12

Trees

Continuing our study of graph theory, we shall now focus on a special type of graph called a tree. First used in 1847 by Gustav Kirchhoff (1824–1887) in his work on electrical networks, trees were later redeveloped and named by Arthur Cayley (1821–1895). In 1857 Cayley used these special graphs in order to enumerate the different isomers of the saturated hydrocarbons C_nH_{2n+2}, $n \in \mathbf{Z}^+$.

With the advent of digital computers, many new applications were found for trees. Special types of trees are prominent in the study of data structures, sorting, and coding theory, and in the solution of certain optimization problems.

12.1 Definitions, Properties, and Examples

Definition 12.1 Let $G = (V, E)$ be a loop-free undirected graph. The graph G is called a *tree*[†] if G is connected and contains no cycles.

In Fig. 12.1 the graph G_1 is a tree, but the graph G_2 is not a tree because it contains the cycle $\{a, b\}, \{b, c\}, \{c, a\}$. The graph G_3 is not connected, so it cannot be a tree. However, each component of G_3 is a tree, and in this case we call G_3 a *forest.*

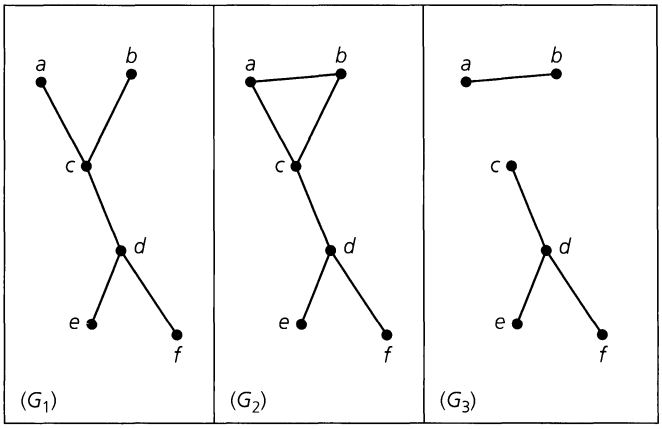


Figure 12.1

[†]As in the case of graphs, the terminology in the study of trees is not standard and the reader may find some differences from one textbook to another.

When a graph is a tree we write T instead of G to emphasize this structure.

In Fig. 12.1 we see that G_1 is a subgraph of G_2 where G_1 contains all the vertices of G_2 and G_1 is a tree. In this situation G_1 is a spanning tree for G_2. Hence a *spanning tree* for a connected graph is a spanning subgraph that is also a tree. We may think of a spanning tree as providing minimal connectivity for the graph and as a minimal skeletal framework holding the vertices together. The graph G_3 provides a *spanning forest* for the graph G_2.

We now examine some properties of trees.

THEOREM 12.1

If a, b are distinct vertices in a tree $T = (V, E)$, then there is a unique path that connects these vertices.

Proof: Since T is connected, there is at least one path in T that connects a and b. If there were more, then from two such paths some of the edges would form a cycle. But T has no cycles.

THEOREM 12.2

If $G = (V, E)$ is an undirected graph, then G is connected if and only if G has a spanning tree.

Proof: If G has a spanning tree T, then for every pair a, b of distinct vertices in V a subset of the edges in T provides a (unique) path between a and b, and so G is connected. Conversely, if G is connected and G is not a tree, remove all loops from G. If the resulting subgraph G_1 is not a tree, then G_1 must contain a cycle C_1. Remove an edge e_1 from C_1 and let $G_2 = G_1 - e_1$. If G_2 contains no cycles, then G_2 is a spanning tree for G because G_2 contains all the vertices in G, is loop-free, and is connected. If G_2 does contain a cycle — say, C_2 — then remove an edge e_2 from C_2 and consider the subgraph $G_3 = G_2 - e_2 = G_1 - \{e_1, e_2\}$. Once again, if G_3 contains no cycles, then we have a spanning tree for G. Otherwise we continue this procedure a finite number of additional times until we arrive at a spanning subgraph of G that is loop-free and connected and contains no cycles (and, consequently, is a spanning tree for G).

Figure 12.2 shows three nonisomorphic trees that exist for five vertices. Although they are not isomorphic, they all have the same number of edges, namely, four. This leads us to the following general result.

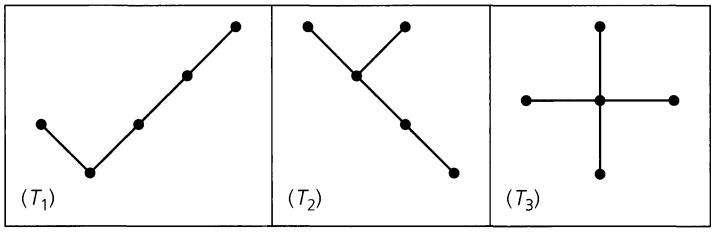


Figure 12.2

THEOREM 12.3

In every tree $T = (V, E)$, $|V| = |E| + 1$.

Proof: The proof is obtained by applying the alternative form of the Principle of Mathematical Induction to $|E|$. If $|E| = 0$, then the tree consists of a single isolated vertex, as in

Fig. 12.3(a). Here $|V| = 1 = |E| + 1$. Parts (b) and (c) of the figure verify the result for the cases where $|E| = 1$ or 2.

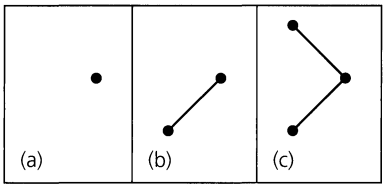


Figure 12.3

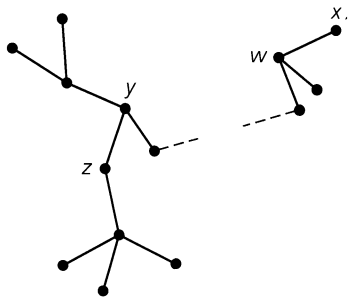


Figure 12.4

Assume the theorem is true for every tree that contains at most k edges, where $k \geq 0$. Now consider a tree $T = (V, E)$, as in Fig. 12.4, where $|E| = k + 1$. [The dotted edge(s) indicates that some of the tree doesn't appear in the figure.] If, for instance, the edge with endpoints y, z is removed from T, we obtain two *subtrees*, $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$, where $|V| = |V_1| + |V_2|$ and $|E_1| + |E_2| + 1 = |E|$. (One of these subtrees could consist of just a single vertex if, for example, the edge with endpoints w, x were removed.) Since $0 \leq |E_1| \leq k$ and $0 \leq |E_2| \leq k$, it follows, by the induction hypothesis, that $|E_i| + 1 = |V_i|$, for $i = 1, 2$. Consequently, $|V| = |V_1| + |V_2| = (|E_1| + 1) + (|E_2| + 1) = (|E_1| + |E_2| + 1) + 1 = |E| + 1$, and the theorem follows by the alternative form of the Principle of Mathematical Induction.

As we examine the trees in Fig. 12.2 we also see that each tree has at least two pendant vertices — that is, vertices of degree 1. This is also true in general.

THEOREM 12.4

For every tree $T = (V, E)$, if $|V| \geq 2$, then T has at least two pendant vertices.

Proof: Let $|V| = n \geq 2$. From Theorem 12.3 we know that $|E| = n - 1$, so by Theorem 11.2 it follows that $2(n - 1) = 2|E| = \sum_{v \in V} \deg(v)$. Since T is connected, we have $\deg(v) \geq 1$ for all $v \in V$. If there are k pendant vertices in T, then each of the other $n - k$ vertices has degree at least 2 and

$$2(n - 1) = 2|E| = \sum_{v \in V} \deg(v) \geq k + 2(n - k).$$

From this we see that $[2(n - 1) \geq k + 2(n - k)] \Rightarrow [(2n - 2) \geq (k + 2n - 2k)] \Rightarrow [-2 \geq -k] \Rightarrow [k \geq 2]$, and the result is consequently established.

EXAMPLE 12.1

In Fig. 12.5 we have two trees, each with 14 vertices (labeled with C's and H's) and 13 edges. Each vertex has degree 4 (C, carbon atom) or degree 1 (H, hydrogen atom). Part (b) of the figure has a carbon atom (C) at the center of the tree. This carbon atom is adjacent to four vertices, three of which have degree 4. There is no vertex (C atom) in part (a) that possesses this property, so the two trees are not isomorphic. They serve as models for the two chemical

isomers that correspond with the saturated[†] hydrocarbon C_4H_{10}. Part (a) represents n-butane (formerly called butane); part (b) represents 2-methyl propane (formerly called isobutane).

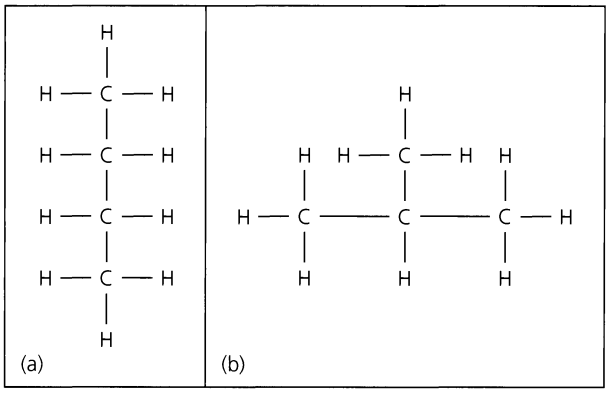


Figure 12.5

A second result from chemistry is given in the following example.

EXAMPLE 12.2

If a saturated hydrocarbon [in particular, an acyclic (no cycles), single-bond hydrocarbon—called an *alkane*] has n carbon atoms, show that it has $2n + 2$ hydrogen atoms.

Considering the saturated hydrocarbon as a tree $T = (V, E)$, let k equal the number of pendant vertices, or hydrogen atoms, in the tree. Then with a total of $n + k$ vertices, where each of the n carbon atoms has degree 4, we find that

$$4n + k = \sum_{v \in V} \deg(v) = 2|E| = 2(|V| - 1) = 2(n + k - 1),$$

and

$$4n + k = 2(n + k - 1) \Rightarrow k = 2n + 2.$$

We close this section with a theorem that provides several different ways to characterize trees.

THEOREM 12.5

The following statements are equivalent for a loop-free undirected graph $G = (V, E)$.

a) G is a tree.

b) G is connected, but the removal of any edge from G disconnects G into two subgraphs that are trees.

c) G contains no cycles, and $|V| = |E| + 1$.

d) G is connected, and $|V| = |E| + 1$.

[†]The adjective *saturated* is used here to indicate that for the number of carbon atoms present in the molecule, we have the maximum number of hydrogen atoms.

e) G contains no cycles, and if $a, b \in V$ with $\{a, b\} \notin E$, then the graph obtained by adding edge $\{a, b\}$ to G has precisely one cycle.

Proof: We shall prove that (a) $\Rightarrow$ (b), (b) $\Rightarrow$ (c), and (c) $\Rightarrow$ (d), leaving to the reader the proofs for (d) $\Rightarrow$ (e) and (e) $\Rightarrow$ (a).

[(a) $\Rightarrow$ (b)]: If G is a tree, then G is connected. So let $e = \{a, b\}$ be any edge of G. Then if $G - e$ is connected, there are at least two paths in G from a to b. But this contradicts Theorem 12.1. Hence $G - e$ is disconnected and so the vertices in $G - e$ may be partitioned into two subsets: (1) vertex a and those vertices that can be reached from a by a path in $G - e$; and (2) vertex b and those vertices that can be reached from b by a path in $G - e$. These two connected components are trees because a loop or cycle in either component would also be in G.

[(b) $\Rightarrow$ (c)]: If G contains a cycle, then let $e = \{a, b\}$ be an edge of the cycle. But then $G - e$ is connected, contradicting the hypothesis in part (b). So G contains no cycles, and since G is a loop-free connected undirected graph, we know that G is a tree. Consequently, it follows from Theorem 12.3 that $|V| = |E| + 1$.

[(c) $\Rightarrow$ (d)]: Let $\kappa(G) = r$ and let $G_1, G_2, \ldots, G_r$ be the components of G. For $1 \leq i \leq r$, select a vertex $v_i \in G_i$ and add the $r - 1$ edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{r-1}, v_r\}$ to G to form the graph $G' = (V, E')$, which is a tree. Since G' is a tree, we know that $|V| = |E'| + 1$ because of Theorem 12.3. But from part (c), $|V| = |E| + 1$, so $|E| = |E'|$ and $r - 1 = 0$. With $r = 1$, it follows that G is connected.

EXERCISES 12.1

1. a) Draw the graphs of all nonisomorphic trees on six vertices.

b) How many isomers does hexane (C_6H_{14}) have?

2. Let $T_1 = (V_1, E_1)$, $T_2 = (V_2, E_2)$ be two trees where $|E_1| = 17$ and $|V_2| = 2|V_1|$. Determine $|V_1|$, $|V_2|$, and $|E_2|$.

