), we have

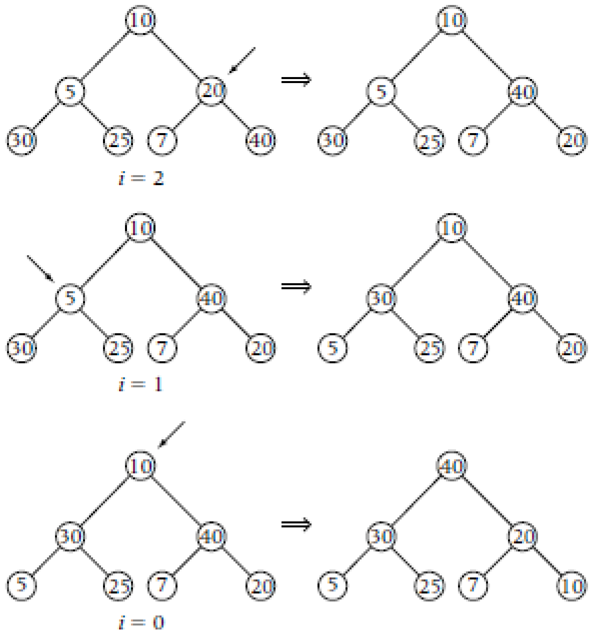
$$W(n) > \sum_{i=0}^{k-1} i2^i = 2^k(k-2) + 2 \in \Omega(n \log n).$$

Hence, W(n) belongs to Ω(n log n) ∩ O(n log n) = Θ(n log n).

An alternative algorithm *MakeMaxHeap2* for creating a heap has worst-case complexity in O(n). *MakeMaxHeap2* works by repeated calls to *AdjustMaxHeap*(A[0:m – 1],n,i). Recall that *AdjustMaxHeap*(A[0:m – 1],n,i) requires that the subtrees in A[0:n – 1] with roots $2i + 1$ and $2i + 2$ are already heaps. This will certainly be true if these latter two roots are leaf nodes of $A[0{:}n-1]$. Thus, we can make a heap by initially calling *AdjustMaxHeap*($A[0{:}m-1],n,i$) with $i = \lfloor (n-1)/2 \rfloor$ and then sequentially calling *AdjustMaxHeap*($A[0{:}m-1],n,i$) as i is decremented by one each time, until we finally reach the root.

```
procedure MakeMaxHeap2(A[0:m – 1],n)
Input: A[0:m – 1] (an array)
          n (values in A[0:n – 1] considered as complete binary tree, n ≤ m)
Output:  A[0:m – 1] (A[0:n – 1] made into a heap)
   for i ← ⌊(n – 1)/2⌋ down to 0 do
      AdjustMaxHeap(A[0:m – 1],n,i)
   endfor
end MakeMaxHeap2
```

The action of *MakeMaxHeap2* is illustrated in Figure 4.23 on the set (10, 5, 20, 30, 25, 7, 40), which was the same set illustrated in Figure 4.22 for *MakeMaxHeap1*. Note that the heaps created are rather different, but both satisfy the heap property. In Figure 4.29 we show the array before and after each call to *AdjustMaxHeap*. We indicate in each "before" picture the particular subtree that is being made into a heap.



Making a max-heap using *MakeMaxHeap2*

Figure 4.23

During the execution of the call to *AdjustMaxHeap*($A[0{:}m-1],n,i$), an element at level j will fall at most $k-j$ levels, where $k = \lfloor \log_2 n \rfloor$. Again, for $j < k$, there are exactly 2^j nodes at level j. We begin calling *AdjustMaxHeap*($A[0{:}m-1],n,i$) with $i = \lfloor (n-1)/2 \rfloor$, which is (the index of) a node at level $k – 1$. Hence, the worst-case complexity of *MakeMaxHeap2* satisfies

$$W(n) \le \sum_{i=0}^{k-1} 2^i (k-i) = \sum_{i=1}^{k} i2^{k-i} = 2^k \sum_{i=1}^{k} \frac{i}{2^i}.$$

Applying identity (1.2.11) with $x = 1/2$, we obtain

$$W(n) \le 2^k \left\lfloor 2 - \frac{k+2}{2^k} \right\rfloor = 2^{k+1} - (k+2) \le 2n \in O(n).$$

Since *MakeMaxHeap2* does $\lfloor n/2 \rfloor$ comparisons in the best case (when we have a heap at the outset), we have shown that the best-case, worst-case, and average complexities of *MakeMaxHeap2* all belong to $\Theta(n)$.