3. a) Let $F_1 = (V_1, E_1)$ be a forest of seven trees where $|E_1| = 40$. What is $|V_1|$?

b) If $F_2 = (V_2, E_2)$ is a forest with $|V_2| = 62$ and $|E_2| = 51$, how many trees determine F_2?

4. If $G = (V, E)$ is a forest with $|V| = v$, $|E| = e$, and κ components (trees), what relationship exists among v, e, and κ?

5. What kind of trees have exactly two pendant vertices?

6. a) Verify that all trees are planar.

b) Derive Theorem 12.3 from part (a) and Euler's Theorem for planar graphs.

7. Give an example of an undirected graph $G = (V, E)$ where $|V| = |E| + 1$ but G is not a tree.

8. a) If a tree has four vertices of degree 2, one vertex of degree 3, two of degree 4, and one of degree 5, how many pendant vertices does it have?

b) If a tree $T = (V, E)$ has v_2 vertices of degree 2, v_3 vertices of degree 3, ..., and v_m vertices of degree m, what are $|V|$ and $|E|$?

9. If $G = (V, E)$ is a loop-free undirected graph, prove that G is a tree if there is a unique path between any two vertices of G.

10. The connected undirected graph $G = (V, E)$ has 30 edges. What is the maximum value that $|V|$ can have?

11. Let $T = (V, E)$ be a tree with $|V| = n \geq 2$. How many distinct paths are there (as subgraphs) in T?

12. Let $G = (V, E)$ be a loop-free connected undirected graph where $V = \{v_1, v_2, v_3, \ldots, v_n\}$, $n \geq 2$, $\deg(v_1) = 1$, and $\deg(v_i) \geq 2$ for $2 \leq i \leq n$. Prove that G must have a cycle.

13. Find two nonisomorphic spanning trees for the complete bipartite graph $K_{2,3}$. How many nonisomorphic spanning trees are there for $K_{2,3}$?

14. For $n \in \mathbf{Z}^+$, how many nonisomorphic spanning trees are there for $K_{2,n}$?

15. Determine the number of nonidentical (though some may be isomorphic) spanning trees that exist for each of the graphs shown in Fig. 12.6.

16. For each graph in Fig. 12.7, determine how many nonidentical (though some may be isomorphic) spanning trees exist.

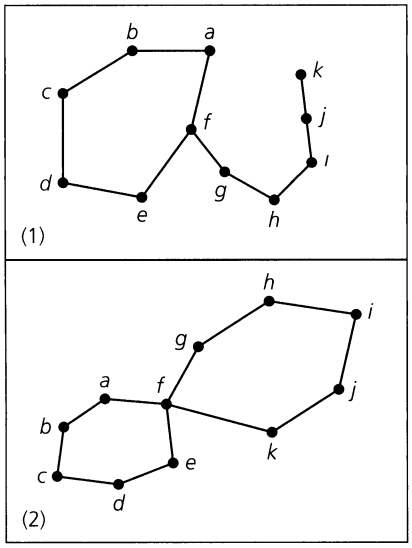


Figure 12.6

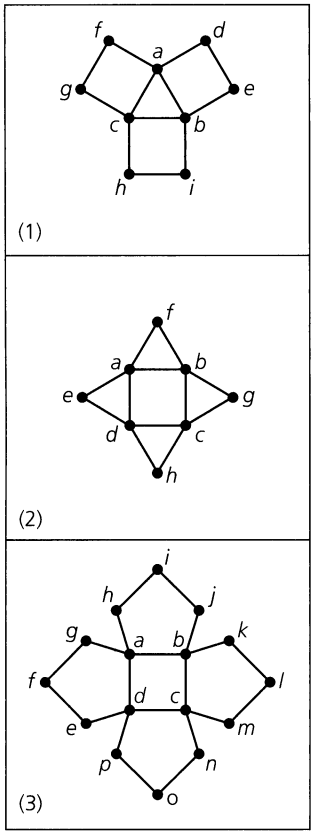


Figure 12.7

17. Let $T = (V, E)$ be a tree where $|V| = n$. Suppose that for each $v \in V$, $\deg(v) = 1$ or $\deg(v) \geq m$, where m is a fixed positive integer and $m \geq 2$.

a) What is the smallest value possible for n?

b) Prove that T has at least m pendant vertices.

18. Suppose that $T = (V, E)$ is a tree with $|V| = 1000$. What is the sum of the degrees of all the vertices in T?

19. Let $G = (V, E)$ be a loop-free connected undirected graph. Let H be a subgraph of G. The *complement of H in G* is the subgraph of G made up of those edges in G that are not in H (along with the vertices incident to these edges).

a) If T is a spanning tree of G, prove that the complement of T in G does not contain a cut-set of G.

b) If C is a cut-set of G, prove that the complement of C in G does not contain a spanning tree of G.

20. Complete the proof of Theorem 12.5.

21. A labeled tree is one wherein the vertices are labeled. If the tree has n vertices, then $\{1, 2, 3, \ldots, n\}$ is used as the set of labels. We find that two trees that are isomorphic without labels may become nonisomorphic when labeled. In Fig. 12.8, the first two trees are isomorphic as labeled trees. The third tree is isomorphic to the other two if we ignore the labels; as a labeled tree, however, it is not isomorphic to either of the other two.

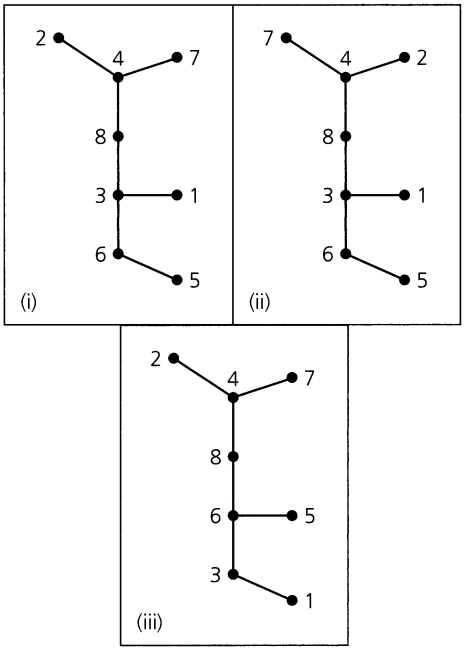


Figure 12.8

The number of nonisomorphic trees with n labeled vertices can be counted by setting up a one-to-one correspondence between these trees and the n^{n-2} sequences (with repetitions allowed) $x_1, x_2, \ldots, x_{n-2}$ whose entries are taken from $\{1, 2, 3, \ldots, n\}$. If T is one such labeled tree, we use the following algorithm to find its corresponding sequence — called the *Prüfer code* for the tree. (Here T has at least one edge.)

Step 1: Set the counter i to 1.

Step 2: Set $T(i) = T$.

Step 3: Since a tree has at least two pendant vertices, select the pendant vertex in $T(i)$ with the smallest label y_i. Now remove the edge $\{x_i, y_i\}$ from $T(i)$ and use x_i for the ith component of the sequence.

Step 4: If $i = n - 2$, we have the sequence corresponding to the given labeled tree $T(1)$. If $i \neq n - 2$, increase i by 1, set $T(i)$ equal to the resulting subtree obtained in step (3), and return to step (3).

a) Find the six-digit sequence (Prüfer code) for trees (i) and (iii) in Fig. 12.8.

b) If v is a vertex in T, show that the number of times the label on v appears in the Prüfer code $x_1, x_2, \ldots, x_{n-2}$ is $\deg(v) - 1$.

c) Reconstruct the labeled tree on eight vertices that is associated with the Prüfer code 2, 6, 5, 5, 5, 5.

d) Develop an algorithm for reconstructing a tree from a given Prüfer code $x_1, x_2, \ldots, x_{n-2}$.

22. Let $n \in \mathbf{Z}^+$, $n \geq 3$. If v is a vertex in K_n, how many of the n^{n-2} spanning trees of K_n have v as a pendant vertex?

23. Characterize the trees whose Prüfer codes

a) contain only one integer, or

b) have distinct integers in all positions.

24. Show that the number of labeled trees with n vertices, k of which are pendant vertices, is $\binom{n}{k}(n-k)!S(n-2, n-k) = (n!/k!)S(n-2, n-k)$, where $S(n-2, n-k)$ is a Stirling number of the second kind. (This result was first established in 1959 by A. Rényi.)

25. Let $G = (V, E)$ be the undirected graph in Fig. 12.9. Show that the edge set E can be partitioned as $E_1 \cup E_2$ so that the subgraphs $G_1 = (V, E_1)$, $G_2 = (V, E_2)$ are isomorphic spanning trees of G.

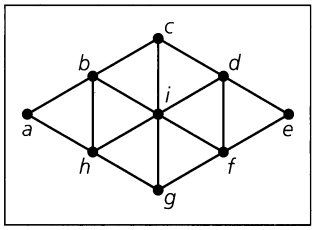


Figure 12.9

12.2 Rooted Trees

We turn now to directed trees. We find a variety of applications for a special type of directed tree called a rooted tree.

Definition 12.2

If G is a directed graph, then G is called a *directed tree* if the undirected graph associated with G is a tree. When G is a directed tree, G is called a *rooted tree* if there is a unique vertex r, called the *root*, in G with the in degree of $r = id(r) = 0$, and for all other vertices v, the in degree of $v = id(v) = 1$.

The tree in part (a) of Fig. 12.10 is directed but not rooted; the tree in part (b) is rooted with root r.

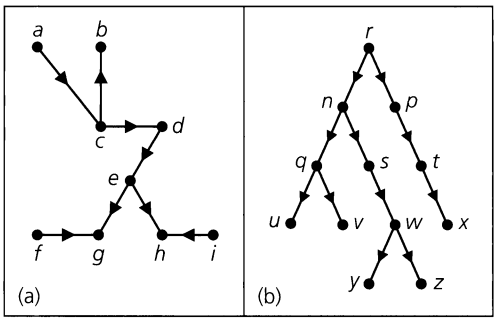


Figure 12.10

We draw rooted trees as in Fig. 12.10(b) but with the directions understood as going from the upper level to the lower level, so that the arrows aren't needed. In a rooted tree, a vertex with out degree 0 is called a *leaf* (or *terminal vertex.*) Vertices u, v, x, y, z are leaves in Fig. 12.10(b). All other vertices are called *branch nodes* (or *internal vertices*).

Consider the vertex s in this rooted tree [Fig. 12.10(b)]. The path from the root, r, to s is of length 2, so we say that s is at *level* 2 in the tree, or that s has *level number* 2. Similarly, x is at level 3, whereas y has level number 4. We call s a *child* of n, and we call n the *parent* of s. Vertices w, y, and z are considered *descendants* of s, n, and r, while s, n, and r are called *ancestors* of w, y, and z. In general, if v_1 and v_2 are vertices in a rooted tree and v_1 has the smaller level number, then v_1 is an ancestor of v_2 (or v_2 is a descendant of v_1) if there is a (directed) path from v_1 to v_2. Two vertices with a common parent are referred to as *siblings*. Such is the case for vertices q and s, whose common parent is vertex n. Finally, if v_1 is any vertex of the tree, the *subtree at* v_1 is the subgraph induced by the root v_1 and all of its descendants (there may be ~~none~~).

EXAMPLE 12.3

In Fig. 12.11(a) a rooted tree is used to represent the table of contents of a three-chapter ($C1$, $C2$, $C3$) book. Vertices with level number 2 are for sections within a chapter; those at level 3 represent subsections within a section. Part (b) of the figure displays the natural order for the table of contents of this book.

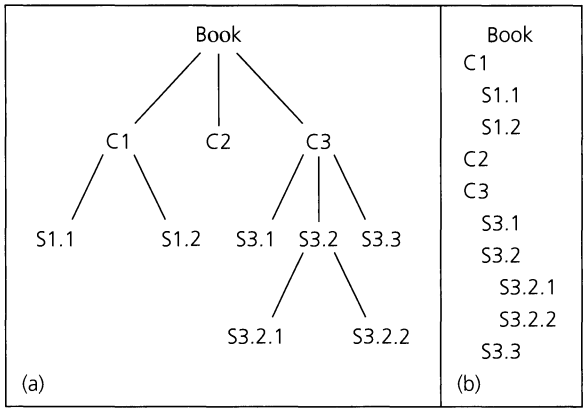


Figure 12.11

The tree in Fig. 12.11(a) suggests an order for the vertices if we examine the subtrees at $C1$, $C2$, and $C3$ from left to right. (This order will recur again in this section, in a more general context.) We now consider a second example that provides such an order.