4.6.5 Sorting by Selection: Selection Sort versus Heap Sort

Certainly one of the simplest strategies for sorting a list $L[0:n – 1]$ of size n in nondecreasing order is as follows. Select the largest element in L and interchange it with $L[n – 1]$. Then select the largest element in the sublist $L[0:n – 2]$ and interchange it with $L[n – 2]$. In general, in the ith step, select the largest element in the sublist $L[0:n – i]$ and interchange it with $L[n – i]$, $i = 1, . . . , n – 1$. Clearly, this strategy yields a sorted list. The most straightforward strategy for selecting the largest elements is to make a linear scan as in the procedure *Max*. The resulting algorithm is known as selectionsort. Clearly, selectionsort performs $(n – 1) + (n – 2) + . . . + 1 = n(n – 1)/2$ comparisons for any input list of size n, so that its worst-case, best-case, and average complexities are all in $\Theta(n^2)$.

The inefficiency of selectionsort results from the procedure used to select the largest elements. Our selection procedure amounted to regarding the list as a priority queue, where the priority is the value (key) of the list element. The algorithm heapsort emulates selectionsort except that the priority queue of list elements is maintained as a heap, thereby reducing the complexity from $\Theta(n^2)$ to $\Theta(n\log n)$.

Procedure *HeapSort* utilizes the algorithm *AdjustMaxHeap* as follows. We first invoke *MakeMaxHeap2* to create the heap $A[0:n – 1]$. Then we interchange the root with the element at the bottom of the heap (position n) and invoke *AdjustMaxHeap*($A[0:m – 1],n–1,1$). The new root is now interchanged with the element at position $n – 1$, and the process is continued. The code for *HeapSort* follows.

```
procedure HeapSort(A[0:n – 1])
Input: A[0:n – 1] (a list of size n)
Output:    A[0:n – 1] sorted in nondecreasing order
    MakeMaxHeap2(A[0:n – 1],n)
    for i ← 1 to n – 1 do
        //interchange A[0] with A[n – i]}
        interchange (A[0],A[n – i])
        AdjustMaxHeap(A[0:n – 1],n – i + 1,0)
    endfor
end HeapSort
```

The call to *MakeMaxHeap2* has best-case and worst-case complexities $\Theta(n)$, whereas the $n - 1$ calls to *AdjustMaxHeap* each have $\Theta(1)$ best-case and $\Theta(\log n)$ worst-case complexities. Thus, *HeapSort* has $\Theta(n)$ best-case complexity and $\Theta(n \log n)$ worst-case complexity. *HeapSort* is a comparison-based sorting algorithm, so by lower bound theory (see Chapter 25), it has $\Omega(n \log n)$ average complexity, which implies that *HeapSort* has $\Theta(n \log n)$ average complexity.

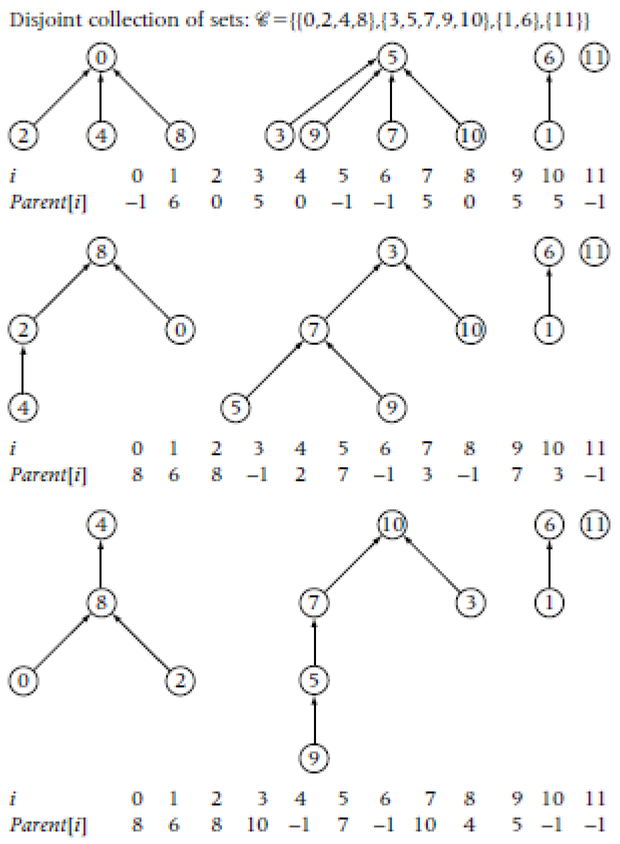
4.7 Implementing Disjoint Sets

Many algorithms, particularly graph algorithms, involve maintaining a (dynamically changing) equivalence relation on some universal (or base) set $S = \{s_1, s_2, \ldots, s_n\}$. For example, an efficient implementation of Kruskal's algorithm for finding a minimum cost spanning tree in a weighted graph involves successively taking unions of vertex sets in different equivalence classes determined by trees in a forest. Since an equivalence relation on S determines a partition of S into disjoint subsets, the problem of maintaining equivalence relations reduces to the problem of maintaining disjoint sets. The two main operations associated with the disjoint set ADT are efficiently finding the set containing a given element and forming the union of two sets.

One method of maintaining disjoint sets would be to represent each subset $A \subseteq S$ by its characteristic vector $(v_1, v_2, \ldots, v_n)$, where $v_i = 0$ if $v_i \notin A$, and $v_i = 1$ if $v_i \in A$, $i = 1,2, \ldots, n$. Given any collection C of k disjoint subsets $A_1, A_2, \ldots, A_k$ of the base set S, finding the set containing a given element s_i would amount to checking which of the k characteristic vectors contains a 1 in position i. For example, if we maintain a linked list of pointers to the k characteristic vectors, then this check has $\Omega(k)$ worst-case complexity, and requires $\Omega(kn)$ storage locations. Taking the union of two sets would amount to making a linear scan of the two characteristic vectors representing the two sets, resulting in linear complexity (again, assuming the above implementation of maintaining the characteristic vectors). Using trees, we now show how to achieve logarithmic behavior for both union and find operations, and also require only a one-dimensional integer array of length n to represent the disjoint sets.

4.7.1 *Union* and *Find* Algorithms

For disjoint sets, an efficient implementation of the union and find operations is based on the parent array representation of a forest. Consider any collection C of k disjoint subsets $A_1, A_2, \ldots, A_k$ of the base set S, where for convenience we assume that $S = \{0,1, \ldots, n - 1\}$. A rooted forest F *represents* the collection C of disjoint sets if it consists of k rooted trees $T_1, T_2, \ldots, T_k$ such that the vertex set of T_i is A_i, $i = 1,2, \ldots, k$. Clearly, the k sets $A_1, A_2, \ldots, A_k$ are (uniquely) determined by the single array *Parent*$[0{:}n - 1]$ of the parent-array implementation of F. Further, once the array *Parent*$[0{:}n - 1]$ is given, the set A_i can be identified by the single element r_i, where r_i is the root of T_i, $i = 1,2, \ldots, k$ (see Figure 4.24).


Disjoint collection of sets: $\mathscr{C} = \{\{0,2,4,8\},\{3,5,7,9,10\},\{1,6\},\{11\}\}$

i	0	1	2	3	4	5	6	7	8	9	10	11
Parent[i]	–1	6	0	5	0	–1	–1	5	0	5	5	–1

i	0	1	2	3	4	5	6	7	8	9	10	11
Parent[i]	8	6	8	–1	2	7	–1	3	–1	7	3	–1

i	0	1	2	3	4	5	6	7	8	9	10	11
Parent[i]	8	6	8	10	–1	7	–1	10	4	5	–1	–1

Three sample forest representation of the collection C

Figure 4.24

Given an element $x \in A_1 \cup A_2 \cup . . . \cup A_k$, the following procedure *Find1* returns the root r of the tree in forest F containing x.

```
procedure Find1(Parent[0:n – 1],x,r)
Input: Parent[0:n – 1] (array representing disjoint subsets of S)
          x (an element of S)
Output:   r (the root of the tree corresponding to the subset containing x)
    r ← x
    while Parent[r] ≥ 0 do
        r ← Parent[r]
    endwhile
end Find1
```

The complexity of *Find1* is measured by the number of iterations of the **while** loop. Clearly, the complexity of *Find1* depends on the depths of the trees T_i in F. Thus, it is desirable to keep the depths of the trees in the forest relatively small.