EXAMPLE 12.4

In the tree T shown in Fig. 12.12, the edges (or branches, as they are often called) leaving each internal vertex are *ordered* from left to right. Hence T is called an *ordered rooted tree.*

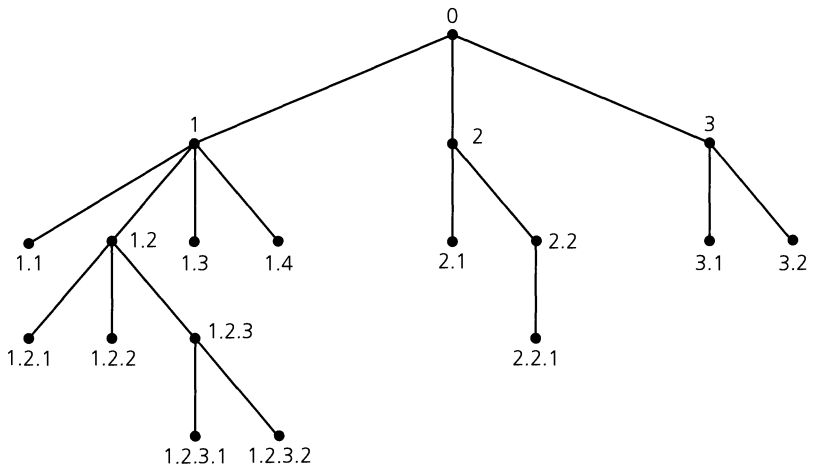


Figure 12.12

We label the vertices for this tree by the following algorithm.

Step 1: First assign the root the label (or *address*) 0.

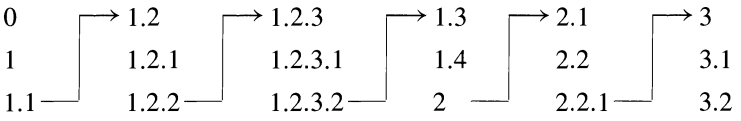
Step 2: Next assign the positive integers 1, 2, 3, . . . to the vertices at level 1, going from left to right.

Step 3: Now let v be an internal vertex at level $n \geq 1$, and let $v_1, v_2, \ldots, v_k$ denote the children of v (going from left to right). If a is the label assigned to vertex v, assign the labels $a.1, a.2, \ldots, a.k$ to the children $v_1, v_2, \ldots, v_k$, respectively.

Consequently, each vertex in T, other than the root, has a label of the form $a_1.a_2.a_3 \ldots . a_n$ if and only if that vertex has level number n. This is known as the *universal address system*.

This system provides a way to *order* all vertices in T. If u and v are two vertices in T with addresses b and c, respectively, we define $b < c$ if (a) $b = a_1.a_2 \ldots . a_m$ and $c = a_1.a_2 \ldots . a_m.a_{m+1} \ldots . a_n$, with $m < n$; or (b) $b = a_1.a_2 \ldots . a_m.x_1 \ldots . y$ and $c = a_1.a_2 \ldots . a_m.x_2 \ldots . z$, where $x_1, x_2 \in \mathbf{Z}^+$ and $x_1 < x_2$.

For the tree under consideration, this ordering yields



Since this resembles the alphabetical ordering in a dictionary, the order is called the *lexicographic*, or *dictionary, order*.

We now consider an application of a rooted tree in the study of computer science.

EXAMPLE 12.5

a) A rooted tree is a *binary* rooted tree if for each vertex v, $od(v) = 0$, 1, or 2 — that is, if v has at most two children. If $od(v) = 0$ or 2 for all $v \in V$, then the rooted tree is called a *complete* binary tree. Such a tree can represent a binary operation, as in parts

(a) and (b) of Fig. 12.13. To avoid confusion when dealing with a noncommutative operation $\circ$, we label the root as $\circ$ and require the result to be $a \circ b$, where a is the left child, and b the right child, of the root.

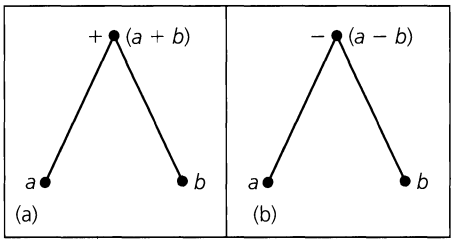


Figure 12.13

b) In Fig. 12.14 we extend the ideas presented in Fig. 12.13 in order to construct the binary rooted tree for the algebraic expression

$$((7 - a)/5) * ((a + b) \uparrow 3),$$

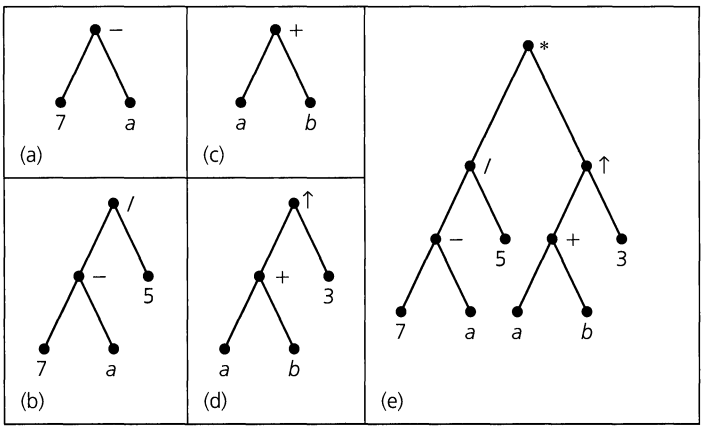


Figure 12.14

where $*$ denotes multiplication and $\uparrow$ denotes exponentiation. Here we construct this tree, as shown in part (e) of the figure, from the bottom up. First, a subtree for the expression $7 - a$ is constructed in part (a) of Fig. 12.14. This is then incorporated (as the left subtree for /) in the binary rooted tree for the expression $(7 - a)/5$ in Fig. 12.14 (b). Then, in a similar way, the binary rooted trees in parts (c) and (d) of the figure are constructed for the expressions $a + b$ and $(a + b) \uparrow 3$, respectively. Finally, the two subtrees in parts (b) and (d) are used as the left and right subtrees, respectively, for $*$ and give us the binary rooted tree [in Fig. 12.14(e)] for $((7 - a)/5) * ((a + b) \uparrow 3)$.

The same ideas are used in Fig. 12.15, where we find the binary rooted trees for the algebraic expressions

$$(a - (3/b)) + 5 \text{ [in part (a)]} \qquad \text{and} \qquad a - (3/(b + 5)) \text{ [in part (b)]}.$$

c) In evaluating $t + (uv)/(w + x - y^z)$ in certain procedural languages, we write the expression in the form $t + (u * v)/(w + x - y \uparrow z)$. When the computer evaluates this expression, it performs the binary operations (within each parenthesized part) according to a hierarchy of operations whereby exponentiation precedes multiplication

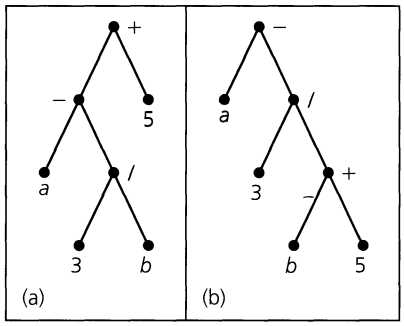


Figure 12.15

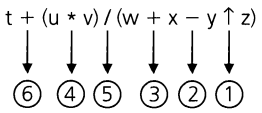


Figure 12.16

and division, which in turn precede addition and subtraction. In Fig. 12.16 we number the operations in the order in which they are performed by the computer. For the computer to evaluate this expression, it must somehow scan the expression in order to perform the operations in the order specified.

Instead of scanning back and forth continuously, however, the machine converts the expression into a notation that is independent of parentheses. This is known as Polish notation, in honor of the Polish (actually Ukrainian) logician Jan Lukasiewicz (1878–1956). Here the *infix* notation $a \circ b$ for a binary operation $\circ$ becomes $\circ\, ab$, the *prefix* (or Polish) notation. The advantage is that the expression in Fig. 12.16 can be rewritten without parentheses as

$$+\, t / * u v + w - x \uparrow y z,$$

where the evaluation proceeds from right to left. When a binary operation is encountered, it is performed on the two operands to its right. The result is then treated as one of the operands for the next binary operation encountered as we continue to the left. For instance, given the assignments $t = 4$, $u = 2$, $v = 3$, $w = 1$, $x = 9$, $y = 2$, $z = 3$, the following steps take place in the evaluation of the expression

$$+\, t / * u v + w - x \uparrow y z.$$

1) $+\ 4\ /\ *\ 2\ 3\ +\ 1\ -\ 9\ \underbrace{\uparrow\ 2\ 3}_{2 \uparrow 3 = 8}$

2) $+\ 4\ /\ *\ 2\ 3\ +\ 1\ \underbrace{-\ 9\ 8}_{9 - 8 = 1}$

3) $+\ 4\ /\ *\ 2\ 3\ \underbrace{+\ 1\ 1}_{1 + 1 = 2}$

4) $+\ 4\ /\ \underbrace{*\ 2\ 3}_{2 * 3 = 6}\ 2$

5) $+\ 4\ \underbrace{/\ 6\ 2}_{6\ /\ 2 = 3}$

6) $\underbrace{+\ 4\ 3}_{4 + 3 = 7}$

So the value of the given expression for the preceding assignments is 7.

The use of Polish notation is important for the compilation of computer programs and can be obtained by representing a given expression by a rooted tree, as shown in Fig. 12.17. Here each variable (or constant) is used to label a leaf of the tree. Each internal vertex is

labeled by a binary operation whose left and right operands are the left and right subtrees it determines. Starting at the root, as we transverse the tree from top to bottom and left to right, as shown in Fig. 12.17, we find the Polish notation by writing down the labels of the vertices in the order in which they are visited.

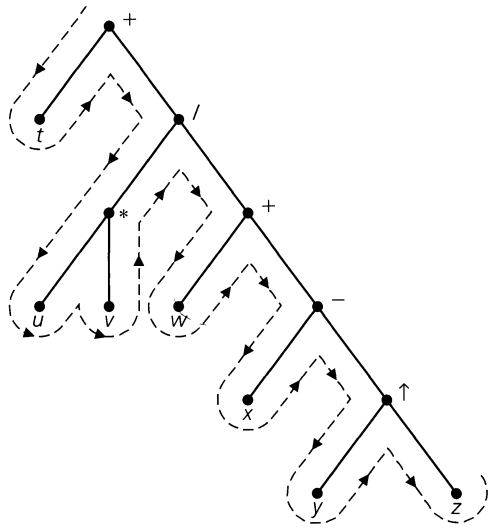


Figure 12.17

The last two examples illustrate the importance of order. Several methods exist for systematically ordering the vertices in a tree. Two of the most prevalent in the study of data structures are the preorder and postorder. These are defined recursively in the following definition.

Definition 12.3

Let $T = (V, E)$ be a rooted tree with root r. If T has no other vertices, then the root by itself constitutes the *preorder* and *postorder traversals* of T. If $|V| > 1$, let $T_1, T_2, T_3, \ldots, T_k$ denote the subtrees of T as we go from left to right (as in Fig. 12.18).

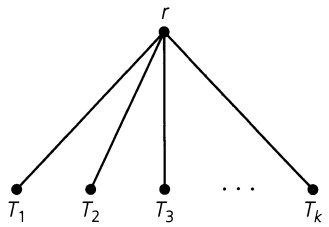


Figure 12.18

a) The *preorder traversal* of T first visits r and then traverses the vertices of T_1 in preorder, then the vertices of T_2 in preorder, and so on until the vertices of T_k are traversed in preorder.

b) The *postorder traversal* of T traverses in postorder the vertices of the subtrees T_1, $T_2, \ldots, T_k$ and then visits the root.

We demonstrate these ideas in the following example.

EXAMPLE 12.6

Consider the rooted tree shown in Fig. 12.19.

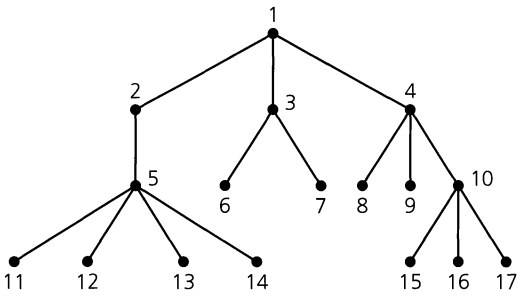


Figure 12.19

a) *Preorder*: After visiting vertex 1 we visit the subtree T_1 rooted at vertex 2. After visiting vertex 2 we proceed to the subtree rooted at vertex 5, and after visiting vertex 5 we go to the subtree rooted at vertex 11. This subtree has no other vertices, so we visit vertex 11 and then return to vertex 5 from which we visit, in succession, vertices 12, 13, and 14. Following this we *backtrack* (14 to 5 to 2 to 1) to the root and then visit the vertices in the subtree T_2 in the preorder 3, 6, 7. Finally, after returning to the root for the last time, we traverse the subtree T_3 in the preorder 4, 8, 9, 10, 15, 16, 17. Hence the preorder listing of the vertices in this tree is 1, 2, 5, 11, 12, 13, 14, 3, 6, 7, 4, 8, 9, 10, 15, 16, 17.