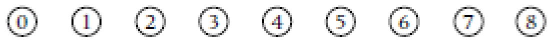
Given $C = \{A_1,A_2, . . . , A_k\}$, let C_{ij} denote the collection of $k – 1$ sets obtained from C by removing sets A_i and A_j and adding the set $A_i \cup A_j$. Consider the two forests F_{ij} and F_{ji}, where F_{ij} is obtained from F by setting *Parent*(r_j) = r_i, and F_{ji} is obtained from F by setting *Parent*(r_i) = r_j. Both forests represent the collection of sets C_{ij}. To help minimize the depth of the tree representing the union of A_i and A_j, we choose the forest F_{ij} if T_i has more vertices than T_j, otherwise we choose the forest F_{ji}. This choice can be made instantly (in constant time) by dynamically keeping track of the number n_i of vertices in each tree T_i. An efficient way of keeping track of n_i without allocating additional memory locations is to (dynamically) store the negative of n_i in *Parent*[r_i]. The root vertices are still distinguishable from the other vertices, since they index those locations of the array *Parent*[0:n – 1] having negative values.

The following pseudocode for the algorithm *Union* computes the parent array implementation of the union of two sets based on the preceding discussion.

```
procedure Union(Parent[0:n – 1],r,s)
Input: Parent[0:n – 1] (an array representing disjoint sets)
          r,s (roots of the trees representing two disjoint sets A,B)
Output:   Parent[0:n – 1] (an array representing disjoint sets after forming
                                                                 A ∪ B)
   sum ← Parent[r] + Parent[s]
   if     Parent[r] > Parent[s] then     //tree rooted at s has more vertices
          Parent[r] ← s                    //than tree rooted at r
          Parent[s] ← sum
   else
          Parent[s] ← r
          Parent[r] ← sum
   endif
end Union
```

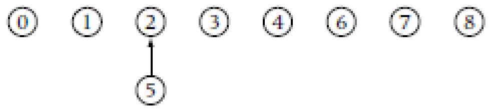
When implementing disjoint sets in a particular algorithm, the initial collection C of disjoint sets is usually the collection of all singleton subsets of the base set S. Figure 4.25 illustrates a sample sequence of calls to the procedure *Union*, starting with the collection of singleton subsets of the base set $S = \{0,1, . . . , 8\}$.

Initial collection: {{0},{1},{2},{3},{4},{5},{6},{7},{8}}



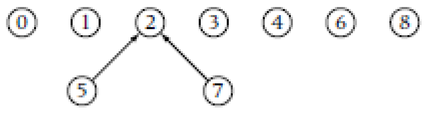
i	0	1	2	3	4	5	6	7	8
Parent[*i*]	−1	−1	−1	−1	−1	−1	−1	−1	−1

Call *Union*[*Parent*[0:8],2,5] ⇒ {{0},{1},{2,5},{3},{4},{6},{7},{8}}



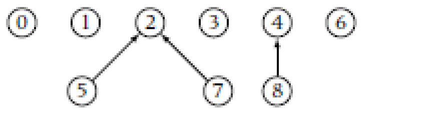
i	0	1	2	3	4	5	6	7	8
Parent[*i*]	−1	−1	−2	−1	−1	2	−1	−1	−1

Call *Union*[*Parent*[0:8],2,7] ⇒ {{0},{1},{2,5,7},{3},{4},{6},{8}}



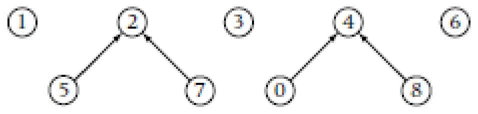
i	0	1	2	3	4	5	6	7	8
Parent[*i*]	−1	−1	−3	−1	−1	2	−1	2	−1

Call *Union*[*Parent*[0:8],4,8] ⇒ {{0},{1},{2,5,7},{3},{4,8},{6}}



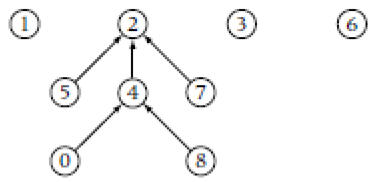
i	0	1	2	3	4	5	6	7	8
Parent[*i*]	−1	−1	−3	−1	−2	2	−1	2	4

Call *Union*[*Parent*[0:8],0,4] ⇒ {{0,4,8},{1},{2,5,7},{3},{6}}



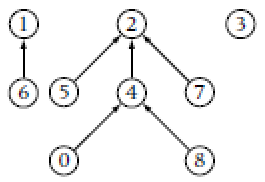
i	0	1	2	3	4	5	6	7	8
Parent[*i*]	4	−1	−3	−1	−3	2	−1	2	4

Call $Union[Parent[0{:}8], 2, 4] \Rightarrow \{\{0,2,4,5,7,8\},\{1\},\{3\},\{6\}\}$



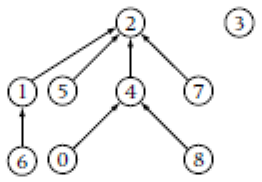
i	0	1	2	3	4	5	6	7	8
$Parent[i]$	4	−1	−6	−1	2	2	−1	2	4

Call $Union[Parent[0{:}8], 1, 6] \Rightarrow \{\{0,2,4,5,7,8\},\{1,6\},\{3\}\}$



i	0	1	2	3	4	5	6	7	8
$Parent[i]$	4	−2	−6	−1	2	2	1	2	4

Call $Union[Parent[0{:}8], 1, 2] \Rightarrow \{\{0,1,2,4,5,6,7,8\},\{3\}\}$



i	0	1	2	3	4	5	6	7	8
$Parent[i]$	4	2	−8	−1	2	2	1	2	4

Representative forests and parent arrays for a sample sequence of calls to procedure *Union*

Figure 4.25

Proposition 4.7.1

Let F be any forest resulting from some sequence of calls to *Union,* where the initial input forest consists of isolated vertices. Then the depth of any tree in F having j vertices is at most $\lfloor \log_2 j \rfloor$.

We leave the proof of Proposition 4.7.1 as an exercise. It follows from Proposition 4.7.1 that if we start with a collection C consisting of singleton sets, and perform a sequence of calls to *Union*, then the computing time of *Find*1 for a given input x is O(log j), where j is the cardinality of the set containing x. Thus, the worst-case complexity in making an intermixed sequence of calls to *Union* and *Find*1, n calls to *Union*, and $m \geq n$ calls to *Find*1, is O(m log n).

4.7.2 Improved *Union* and *Find* Using the Collapsing Rule

Even though the procedures *Union* and *Find1* are very efficient and might seem the best possible, there is still room for improvement. The improvement comes by modifying

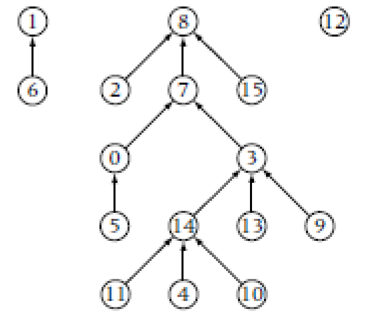
Find1(*Parent*[0:*n* – 1],*x*,*r*) as follows. After *r* is obtained, we can traverse the path from *x* to *r* again and set *Parent*[*v*] = *r* for each vertex *v* encountered. This "collapsing" of the path basically doubles the computing time of *Find1*. However, it tends to reduce the depth of the trees in the forest so that an overall improvement is achieved in the efficiency of making an intermixed sequence of union and find operations. The following procedure *Find2* implements the collapsing rule.

```
procedure Find2(Parent[0:n – 1],x,r)
Input: Parent[0:n – 1] (an array representing disjoint subsets of S)
             x (an element of S)
Output:  r (the root of the tree corresponding to subset containing x)
    r ← x
    while Parent[r] > 0 do
        r ← Parent[r]
    endwhile
    y ← x
    while y ≠ r do
        Temp ← Parent[y]
        Parent[y] ← r
        y ← Temp
    endwhile
end Find2
```

Figure 4.26 illustrates the collapsing rule for a call to *Find2* with *x* = 14, for a sample input forest representing the sets

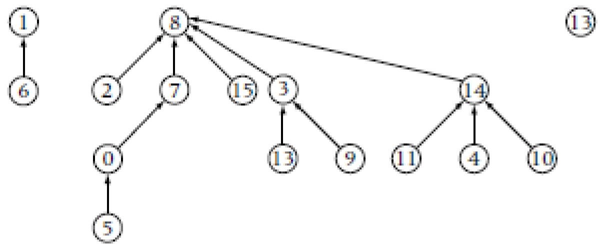
$$\{\{0,2,3,4,5,7,8,9,10,11,13,14,15\},\{1,6\},\{12\}\}.$$