In this ordering we start at the root and build a path as far as we can. At each level we go to the leftmost vertex (not previously visited) at the next level, until we reach a leaf ℓ. Then we backtrack to the parent p of this leaf ℓ and visit ℓ's sibling s (and the subtree that s determines) directly on its right. If no such sibling s exists, we backtrack to the grandparent g of the leaf ℓ and visit, if it exists, a vertex u that is the sibling of p directly to its right in the tree. Continuing in this manner, we eventually visit (the first time each one is encountered) all of the vertices in the tree.

The vertices in Figs. 12.11(a), 12.12, and 12.17 are visited in preorder. The preorder traversal for the tree in Fig. 12.11(a) provides the ordering in Fig. 12.11(b). The lexicographic order in Example 12.4 arises from the preorder traversal of the tree in Fig. 12.12.

b) *Postorder*: For the postorder traversal of a tree, we start at the root r and build the longest path, going to the leftmost child of each internal vertex whenever we can. When we arrive at a leaf ℓ we visit this vertex and then backtrack to its parent p. However, we do not visit p until after all of its descendants are visited. The next vertex we visit is found by applying the same procedure at p that was originally applied at r in obtaining ℓ — except that now we first go from p to the sibling of ℓ directly to the right (of ℓ). And at no time is any vertex visited more than once or before any of its descendants.

For the tree given in Fig. 12.19, the postorder traversal starts with a postorder traversal of the subtree T_1 rooted at vertex 2. This yields the listing 11, 12, 13, 14, 5, 2. We proceed to the subtree T_2, and the postorder listing continues with 6, 7, 3. Then for T_3 we find 8, 9, 15, 16, 17, 10, 4 as the postorder listing. Finally, vertex 1 is visited. Consequently, for this tree, the postorder traversal visits the vertices in the order 11, 12, 13, 14, 5, 2, 6, 7, 3, 8, 9, 15, 16, 17, 10, 4, 1.

In the case of binary rooted trees, a third type of tree traversal called the inorder traversal may be used. Here we do *not* consider subtrees as first and second, but rather in terms of left and right. The formal definition is recursive, as were the definitions of preorder and postorder traversals.

Definition 12.4

Let $T = (V, E)$ be a binary rooted tree with vertex r the root.

1) If $|V| = 1$, then the vertex r constitutes the *inorder traversal* of T.
2) When $|V| > 1$, let T_L and T_R denote the left and right subtrees of T. The *inorder traversal* of T first traverses the vertices T_L in inorder, then it visits the root r, and then it traverses, in inorder, the vertices of T_R.

We realize that here a left or right subtree may be empty. Also, if v is a vertex in such a tree and $od(v) = 1$, then if w is the child of v, we must distinguish between w's being the left child and its being the right child.

EXAMPLE 12.7

As a result of the previous comments, the two binary rooted trees shown in Fig. 12.20 are not considered the same, when viewed as *ordered trees*. As rooted binary trees they are the same. (Each tree has the same set of vertices and the same set of directed edges.) However, when we consider the additional concept of left and right children, we see that in part (a) of the figure vertex v has right child a, whereas in part (b) vertex a is the left child of v. Consequently, when the difference between left and right children is taken into consideration, these trees are no longer viewed as the same tree.

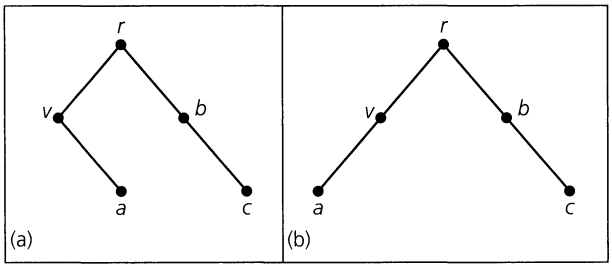


Figure 12.20

In visiting the vertices for the tree in part (a) of Fig. 12.20, we first visit in inorder the left subtree of the root r. This subtree consists of the root v and its *right child* a. (Here the left child is *null*, or nonexistent.) Since v has no left subtree, we visit in inorder vertex v and then its right subtree, namely, a. Having traversed the left subtree of r, we now visit vertex r and then traverse, in inorder, the vertices in the right subtree of r. This results in our visiting first vertex b (because b has no left subtree) and then vertex c. Hence the inorder listing for the tree shown in Fig. 12.20(a) is v, a, r, b, c.

When we consider the tree in part (b) of the figure, once again we start by visiting, in inorder, the vertices in the left subtree of the root r. Here, however, this left subtree consists of vertex v (the root of the subtree) and its *left* child a. (In this case, the right child of v is null, or nonexistent.) Therefore this inorder traversal first visits vertex a (the left subtree of v), and then vertex v. Since v has no right subtree, we are now finished visiting the left subtree of r, in inorder. So next the root r is visited, and then the vertices of the right subtree

of r are traversed, in inorder. This results in the inorder listing a, v, r, b, c for the tree shown in Fig. 12.20(b).

We should note, however, that for the preorder traversal *in this particular example*,[†] the same result is obtained for both trees:

$$\text{Preorder listing:} \quad r, v, a, b, c.$$

Likewise, this particular example is such that the postorder traversal for either tree gives us the following:

$$\text{Postorder listing:} \quad a, v, c, b, r.$$

It is only for the inorder traversal, with its distinctions between left and right children and between left and right subtrees, that a difference occurs. For the trees in parts (a) and (b) of Fig. 12.20 we found the respective inorder listings to be

$$\text{(a) } v, a, r, b, c \qquad \text{and} \qquad \text{(b) } a, v, r, b, c.$$

EXAMPLE 12.8

If we apply the inorder traversal to the binary rooted tree shown in Fig. 12.21, we find that the inorder listing for the vertices is $p, j, q, f, c, k, g, a, d, r, b, h, s, m, e, i, t, n, u$.

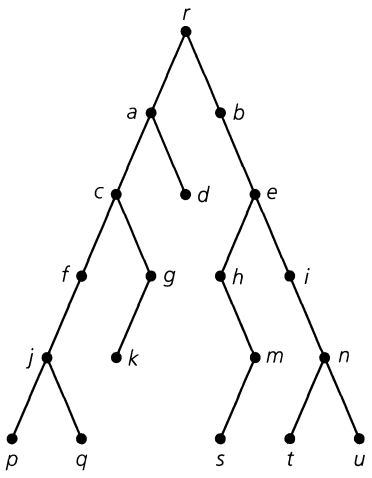


Figure 12.21

Our next example shows how the preorder traversal can be used in a counting problem dealing with binary trees.

EXAMPLE 12.9[‡]

For $n \geq 0$, consider the complete binary trees on $2n + 1$ vertices. The cases for $0 \leq n \leq 3$ are shown in Fig. 12.22. Here we distinguish left from right. So, for example, the two

---

[†]A note of caution! If we interchange the order of the two existing children (of a certain parent) in a binary rooted tree, then a change results in the preorder, postorder, and inorder traversals. If one child is "null," however, then only the inorder traversal changes.

[‡]This example uses material developed in the optional Sections 1.5 and 10.5. It may be omitted with no loss of continuity.

complete binary trees for $n = 2$ are considered distinct. [If we do not distinguish left from right, these trees are (isomorphic and) no longer counted as two different trees.]

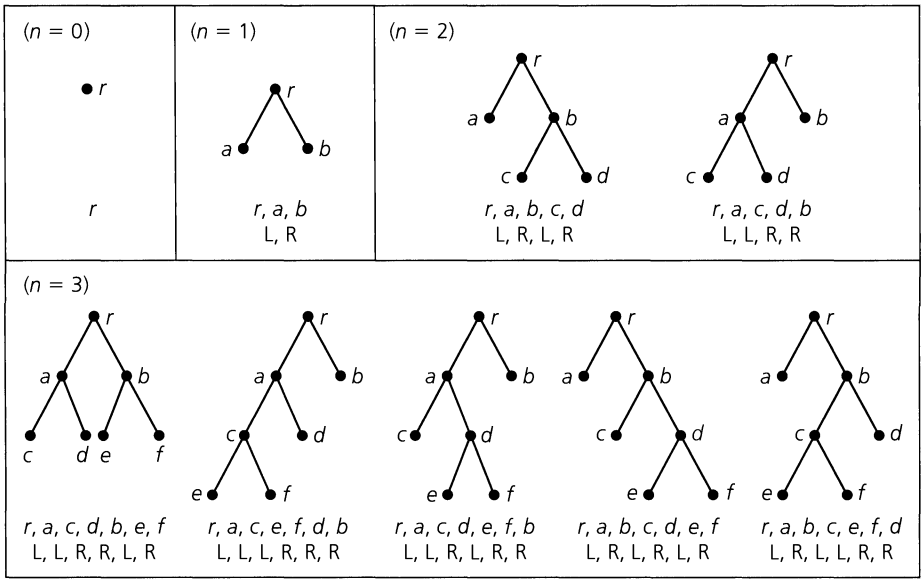


Figure 12.22

Below each tree in the figure we list the vertices for a preorder traversal. In addition, for $1 \leq n \leq 3$, we find a list of n L's and n R's under each preorder traversal. These lists are determined as follows. The first tree for $n = 2$, for instance, has the list L, R, L, R because, after visiting the root r, we go to the left (L) subtree rooted at a and visit vertex a. Then we backtrack to r and go to the right (R) subtree rooted at b. After visiting vertex b we go to the left (L) subtree of b rooted at c and visit vertex c. Then, lastly, we backtrack to b and go to its right (R) subtree to visit vertex d. This generates the list L, R, L, R and the other seven lists of L's and R's are obtained in the same way.

Since we are traversing these trees in preorder, each list starts with an L. There is an equal number of L's and R's in each list because the trees are complete binary trees. Finally, the number of R's never exceeds the number of L's as a given list is read from left to right — again, because we have a preorder traversal. Should we replace each L by a 1 and each R by a -1, for the five trees for $n = 3$, we find ourselves back in part (a) of Example 1.43, where we have one of our early examples of the Catalan numbers. Hence, for $n \geq 0$, we see that the number of complete binary trees on $2n + 1$ vertices is $\frac{1}{n+1}\binom{2n}{n}$, the nth Catalan number. [Note that if we *prune* the five trees for $n = 3$ by removing the four leaves for each tree, we obtain the five rooted ordered binary trees in Fig. 10.18.]

The notion of preorder now arises in the following procedure for finding a spanning tree for a connected graph.

Let $G = (V, E)$ be a loop-free connected undirected graph with $r \in V$. Starting from r, we construct a path in G that is as long as possible. If this path includes every vertex in V, then the path is a spanning tree T for G and we are finished. If not, let x and y be the last two vertices visited along this path, with y the last vertex. We then return, or *backtrack*, to the vertex x and construct a second path in G that is as long as possible, starts at x, and

doesn't include any vertex already visited. If no such path exists, backtrack to the parent p of x and see how far it is possible to branch off from p, building a path (that is as long as possible and has no previously visited vertices) to a new vertex y_1 (which will be a new leaf for T). Should all edges from the vertex p lead to vertices already encountered, backtrack one level higher and continue the process. Since the graph is finite and connected, this technique, which is called *backtracking*, or *depth-first search*, eventually determines a spanning tree T for G, where r is regarded as the root of T. Using T, we then order the vertices of G in a preorder listing.

The depth-first search serves as a framework around which many algorithms can be designed to test for certain graph properties. One such algorithm will be examined in detail in Section 12.5.

One way to help implement the depth-first search in a computer program is to assign a fixed order to the vertices of the given graph $G = (V, E)$. Then if there are two or more vertices adjacent to a vertex v and none of these vertices has already been visited, we shall know exactly which vertex to visit first. This order now helps us to develop the foregoing description of the depth-first search as an algorithm.

Let $G = (V, E)$ be a loop-free connected undirected graph where $|V| = n$ and the vertices are ordered as $v_1, v_2, v_3, \ldots, v_n$. To find the rooted ordered depth-first spanning tree for the prescribed order, we apply the following algorithm, wherein the variable v is used to store the vertex presently being examined.

Depth-First Search Algorithm

Step 1: Assign v_1 to the variable v and initialize T as the tree consisting of just this one vertex. (The vertex v_1 will be the root of the spanning tree that develops.) Visit v_1.

Step 2: Select the smallest subscript i, for $2 \leq i \leq n$, such that $\{v, v_i\} \in E$ and v_i has not already been visited.

If no such subscript is found, then go to step (3). Otherwise, perform the following: (1) Attach the edge $\{v, v_i\}$ to the tree T and visit v_i; (2) Assign v_i to v; and (3) Return to step (2).

Step 3: If $v = v_1$, the tree T is the (rooted ordered) spanning tree for the order specified.

Step 4: For $v \neq v_1$, backtrack from v to its parent u in T. Then assign u to v and return to step (2).