Forest representing collection {{0,2,3,4,5,7,8,9,10,11,13,14,15},{1,6},{12}} before call to *Find2*:



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Parent[i]	7	−2	8	7	14	0	1	8	−13	3	14	14	−1	3	3	0

Forest representing collection {{0,2,3,4,5,7,8,9,10,11,13,14,15},{1,6},{12}} after call to *Find2* with x = 14:



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Parent[i]	7	−2	8	8	14	0	1	8	−13	3	14	14	−1	3	8	0

Illustration of collapsing rule for sample call to *Find2*

Figure 4.26

Tarjan has shown that the worst-case complexity in making an intermixed sequence of calls to *Union* and *Find2*, $n - 1$ calls to *Union* and $m \geq n$ calls to *Find2*, is O($m\ \alpha(m,n)$), where $\alpha(m,n)$ is an extremely slow-growing function. In fact, it grows so slowly that, for all practical purposes, we can regard $\alpha(m,n)$ as a constant function of m and n. The function $\alpha(m,n)$ is related to a functional inverse of the extremely fast-growing Ackermann's function $A(m,n)$ given by the recurrence relation:

$$A(m,n) = A(m-1, A(m,n-1)),\ m \geq 1 \text{ and } n \geq 2,$$
$$A(0,n) = 2n,\ A(m,0) = 0,\ m \geq 1,\ A(m,1) = 2,\ m \geq 1.$$

4.8 Closing Remarks

Further applications of trees will be found throughout the text. For example, in Chapter 5 we discuss the depth-first and breadth-first search trees in graphs and digraphs, and in Chapter 6 we describe various algorithms for minimum spanning trees and shortest-path trees in weighted graphs and digraphs. In Chapter 6 we describe the tree associated with Huffman coding for data compression. In Chapter 8 we use dynamic programming to construct optimal binary search trees. In Chapter 9 we discuss the state space tree associated with backtracking and branch-and-bound algorithms. In Chapter 12 we discuss the application of trees to routing tables in a network.

Exercises

Section 4.2 Mathematical Properties of Binary Trees

4.1 Show that if a binary tree T is full at level i, then it is full at every level j smaller than i.

4.2 Show that the depth of the complete binary tree T_n for a general n is given by $D(T_n) = \lfloor \log_2 n \rfloor$.

4.3 Using induction, prove Proposition 4.2.1.

4.4 Using induction, prove Proposition 4.2.2.

4.5 Using induction, give a direct proof of Proposition 4.2.3 without using the transformation to 2-trees.

4.6 Prove Proposition 4.2.4.

4.7 Prove Proposition 4.2.5.

4.8 Prove Proposition 4.2.6.

4.9 Prove Proposition 4.2.7.

4.10 Complete the verification of Theorem 4.2.8 by establishing the inequality
$x\lfloor \log_2 x \rfloor + 2(x - 2^{\lfloor \log_2 x \rfloor}) \geq \lceil x \log_2 x \rceil$ for any integer $x \geq 1$.

4.11 Given a 2-tree T having L leaf nodes, let T_{left} and T_{right} denote the left and right subtrees (of the root) of T, respectively.
a) Give a recurrence relation for $LPL(T)$ in terms of $LPL(T_{\text{left}})$, $LPL(T_{\text{right}})$, and L.
b) Solve the recurrence relation in (a) to give an alternate proof of Corollary 4.2.9.

4.12 Show that the implicit search tree for *BinarySearch* is a 2-tree that is full at the second-deepest level.

4.13 A *k-ary tree T* is a rooted tree where every node has at most k children. A *k-tree T* is a k-ary tree where every nonleaf node has exactly k children. In each of the following, state and prove a generalization of the result for binary trees or 2-trees to k-ary trees or k-trees.
a) Proposition 4.2.1
b) Proposition 4.2.2
c) Proposition 4.2.3
d) Proposition 4.2.4
e) Proposition 4.2.5
f) Proposition 4.2.6
g) Proposition 4.2.7
h) Theorem 4.2.8

4.14 Prove by induction that
$$\binom{2n}{n} \geq \frac{4^n}{2n+1}, \quad n \geq 1,$$
and conclude that the nth Catalan number $b_n = \frac{1}{n+1}\binom{2n}{n} \in \Omega(4^n/n^2)$.