EXAMPLE 12.10

We now apply this algorithm to the graph $G = (V, E)$ shown in Fig. 12.23(a). Here the order for the vertices is alphabetic: $a, b, c, d, e, f, g, h, i, j$.

First we assign the vertex a to the variable v and initialize T as just the vertex a (the root). We visit vertex a. Then, going to step (2), we find that the vertex b is the first vertex w such that $\{a, w\} \in E$ and w has not been visited earlier. So we attach edge $\{a, b\}$ to T and visit b, assign b to v, and then return to step (2).

At $v = b$ we find that the first vertex (not visited earlier) that provides an edge for the spanning tree is d. Consequently, the edge $\{b, d\}$ is attached to T and d is visited, then d is assigned to v, and we again return to step (2).

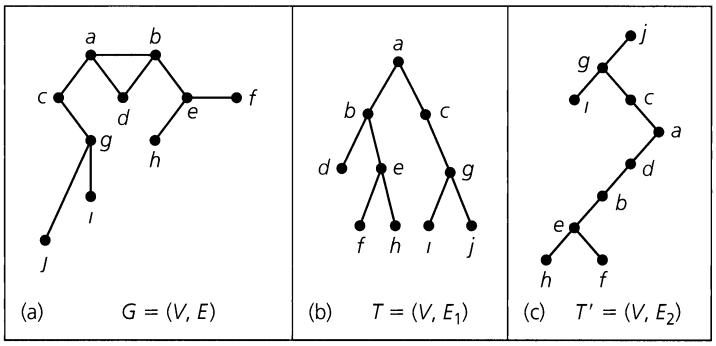


Figure 12.23

This time, however, there is no new vertex that we can obtain from d, because vertices a and b have already been visited. So we go to step (3). But here the value of v is d, not a, and we go to step (4). Now we backtrack from d, assigning the vertex b to v, and then we return to step (2). At this time we add the edge $\{b, e\}$ to T and visit e.

Continuing the process, we attach the edge $\{e, f\}$ (and visit f) and then the edge $\{e, h\}$ (and visit h). But now the vertex h has been assigned to v, and we must backtrack from h to e to b to a. When v is assigned the vertex a this (second) time, the new edge $\{a, c\}$ is obtained and vertex c is visited. Then we proceed to attach the edges $\{c, g\}$, $\{g, i\}$, and $\{g, j\}$ (visiting the vertices g, i, and j, respectively). At this point all of the vertices in G have been visited, and we backtrack from j to g to c to a. With $v = a$ once again we return to step (2) and from there to step (3), where the process terminates.

The resulting tree $T = (V, E_1)$ is shown in part (b) of Fig. 12.23. Part (c) of the figure shows the tree T' that results for the vertex ordering: $j, i, h, g, f, e, d, c, b, a$.

A second method for searching the vertices of a loop-free connected undirected graph is the *breadth-first search*. Here we designate one vertex as the root and fan out to all vertices adjacent to the root. From each child of the root we then fan out to those vertices (not previously visited) that are adjacent to one of these children. As we continue this process, we never list a vertex twice, so no cycle is constructed, and with G finite the process eventually terminates.

We actually used this technique earlier in Example 11.28 of Section 11.5.

A certain data structure proves useful in developing an algorithm for this second searching method. A *queue* is an ordered list wherein items are inserted at one end (called the *rear*) of the list and deleted at the other end (called the *front*). The *first* item inserted *in* the queue is the *first* item that can be taken out of it. Consequently, a queue is referred to as a "first-in, first-out," or FIFO, structure.

As in the depth-first search, we again assign an order to the vertices of our graph.

We start with a loop-free connected undirected graph $G = (V, E)$, where $|V| = n$ and the vertices are ordered as $v_1, v_2, v_3, \ldots, v_n$. The following algorithm generates the (rooted ordered) breadth-first spanning tree T of G for the given order.

Breadth-First Search Algorithm

Step 1: Insert vertex v_1 at the rear of the (initially empty) queue Q and initialize T as the tree made up of this one vertex v_1 (the root of the final version of T). Visit v_1.

Step 2: While the queue Q is not empty, delete the vertex v from the front of Q. Now examine the vertices v_i (for $2 \leq i \leq n$) that are adjacent to v — in the specified order. If v_i has not been visited, perform the following: (1) Insert v_i at the rear of Q; (2) Attach the edge $\{v, v_i\}$ to T; and (3) Visit vertex v_i. [If we examine *all* of the vertices previously in the queue Q and obtain no new edges, then the tree T (generated to this point) is the (rooted ordered) spanning tree for the given order.]

EXAMPLE 12.11

We shall employ the graph of Fig. 12.23(a) with the prescribed order $a, b, c, d, e, f, g, h, i, j$ to illustrate the use of the algorithm for the breadth-first search.

Start with vertex a. Insert a at the rear of (the presently empty) queue Q, initialize T as this one vertex (the root of the resulting tree), and visit vertex a.

In step (2) we now delete a from (the front of) Q and examine the vertices adjacent to a — namely, the vertices b, c, d. (These vertices have not been previously visited.) This results in our (i) inserting vertex b at the rear of Q, attaching the edge $\{a, b\}$ to T, and visiting vertex b; (ii) inserting vertex c at the rear of Q (after b), attaching the edge $\{a, c\}$ to T, and visiting vertex c; and (iii) inserting vertex d at the rear of Q (after c), attaching the edge $\{a, d\}$ to T, and visiting vertex d.

Since the queue Q is not empty, we execute step (2) again. Upon deleting vertex b from the front of Q, we now find that the only vertex adjacent to b (that has not been previously visited) is e. So we insert vertex e at the rear of Q (after d), attach the edge $\{b, e\}$ to T, and visit vertex e. Continuing with vertex c we obtain the new (unvisited) vertex g. So we insert vertex g at the rear of Q (after e), attach the edge $\{c, g\}$ to T, and visit vertex g. And now we delete vertex d from the front of Q. But at this point there are no unvisited vertices adjacent to d, so we then delete vertex e from the front of Q. This vertex leads to the following: inserting vertex f at the rear of Q (after g), attaching the edge $\{e, f\}$ to T, and visiting vertex f. This is followed by: inserting vertex h at the rear of Q (after f), attaching edge $\{e, h\}$ to T, and visiting vertex h. Continuing with vertex g, we insert vertex i at the rear of Q (after h), attach edge $\{g, i\}$ to T, and visit vertex i, and then we insert vertex j at the rear of Q (after i), attach edge $\{g, j\}$ to T, and visit vertex j.

Once again we return to the beginning of step (2). But now when we delete (from the front of Q) and examine each of the vertices f, h, i, and j (in this order), we find no unvisited vertices for any of these four vertices. Consequently, the queue Q now remains empty and the tree T in Fig. 12.24(a) is the breadth-first spanning tree for G, for the order

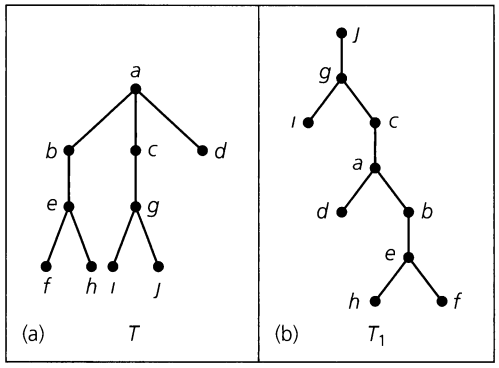


Figure 12.24

prescribed. (The tree T_1, shown in part (b) of the figure, arises for the order $j, i, h, g, f, e, d, c, b, a$.)

Let us apply these ideas on graph searching to one more example.

EXAMPLE 12.12

Let $G = (V, E)$ be an undirected graph (with loops) where the vertices are ordered as $v_1, v_2, \ldots, v_7$. If Fig. 12.25(a) is the adjacency matrix $A(G)$ for G, how can we use this representation of G to determine whether G is connected, without drawing the graph?

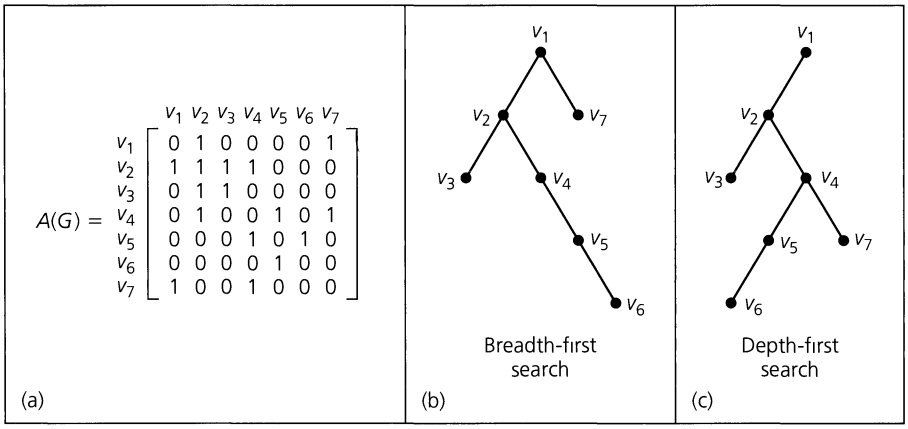


Figure 12.25

Using v_1 as the root, in part (b) of the figure we search the graph by means of its adjacency matrix, using a breadth-first search. [Here we ignore the loops by ignoring any 1's on the main diagonal (extending from the upper left to the lower right).] First we visit the vertices adjacent to v_1, listing them in ascending order according to the subscripts on the v's in $A(G)$. The search continues, and as all vertices in G are reached, G is shown to be connected.

The same conclusion follows from the depth-first search in part (c). The tree here also has v_1 as its root. As the tree branches out to search the graph, it does so by listing the first vertex found adjacent to v_1 according to the row in $A(G)$ for v_1. Likewise, from v_2 the first new vertex in this search is found from $A(G)$ to be v_3. The vertex v_3 is a leaf in this tree because no new vertex can be visited from v_3. As we backtrack to v_2, row 2 of $A(G)$ indicates that v_4 can now be visited from v_2. As this process continues, the connectedness of G follows from part (c) of the figure.

It is time now to return to our main discussion on rooted trees. The following definition generalizes the ideas that were introduced for Example 12.5.

Definition 12.5

Let $T = (V, E)$ be a rooted tree, and let $m \in \mathbf{Z}^+$.

We call T an *m-ary tree* if $od(v) \leq m$ for all $v \in V$. When $m = 2$, the tree is called a *binary tree*.

If $od(v) = 0$ or m, for all $v \in V$, then T is called a *complete m-ary tree*. The special case of $m = 2$ results in a *complete binary tree*.

In a complete m-ary tree, each internal vertex has exactly m children. (Each leaf of this tree still has no children.)

Some properties of these trees are considered in the following theorem.

THEOREM 12.6

Let $T = (V, E)$ be a complete m-ary tree with $|V| = n$. If T has ℓ leaves and i internal vertices, then (a) $n = mi + 1$; (b) $\ell = (m - 1)i + 1$; and (c) $i = (\ell - 1)/(m - 1) = (n - 1)/m$.

Proof: This proof is left for the Section Exercises.

EXAMPLE 12.13

The Wimbledon tennis championship is a single-elimination tournament wherein a player (or doubles team) is eliminated after a single loss. If 27 women compete in the singles championship, how many matches must be played to determine the number-one female player?

Consider the tree shown in Fig. 12.26. With 27 women competing, there are 27 leaves in this complete binary tree, so from Theorem 12.6(c) the number of internal vertices (which is the number of matches) is $i = (\ell - 1)/(m - 1) = (27 - 1)/(2 - 1) = 26$.

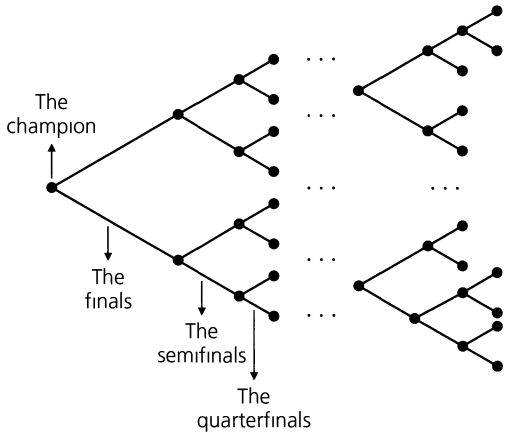


Figure 12.26

EXAMPLE 12.14

A classroom contains 25 microcomputers that must be connected to a wall socket that has four outlets. Connections are made by using extension cords that have four outlets each. What is the least number of cords needed to get these computers set up for class use?

The wall socket is considered the root of a complete m-ary tree for $m = 4$. The microcomputers are the leaves of this tree, so $\ell = 25$. Each internal vertex, except the root, corresponds with an extension cord. So by part (c) of Theorem 12.6, there are $(\ell - 1)/(m - 1) = (25 - 1)/(4 - 1) = 8$ internal vertices. Hence we need $8 - 1$ (where the 1 is subtracted for the root) $= 7$ extension cords.

Definition 12.6

If $T = (V, E)$ is a rooted tree and h is the largest level number achieved by a leaf of T, then T is said to have *height* h. A rooted tree T of height h is said to be *balanced* if the level number of every leaf in T is $h - 1$ or h.

The rooted tree shown in Fig. 12.19 is a balanced tree of height 3. Tree T' in Fig. 12.23(c) has height 7 but is not balanced. (Why?)

The tree for the tournament in Example 12.13 must be balanced so that the tournament will be as fair as possible. If it is not balanced, some competitor will receive more than one bye (an opportunity to advance without playing a match).

Before stating our next theorem, let us recall that for all $x \in \mathbf{R}$, $\lfloor x \rfloor$ denotes the greatest integer in x, or floor of x, whereas $\lceil x \rceil$ designates the ceiling of x.

THEOREM 12.7

Let $T = (V, E)$ be a complete m-ary tree of height h with ℓ leaves. Then $\ell \leq m^h$ and $h \geq \lceil \log_m \ell \rceil$.

Proof: The proof that $\ell \leq m^h$ will be established by induction on h. When $h = 1$, T is a tree with a root and m children. In this case $\ell = m = m^h$, and the result is true. Assume the result true for all trees of height $< h$, and consider a tree T with height h and ℓ leaves. (The level numbers that are possible for these leaves are $1, 2, \ldots, h$, with at least m of the leaves at level h.) The ℓ leaves of T are also the ℓ leaves (total) for the m subtrees T_i, $1 \leq i \leq m$, of T rooted at each of the children of the root. For $1 \leq i \leq m$, let ℓ_i be the number of leaves in subtree T_i. (In the case where leaf and root coincide, $\ell_i = 1$. But since $m \geq 1$ and $h - 1 \geq 0$, we have $m^{h-1} \geq 1 = \ell_i$.) By the induction hypothesis, $\ell_i \leq m^{h(T_i)} \leq m^{h-1}$, where $h(T_i)$ denotes the height of the subtree T_i, and so $\ell = \ell_1 + \ell_2 + \cdots + \ell_m \leq m(m^{h-1}) = m^h$.

With $\ell \leq m^h$, we find that $\log_m \ell \leq \log_m(m^h) = h$, and since $h \in \mathbf{Z}^+$, it follows that $h \geq \lceil \log_m \ell \rceil$.

COROLLARY 12.1

Let T be a balanced complete m-ary tree with ℓ leaves. Then the height of T is $\lceil \log_m \ell \rceil$.

Proof: This proof is left as an exercise.

We close this section with an application that uses a complete ternary ($m = 3$) tree.

EXAMPLE 12.15

Decision Trees. There are eight coins (identical in appearance) and a pan balance. If exactly one of these coins is counterfeit and heavier than the other seven, find the counterfeit coin.

Let the coins be labeled $1, 2, 3, \ldots, 8$. In using the pan balance to compare sets of coins there are three outcomes to consider: (a) the two sides balance to indicate that the coins in the two pans are not counterfeit; (b) the left pan of the balance goes down, indicating that the counterfeit coin is in the left pan; or (c) the right pan goes down, indicating that it holds the counterfeit coin.

In Fig. 12.27(a), we search for the counterfeit coin by first balancing coins 1, 2, 3, 4 against 5, 6, 7, 8. If the balance tips to the right, we follow the right branch from the root to then analyze coins 5, 6 against 7, 8. If the balance tips to the left, we test coins 1, 2 against 3, 4. At each successive level, we have half as many coins to test, so at level 3 (after three weighings) the heavier counterfeit coin has been identified.

The tree in part (b) of the figure finds the heavier coin in two weighings. The first weighing balances coins 1, 2, 3 against 6, 7, 8. Three possible outcomes can occur: (i) the balance tips to the right, indicating that the heavier coin is 6, 7, or 8, and we follow the right branch from the root; (ii) the balance tips to the left and we follow the left branch to find which of 1, 2, 3 is the heavier; or (iii) the pans balance and we follow the center branch to find which of 4, 5 is heavier. At each internal vertex the label indicates which coins are being compared.

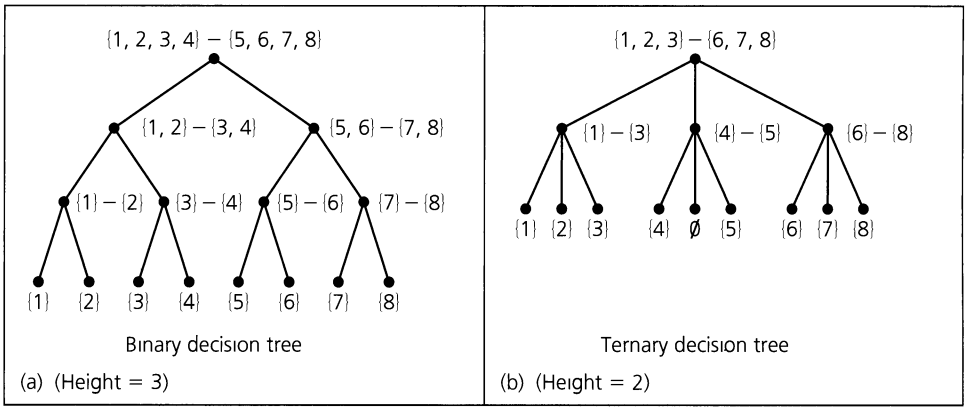


Figure 12.27

Unlike part (a), a conclusion may be deduced in part (b) when a coin is not included in a weighing. Finally, when comparing coins 4 and 5, because equality cannot take place we label the center leaf with $\emptyset$.

In this particular problem, we claim that the height of the complete ternary tree used must be at least 2. With eight coins involved, the tree will have at least eight leaves. Consequently, with $\ell \geq 8$, it follows from Theorem 12.7 that $h \geq \lceil \log_3 \ell \rceil \geq \lceil \log_3 8 \rceil = 2$, so at least two weighing are needed. If n coins are involved, the complete ternary tree will have ℓ leaves where $\ell \geq n$, and its height h satisfies $h \geq \lceil \log_3 n \rceil$.

EXERCISES 12.2

1. Answer the following questions for the tree shown in Fig. 12.28.

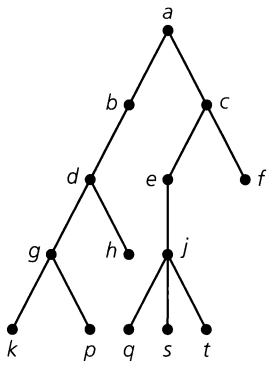


Figure 12.28

a) Which vertices are the leaves?

b) Which vertex is the root?

c) Which vertex is the parent of g?

d) Which vertices are the descendants of c?

e) Which vertices are the siblings of s?

f) What is the level number of vertex f?

g) Which vertices have level number 4?

2. Let $T = (V, E)$ be a binary tree. In Fig. 12.29 we find the subtree of T rooted at vertex p. (The dashed line coming into vertex p indicates that there is more to the tree T than what appears in the figure.) If the level number for vertex u is 37, (a) what are the level numbers for vertices p, s, t, v, w, x, y, and z? (b) how many ancestors does vertex u have? (c) how many ancestors does vertex y have?

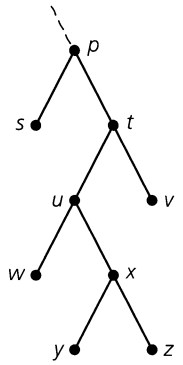


Figure 12.29

3. a) Write the expression $(w + x - y)/(\pi * z^3)$ in Polish notation, using a rooted tree.

b) What is the value of the expression (in Polish notation) $/ \uparrow a - bc + d * ef$, if $a = c = d = e = 2, b = f = 4$?

4. Let $T = (V, E)$ be a rooted tree ordered by a universal address system. (a) If vertex v in T has address 2.1.3.6, what is the smallest number of siblings that v must have? (b) For the vertex v in part (a), find the address of its parent. (c) How many ancestors does the vertex v in part (a) have? (d) With the presence of v in T, what other addresses must there be in the system?

5. For the tree shown in Fig. 12.30, list the vertices according to a preorder traversal, an inorder traversal, and a postorder traversal.

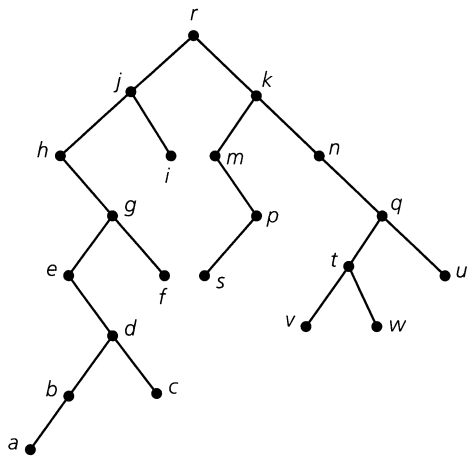


Figure 12.30

6. List the vertices in the tree shown in Fig. 12.31 when they are visited in a preorder traversal and in a postorder traversal.

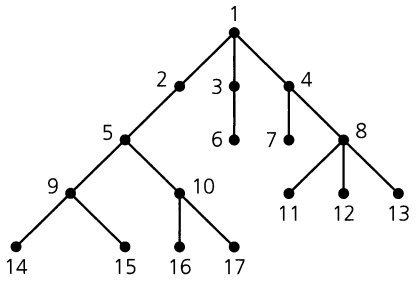


Figure 12.31

7. a) Find the depth-first spanning tree for the graph shown in Fig. 11.72(a) if the order of the vertices is given as (i) a, b, c, d, e, f, g, h; (ii) h, g, f, e, d, c, b, a; (iii) a, b, c, d, h, g, f, e.

b) Repeat part (a) for the graph shown in Fig. 11.85(i).

8. Find the breadth-first spanning trees for the graphs and prescribed orders given in Exercise 7.

9. Let $G = (V, E)$ be an undirected graph with adjacency matrix $A(G)$ as shown here.

$$A(G) = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \\ v_8 \end{array} \begin{array}{c} \begin{array}{cccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Use a breadth-first search based on $A(G)$ to determine whether G is connected.

10. a) Let $T = (V, E)$ be a binary tree. If $|V| = n$, what is the maximum height that T can attain?

b) If $T = (V, E)$ is a complete binary tree and $|V| = n$, what is the maximum height that T can reach in this case?

11. Prove Theorem 12.6 and Corollary 12.1.

12. With m, n, i, ℓ as in Theorem 12.6, prove that

a) $n = (m\ell - 1)/(m - 1)$. **b)** $\ell = [(m - 1)n + 1]/m$.

13. a) A complete ternary (or 3-ary) tree $T = (V, E)$ has 34 internal vertices. How many edges does T have? How many leaves?

b) How many internal vertices does a complete 5-ary tree with 817 leaves have?

14. The complete binary tree $T = (V, E)$ has $V = \{a, b, c, \ldots, i, j, k\}$. The postorder listing of V yields $d, e, b, h, i, f, j, k, g, c, a$. From this information draw T if (a) the height of T is 3; (b) the height of the left subtree of T is 3.

15. For $m \geq 3$, a complete m-ary tree can be transformed into a complete binary tree by applying the idea shown in Fig. 12.32.

a) Use this technique to transform the complete ternary decision tree shown in Fig. 12.27(b).

b) If T is a complete quaternary tree of height 3, what is the maximum height that T can have after it is transformed into a complete binary tree? What is the minimum height?

c) Answer part (b) if T is a complete m-ary tree of height h.

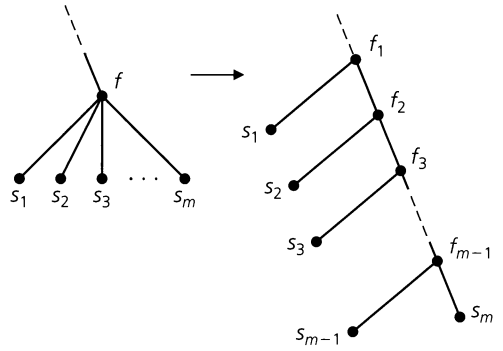


Figure 12.32

16. **a)** At a men's singles tennis tournament, each of 25 players brings a can of tennis balls. When a match is played, one can of balls is opened and used, then kept by the loser. The winner takes the unopened can on to his next match. How many cans of tennis balls will be opened during this tournament? How many matches are played in the tournament?

b) In how many matches did the tournament champion play?

17. What is the maximum number of internal vertices that a complete quaternary tree of height 8 can have? What is the number for a complete m-ary tree of height h?

18. On the first Sunday of 2003 Rizzo and Frenchie start a chain letter, each of them sending five letters (to ten different friends between them). Each person receiving the letter is to send five copies to five new people on the Sunday following the letter's arrival. After the first seven Sundays have passed, what is the total number of chain letters that have been mailed? How many were mailed on the last three Sundays?