4.15 Verify that the number b_n of different binary trees on n nodes satisfies the following recurrence relation:

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1} = b_0 b_{n-1} + b_1 b_{n-2} + \cdots + b_{n-1} b_0, \text{init. cond. } b_0 = 1.$$

Section 4.3 Implementation of Trees and Forests

4.16 Write a program that inputs a tree using its parent array representation and converts the tree to its child-sibling representation.

Sections 4.4 Tree Traversals

4.17 Consider the child-sibling implementation of a general tree using pointer and dynamic variables.

a) Give pseudocode for a recursive procedure *TreePreorder* that performs a preorder traversal of the tree whose root node is pointed to by the pointer variable *Root*.

b) In the case where T is a nonempty binary tree, show that the traversal of T generated by *TreePreorder* coincides with the traversal generated by the algorithm *Preorder* for binary trees.

c) Further, show that if T is any tree and T' is the binary tree obtained from T via the Knuth transformation, then the traversal of T generated by *TreePreorder* coincides with the traversal of T' generated by *Preorder*.

4.18 a) Show that a binary tree T can be reconstructed from its inorder and preorder traversal sequences.

b) Show that part (a) is not true in general for its postorder and preorder traversal sequences.

4.19 Give a recursive procedure for swapping the left and right children of every node in a binary tree.

4.20 Associated with each arithmetic expression E involving binary or unary operations is a binary expression tree T defined recursively as follows. The leaf nodes of T correspond to the operands in E and the internal nodes correspond to the operators in E and are labeled accordingly. We may assume without loss of generality that E is fully bracketed. If $E = (E_1 \odot E_2)$ then the root node is labeled $\odot$, and the left and right subtrees of the root are expression trees for E_1 and E_2, respectively.The reverse Polish (the postfix) form of expression E is defined recursively as follows. If F_1 and F_2 are reverse Polish expressions for E_1 and E_2 and $\odot$ is binary operator, then the reverse Polish expression of $E_1 \odot E_2$ is $F_1F_2\odot$. If $\odot$ is a unary operator, then the reverse Polish expression of $\odot E_1$ is $E_1\odot$. For example, the reverse Polish expression for $(A + B)*(–C)$ is $AB + C – *$.

a) Given a string representing a fully bracketed arithmetic expression E involving the binary operators +, –, *, / and the unary operator of minus, write a program that creates an expression tree for E.

b) Show that a postorder traversal of the expression tree for E yields the reverse Polish expression for E.

4.21 Give pseudocode for a version of the inorder-successor function *InorderSucc* when there is a *Parent* field in each node.

4.22 Give pseudocode for a simple modification of the algorithm *Inorder*3 that creates an inorder threading.

4.23 Give pseudocode for an inorder traversal of an inorder-threaded binary tree.

4.24 Give pseudocode for an inorder traversal of a binary tree that does not use an additional field for the parent pointer (or inorder threading), but instead simulates a parent pointer by dynamically reversing and resetting child pointers during the course of the traversal.

4.25 Show that a binary code allows for unambiguous (unique) decoding if, and only if, it is a binary prefix code.

4.26 Show that if *T* is any tree and *T*′ is the binary tree obtained from *T* via the Knuth transformation, then the traversal of *T* generated by *TreeInorder* coincides with the traversal of *T*′ generated by *Inorder*.

4.27 Give pseudocode for nonrecursive versions of
a) preorder traversal.
b) postorder traversal.

Section 4.5 Binary Search Trees

4.28 Give pseudocode for a recursive version of *BinSrchTreeInsert*.

4.29 a) Give pseudocode for a procedure *CreateBinSrchTree*, which creates a binary search tree by repeated calls to *BinSrchTreeInsert*.
b) Show the binary search tree created by *CreateBinSrchTree* for the keys 22,11,0,72,27,55,23,108,1, inserted in that order.
c) Give three orderings for insertion of the keys in part (b) for which the binary search tree created by *CreateBinSrchTree* is a path; that is, each node has at most one child.
d) Give two orderings for insertion of the keys in part (b) for which the binary search tree created by *CreateBinSrchTree* is complete.