19. Use a complete ternary decision tree to repeat Example 12.15 for a set of 12 coins, exactly one of which is heavier (and counterfeit).

20. Let $T = (V, E)$ be a balanced complete m-ary tree of height $h \geq 2$. If T has ℓ leaves and b_{h-1} internal vertices at level $h - 1$, explain why $\ell = m^{h-1} + (m - 1)b_{h-1}$.

21. Consider the complete binary trees on 31 vertices. (Here we distinguish left from right as in Example 12.9.) How many of these trees have 11 vertices in the left subtree of the root? How many have 21 vertices in the right subtree of the root?

22. For $n \geq 0$, let a_n count the number of complete binary trees on $2n + 1$ vertices. (Here we distinguish left from right as in Example 12.9.) How is a_{n+1} related to $a_0, a_1, a_2, \ldots, a_{n-1}, a_n$?

23. Consider the following algorithm where the input is a rooted tree with root r.

Step 1: Push r onto the (empty) stack
Step 2: While the stack is not empty
    Pop the vertex at the top of
        the stack and record its label
    Push the children — going from
        right to left — of this vertex
        onto the stack

(The stack data structure was explained in Example 10.43).

What is the output when this algorithm is applied to (a) the tree in Fig. 12.19? (b) any rooted tree?

24. Consider the following algorithm where the input is a rooted tree with root r.

Step 1: Push r onto the (empty) stack
Step 2: While the stack is not empty
    If the entry at the top of the stack is
        not marked
        Then mark it and push its
            children — right to left — onto
            the stack
    Else
        Pop the vertex at the top of the
            stack and record its label

What is the output when the algorithm is applied to (a) the tree in Fig. 12.19? (b) any rooted tree?

12.3 Trees and Sorting

In Example 10.5, the bubble sort was introduced. There we found that the number of comparisons needed to sort a list of n items is $n(n - 1)/2$. Consequently, this algorithm determines a function $h: \mathbf{Z}^+ \to \mathbf{R}$ defined by $h(n) = n(n - 1)/2$. This is the (worst-case) time-complexity function for the algorithm, and we often express this by writing $h \in O(n^2)$. Consequently, the bubble sort is said to require $O(n^2)$ comparisons. We interpret this to mean that for large n, the number of comparisons is bounded above by cn^2, where c is a constant that is generally not specified because it depends on such factors as the compiler and the computer that are used.

In this section we shall study a second method for sorting a given list of n items into ascending order. The method is called the *merge sort*, and we shall find that the order of its worst-case time-complexity function is $O(n \log_2 n)$. This will be accomplished in the following manner:

1) First we shall measure the number of comparisons needed when n is a power of 2. Our method will employ a pair of balanced complete binary trees.

2) Then we shall cover the case for general n by using the optional material on divide-and-conquer algorithms in Section 10.6.

For the case where n is an arbitrary positive integer, we start by considering the following procedure.

Given a list of n items to sort into ascending order, the *merge sort* recursively splits the given list and all subsequent sublists in half (or as close as possible to half) until each sublist contains a single element. Then the procedure merges these sublists in ascending order until the original n items have been so sorted. The splitting and merging processes can best be described by a pair of balanced complete binary trees, as in the next example.

EXAMPLE 12.16

Merge Sort. Using the merge sort, Fig. 12.33 sorts the list 6, 2, 7, 3, 4, 9, 5, 1, 8. The tree at the top of the figure shows how the process first splits the given list into sublists of size 1. The merging process is then outlined by the tree at the bottom of the figure.

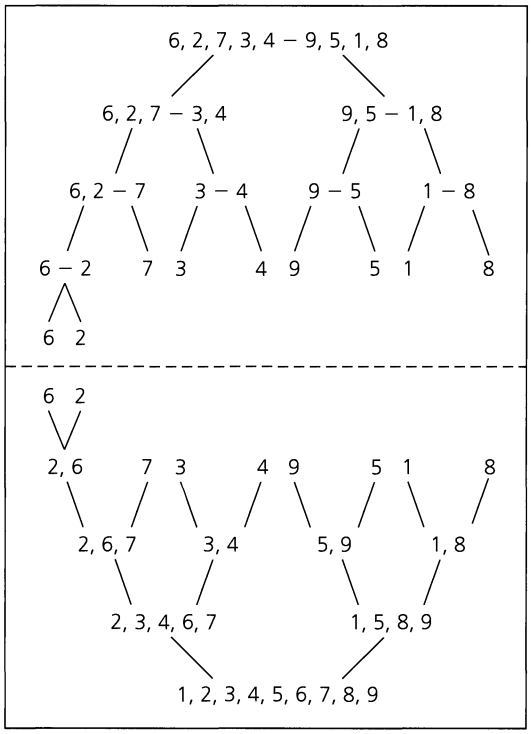


Figure 12.33

To compare the merge sort to the bubble sort, we want to determine its (worst-case) time-complexity function. The following lemma will be needed for this task.

LEMMA 12.1

Let L_1 and L_2 be two sorted lists of ascending numbers, where L_i contains n_i elements, for $i = 1, 2$. Then L_1 and L_2 can be merged into one ascending list L using at most $n_1 + n_2 - 1$ comparisons.

Proof: To merge L_1, L_2 into list L, we perform the following algorithm.

Step 1: Set L equal to the empty list $\emptyset$.

Step 2: Compare the first elements in L_1, L_2. Remove the smaller of the two from the list it is in and place it at the end of L.

Step 3: For the *present* lists L_1, L_2 [one change is made in one of these lists each time step (2) is executed], there are two considerations.

a) If either of L_1, L_2 is empty, then the other list is concatenated to the end of L. This completes the merging process.

b) If not, return to step (2).

Each comparison of a number from L_1 with one from L_2 results in the placement of an element at the end of list L, so there cannot be more than $n_1 + n_2$ comparisons. When one of the lists L_1 or L_2 becomes empty no further comparisons are needed, so the maximum number of comparisons needed is $n_1 + n_2 - 1$.

To determine the (worst-case) time-complexity function of the merge sort, consider a list of n elements. For the moment, we do not treat the general problem, assuming here that $n = 2^h$.[†] In the splitting process, the list of 2^h elements is first split into two sublists of size 2^{h-1}. (These are the level 1 vertices in the tree representing the splitting process.) As the process continues, each successive list of size 2^{h-k}, $h > k$, is at level k and splits into two sublists of size $(1/2)(2^{h-k}) = 2^{h-k-1}$. At level h the sublists each contain $2^{h-h} = 1$ element.

Reversing the process, we first merge the $n = 2^h$ leaves into 2^{h-1} ordered sublists of size 2. These sublists are at level $h - 1$ and require $(1/2)(2^h) = 2^{h-1}$ comparisons (one per pair). As this merging process continues, at each of the 2^k vertices at level k, $1 \leq k < h$, there is a sublist of size 2^{h-k}, obtained from merging the two sublists of size 2^{h-k-1} at its children (on level $k + 1$). From Lemma 12.1, this merging requires at most $2^{h-k-1} + 2^{h-k-1} - 1 = 2^{h-k} - 1$ comparisons. When the children of the root are reached, there are two sublists of size 2^{h-1} (at level 1). To merge these sublists into the final list requires at most $2^{h-1} + 2^{h-1} - 1 = 2^h - 1$ comparisons.

Consequently, for $1 \leq k \leq h$, at level k there are 2^{k-1} pairs of vertices. At each of these vertices is a sublist of size 2^{h-k}, so it takes at most $2^{h-k+1} - 1$ comparisons to merge each pair of sublists. With 2^{k-1} pairs of vertices at level k, the total number of comparisons at level k is at most $2^{k-1}(2^{h-k+1} - 1)$. When we sum over all levels k, where $1 \leq k \leq h$, we find that the total number of comparisons is at most

$$\sum_{k=1}^{h} 2^{k-1}(2^{h-k+1} - 1) = \sum_{k=0}^{h-1} 2^k(2^{h-k} - 1) = \sum_{k=0}^{h-1} 2^h - \sum_{k=0}^{h-1} 2^k = h \cdot 2^h - (2^h - 1).$$

With $n = 2^h$, we have $h = \log_2 n$ and

$$h \cdot 2^h - (2^h - 1) = n \log_2 n - (n - 1) = n \log_2 n - n + 1,$$

[†]The result obtained here for $n = 2^h$, $h \in \mathbf{N}$, is actually true for all $n \in \mathbf{Z}^+$. However, the derivation for general n requires the optional material in Section 10.6. That is why this counting argument is included here—for the benefit of those readers who did not cover Section 10.6.

where $n \log_2 n$ is the dominating term for large n. Thus the (worst-case) time-complexity function for this sorting procedure is $g(n) = n \log_2 n - n + 1$ and $g \in O(n \log_2 n)$, for $n = 2^h$, $h \in \mathbf{Z}^+$. Hence the number of comparisons needed to merge sort a list of n items is bounded above by $dn \log_2 n$ for some constant d, and for all $n \geq n_0$, where n_0 is some particular (large) positive integer.

To show that the order of the merge sort is $O(n \log_2 n)$ for all $n \in \mathbf{Z}^+$, our second approach will use the result of Exercise 9 from Section 10.6. We state that now:

Let $a, b, c \in \mathbf{Z}^+$, with $b \geq 2$. If $g: \mathbf{Z}^+ \to \mathbf{R}^+ \cup \{0\}$ is a monotone increasing function, where

$$g(1) \leq c,$$
$$g(n) \leq ag(n/b) + cn, \quad \text{for } n = b^h, h \in \mathbf{Z}^+,$$

then for the case where $a = b$, we have $g \in O(n \log n)$, for *all* $n \in \mathbf{Z}^+$. (The base for the log function may be any real number greater than 1. Here we shall use the base 2.)

Before we can apply this result to the merge sort, we wish to formulate this sorting process (illustrated in Fig. 12.33) as a precise algorithm. To do so, we call the procedure outlined in Lemma 12.1 the "merge" algorithm. Then we shall write "merge (L_1, L_2)" in order to represent the application of that procedure to the lists L_1, L_2, which are in ascending order.

The algorithm for merge sort is a recursive procedure because it may invoke itself. Here the input is an array (called List) of n items, such as real numbers.

The MergeSort Algorithm

Step 1: If $n = 1$, then List is already sorted and the process terminates. If $n > 1$, then go to step (2).

Step 2: (Divide the array and sort the subarrays.) Perform the following:

1) Assign m the value $\lfloor n/2 \rfloor$.

2) Assign to List 1 the subarray

$$\text{List}[1], \text{List}[2], \ldots, \text{List}[m].$$

3) Assign to List 2 the subarray

$$\text{List}[m+1], \text{List}[m+2], \ldots, \text{List}[n].$$

4) Apply MergeSort to List 1 (of size m) and to List 2 (of size $n - m$).

Step 3: Merge (List 1, List 2).

The function $g: \mathbf{Z}^+ \to \mathbf{R}^+ \cup \{0\}$ will measure the (worst-case) time-complexity for this algorithm by counting the maximum number of comparisons needed to merge sort an array of n items. For $n = 2^h$, $h \in \mathbf{Z}^+$, we have

$$g(n) = 2g(n/2) + [(n/2) + (n/2) - 1].$$

The term $2g(n/2)$ results from step (2) of the MergeSort algorithm, and the summand $[(n/2) + (n/2) - 1]$ follows from step (3) of the algorithm and Lemma 12.1.

With $g(1) = 0$, the preceding equation provides the inequalities

$$g(1) = 0 \leq 1,$$
$$g(n) = 2g(n/2) + (n-1) \leq 2g(n/2) + n, \quad \text{for } n = 2^h,\ h \in \mathbf{Z}^+.$$

We also observe that $g(1) = 0$, $g(2) = 1$, $g(3) = 3$, and $g(4) = 5$, so $g(1) \leq g(2) \leq g(3) \leq g(4)$. Consequently, it appears that g may be a monotone increasing function. The proof that it is monotone increasing is similar to that given for the time-complexity function of binary search. This follows Example 10.49 in Section 10.6, so we leave the details showing that g is monotone increasing to the Section Exercises.

Now with $a = b = 2$ and $c = 1$, the result stated earlier implies that $g \in O(n \log_2 n)$ for *all* $n \in \mathbf{Z}^+$.

Although $n \log_2 n \leq n^2$ for all $n \in \mathbf{Z}^+$, it does *not* follow that because the bubble sort is $O(n^2)$ and the merge sort is $O(n \log_2 n)$, the merge sort is more efficient than the bubble sort for all $n \in \mathbf{Z}^+$. The bubble sort requires less programming effort and generally takes less time than the merge sort for small values of n (depending on factors such as the programming language, the compiler, and the computer). However, as n increases, the ratio of the worst-case running times, as measured by $(cn^2)/(dn \log_2 n) = (c/d)(n/\log_2 n)$, gets arbitrarily large. Consequently, as the input list increases in size, the $O(n^2)$ algorithm (bubble sort) takes significantly more time than the $O(n \log_2 n)$ algorithm (merge sort).