4.30 Given any set of *n* distinct keys (identifiers) and an *arbitrary* binary tree *T* on *n* nodes, show that there is a unique assignment of the keys to the nodes in *T* such that *T* becomes a binary search tree for the keys.

4.31 Show that a binary search tree without the (implicit) external leaf nodes added is full at the second-deepest level if, and only if, the associated binary tree with the external leaf nodes added is full at the second-deepest level.

4.32 Give pseudocode for an altered *BinSrchTreeSearch* that determines the locations of both the search element and its parent.

4.33 a) Our pseudocode for inserting a key into a binary search tree assumed that the tree did not contain duplicate keys. Give pseudocode for an alternate version *BinSrchTreeSearch2* that allows duplicate keys to be inserted. When inserting a duplicate key *X*, assume that you always move to the right child of a node already containing *X* (the *move-to-the-right rule*).
b) Show that when creating a binary search tree by successively inserting a sequence of keys using the move-to-the-right rule, an inorder traversal of the resulting search tree will visit duplicate keys in the order in which they were inserted. Moreover, when using *BinSrchTreeSearch2* for a tree built with the move-to-the-right rule, show that the position in the tree of first key inserted amongst a given set of duplicate keys is found. Thus, all keys having a given value can be found by executing *BinSrchTreeSearch* to find the position of the first one that was inserted and then executing an inorder traversal of the right subtree of this position until a key having a different value is encountered.

4.34 Discuss an implementation of an m-way search tree using pointers and dynamic variables.

4.35 For the implementation of the m-way search tree given in the previous exercise, give pseudocode for the high-level searching procedure described in Figure 4.22.

4.36 Show by induction that an inorder traversal of a binary search tree visits the nodes in nondecreasing order of the keys.

4.37 Prove that *TreeSort* is a stable sorting algorithm.

4.38 Given a permutation π of a set of keys, the algorithm *BinSearchTreeCreate* creates a binary search tree T_π. Given T_π, show that π can be uniquely determined if, and only if, T_π is a path.

4.39 Show that the best-case and worst-case complexities of *BinSrchTreeCreate* for a list of size n belong to $\Theta(n \log n)$, $\Theta(n^2)$, respectively.

4.40 a) Show that an inorder traversal of a multiway search tree visits the keys in increasing order.

b) Design a sorting algorithm based on a multiway search tree that generalizes *TreeSort*.

c) Discuss how to guarantee that your sorting algorithm in (a) is stable.

Section 4.6 Priority Queues and Heaps

4.41 Demonstrate the action of *MakeMaxHeap1* on the set of elements

55, 23, 65, 108, 2, 73, 41, 52, 34.

(See Figure 4.22.)

4.42 Demonstrate the action of *MakeMaxHeap2* (see Figure 4.23) on the same set of elements used in Exercise 4.41.

4.43 Another priority queue operation that is sometimes useful is *ChangePriority*(Q,x,v), which changes the priority value of an element x in the queue Q to v (and maintains a priority queue). Give pseudocode for *ChangePriority* when the priority queue is implemented as a min-heap. *ChangePriority* should have worst-case complexity O(log n), where n is the number of elements in the min-heap.

4.44 The concept of a complete binary tree extends to the concept of a complete k-ary tree in the obvious way, $k \geq 2$.

a) Show how a complete k-ary tree can be implemented efficiently using an array.

b) Design and analyze a procedure for inserting an element into a k-ary heap implemented using an array.

c) Design and analyze a procedure for deleting an element from a k-ary heap implemented using an array.

d) Design and analyze a k-ary heapsort.

e) Compare the result in (d) to that for an ordinary heap sort ($k = 2$).

Section 4.7 Implementing Disjoint Sets

4.45 Prove Proposition 4.7.1.

4.46 Suppose we have the following parent implementation of a forest representing a partition of a set of 17 elements:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Parent[*i*]	7	–3	8	7	14	0	1	8	–13	3	14	4	–1	3	3	0	1

a) Sketch the trees in *F*.

b) Show the state of *Parent*[0:16] after a call to *Union*(*Parent*[0:16],1,8) and sketch the trees in *F*.

c) Given the state of *Parent*[0:16] in part (b), show the state of *Parent*[0:16] after an invocation of *Find2*(*Parent*[0:16],4) and sketch the trees in *F*.