For more on sorting algorithms and their time-complexity functions, the reader should examine [1], [3], [4], [7], and [8] in the chapter references.

EXERCISES 12.3

1. a) Give an example of two lists L_1, L_2, each of which is in ascending order and contains five elements, and where nine comparisons are needed to merge L_1, L_2 by the algorithm given in Lemma 12.1.

b) Let $m, n \in \mathbf{Z}^+$ with $m < n$. Give an example of two lists L_1, L_2, each of which is in ascending order, where L_1 has m elements, L_2 has n elements, and $m + n - 1$ comparisons are needed to merge L_1, L_2 by the algorithm given in Lemma 12.1.

2. Apply the merge sort to each of the following lists. Draw the splitting and merging trees for each application of the procedure.

a) $-1, 0, 2, -2, 3, 6, -3, 5, 1, 4$

b) $-1, 7, 4, 11, 5, -8, 15, -3, -2, 6, 10, 3$

3. Related to the merge sort is a somewhat more efficient procedure called the *quick sort*. Here we start with a list $L: a_1, a_2, \ldots, a_n$, and use a_1 as a pivot to develop two sublists L_1 and L_2 as follows. For $i > 1$, if $a_i < a_1$, place a_i at the end of the first list being developed (this is L_1 at the end of the process); otherwise, place a_i at the end of the second list L_2.

After all a_i, $i > 1$, have been processed, place a_1 at the end of the first list. Now apply quick sort recursively to each of the lists L_1 and L_2 to obtain sublists L_{11}, L_{12}, L_{21}, and L_{22}. Continue the process until each of the resulting sublists contains one element. The sublists are then ordered, and their concatenation gives the ordering sought for the original list L.

Apply quick sort to each list in Exercise 2.

4. Prove that the function g used in the second method to analyze the (worst-case) time-complexity of the merge sort is monotone increasing.

12.4 Weighted Trees and Prefix Codes

Among the topics to which discrete mathematics is applied, coding theory is one wherein different finite structures play a major role. These structures enable us to represent and transmit information that is coded in terms of the symbols in a given alphabet. For instance, the way we most often code, or represent, characters internally in a computer is by means of strings of fixed length, using the symbols 0 and 1.

The codes developed in this section, however, will use strings of different lengths. Why a person should want to develop such a coding scheme and how the scheme can be constructed will be our major concerns in this section.

Suppose we wish to develop a way to represent the letters of the alphabet using strings of 0's and 1's. Since there are 26 letters, we should be able to encode these symbols in terms of sequences of five bits, given that $2^4 < 26 < 2^5$. However, in the English (or any other) language, not all letters occur with the same frequency. Consequently, it would be more efficient to use binary sequences of different lengths, with the most frequently occurring letters (such as e, i, t) represented by the shortest possible sequences. For example, consider $S = \{a, e, n, r, t\}$, a subset of the alphabet. Represent the elements of S by the binary sequences

$$a: 01 \qquad e: 0 \qquad n: 101 \qquad r: 10 \qquad t: 1.$$

If the message "*ata*" is to be transmitted, the binary sequence 01101 is sent. Unfortunately, this sequence is also transmitted for the messages "*etn*", "*atet*", and "*an*".

Consider a second encoding scheme, one given by

$$a: 111 \qquad e: 0 \qquad n: 1100 \qquad r: 1101 \qquad t: 10.$$

Here the message "*ata*" is represented by the sequence 11110111 and there are no other possibilities to confuse the situation. What's more, the labeled complete binary tree shown in Fig. 12.34 can be used to decode the sequence 11110111. Starting at the root, traverse the edge labeled 1 to the right child (of the root). Continuing along the next two edges labeled with 1, we arrive at the leaf labeled a. Hence the unique path from the root to the vertex at a is unambiguously determined by the first three 1's in the sequence 11110111. After we return to the root, the next two symbols in the sequence — namely, 10 — determine the unique path along the edge from the root to its right child, followed by the edge from that child to its left child. This terminates at the vertex labeled t. Again returning to the root, the final three bits of the sequence determine the letter a for a second time. Hence the tree "decodes" 11110111 as *ata*.

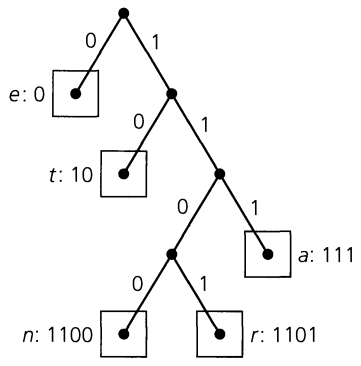


Figure 12.34

Why did the second encoding scheme work out so readily when the first led to ambiguities? In the first scheme, r is represented as 10 and n as 101. If we encounter the symbols 10, how can we determine whether the symbols represent r or the first two symbols of 101, which represent n? The problem is that the sequence for r is a prefix of the sequence for

n. This ambiguity does not occur in the second encoding scheme, suggesting the following definition.

Definition 12.7

A set P of binary sequences (representing a set of symbols) is called a *prefix code* if no sequence in P is the prefix of any other sequence in P.

Consequently, the binary sequences 111, 0, 1100, 1101, 10 constitute a prefix code for the letters a, e, n, r, t, respectively. But how did the complete binary tree of Fig. 12.34 come about? To deal with this problem, we need the following concept.

Definition 12.8

If T is a complete binary tree of height h, then T is called a *full* binary tree if all the leaves in T are at level h.

EXAMPLE 12.17

For the prefix code $P = \{111, 0, 1100, 1101, 10\}$, the longest binary sequence has length 4. Draw the labeled full binary tree of height 4, as shown in Fig. 12.35. The elements of P are assigned to the vertices of this tree as follows. For example, the sequence 10 traces the path from the root r to its right child c_R. Then it continues to the left child of c_R, where the box (marked with the asterisk) indicates completion of the sequence. Returning to the root, the other four sequences are traced out in similar fashion, resulting in the other four boxed vertices. For each boxed vertex remove the subtree (except for the root) that it determines. The resulting pruned tree is the complete binary tree of Fig. 12.34, where no "box" is an ancestor of another "box."

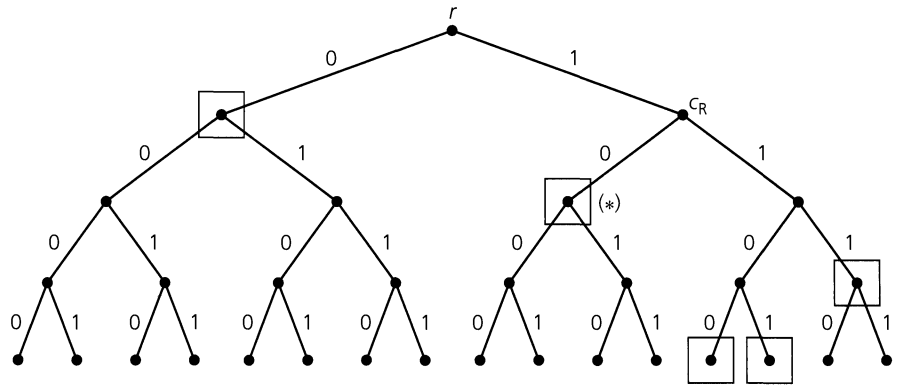


Figure 12.35

We turn now to a method for determining a labeled tree that models a prefix code, where the frequency of occurrence of each symbol in the average text is taken into account — in other words, a prefix code wherein the shorter sequences are used for the more frequently occurring symbols. If there are many symbols, such as all 26 letters of the alphabet, a trial-and-error method for constructing such a tree is not efficient. An elegant construction developed by David A. Huffman (1925–1999) provides a technique for constructing such trees.

The general problem of constructing an efficient tree can be described as follows.

Let $w_1, w_2, \ldots, w_n$ be a set of positive numbers called *weights*, where $w_1 \leq w_2 \leq \cdots \leq w_n$. If $T = (V, E)$ is a complete binary tree with n leaves, assign these weights (in

any one-to-one manner) to the n leaves. The result is called a *complete binary tree for the weights* $w_1, w_2, \ldots, w_n$. The *weight of the tree*, denoted $W(T)$, is defined as $\sum_{i=1}^{n} w_i \ell(w_i)$ where, for each $1 \le i \le n$, $\ell(w_i)$ is the level number of the leaf assigned the weight w_i. The objective is to assign the weights so that $W(T)$ is as small as possible. A complete binary tree T' for these weights is said to be an *optimal* tree if $W(T') \le W(T)$ for any other complete binary tree T for the weights.

Figure 12.36 shows two complete binary trees for the weights 3, 5, 6, and 9. For tree T_1, $W(T_1) = \sum_{i=1}^{4} w_i \ell(w_i) = (3 + 9 + 5 + 6) \cdot 2 = 46$ because each leaf has level number 2. In the case of T_2, $W(T_2) = 3 \cdot 3 + 5 \cdot 3 + 6 \cdot 2 + 9 \cdot 1 = 45$, which we shall find is optimal.

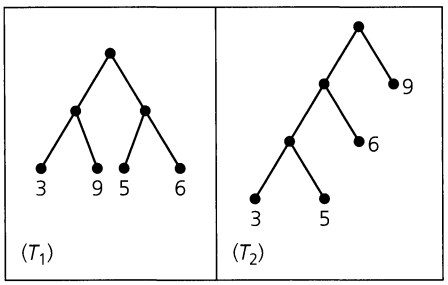


Figure 12.36

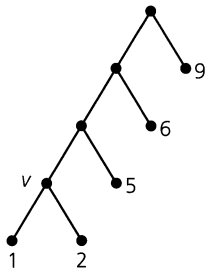


Figure 12.37

The major idea behind Huffman's construction is that in order to obtain an optimal tree T for the n weights $w_1, w_2, w_3, \ldots, w_n$, one considers an optimal tree T' for the $n - 1$ weights $w_1 + w_2, w_3, \ldots, w_n$. (It cannot be assumed that $w_1 + w_2 \le w_3$.) In particular, the tree T' is transformed into T by replacing the leaf v having weight $w_1 + w_2$ by a tree rooted at v of height 1 with left child of weight w_1 and right child of weight w_2. To illustrate, if the tree T_2 in Fig. 12.36 is optimal for the four weights $1 + 2$, 5, 6, 9, then the tree in Fig. 12.37 will be optimal for the five weights 1, 2, 5, 6, 9.

We need the following lemma to establish these claims.

LEMMA 12.2

If T is an optimal tree for the n weights $w_1 \le w_2 \le \cdots \le w_n$, then there exists an optimal tree T' in which the leaves of weights w_1 and w_2 are siblings at the maximal level (in T').

Proof: Let v be an internal vertex of T where the level number of v is maximal for all internal vertices. Let w_x and w_y be the weights assigned to the children x, y of vertex v, with $w_x \le w_y$. By the choice of vertex v, $\ell(w_x) = \ell(w_y) \ge \ell(w_1), \ell(w_2)$. Consider the case of $w_1 < w_x$. (If $w_1 = w_x$, then w_1 and w_x can be interchanged and we would consider the case of $w_2 < w_y$. Applying the following proof to this case, we would find that w_y and w_2 can be interchanged.)

If $\ell(w_x) > \ell(w_1)$, let $\ell(w_x) = \ell(w_1) + j$, for some $j \in \mathbf{Z}^+$. Then $w_1\ell(w_1) + w_x\ell(w_x) = w_1\ell(w_1) + w_x[\ell(w_1) + j] = w_1\ell(w_1) + w_x j + w_x\ell(w_1) > w_1\ell(w_1) + w_1 j + w_x\ell(w_1) = w_1\ell(w_x) + w_x\ell(w_1)$. So $W(T) = w_1\ell(w_1) + w_x\ell(w_x) + \sum_{i \ne 1,x} w_i\ell(w_i) > w_1\ell(w_x) + w_x\ell(w_1) + \sum_{i \ne 1,x} w_i\ell(w_i)$. Consequently, by interchanging the locations of the weights w_1 and w_x, we obtain a tree of smaller weight. But this contradicts the choice of T as an optimal tree. Therefore $\ell(w_x) = \ell(w_1) = \ell(w_y)$. In a similar manner, it can be shown that $\ell(w_y) = \ell(w_2)$, so $\ell(w_x) = \ell(w_y) = \ell(w_1) = \ell(w_2)$. Interchanging the locations of the pair w_1, w_x, and the pair w_2, w_y, we obtain an optimal tree T', where w_1, w_2 are siblings.

From this lemma we see that smaller weights will appear at the higher levels (and thus have higher level numbers) in an optimal tree.

THEOREM 12.8

Let T be an optimal tree for the weights $w_1 + w_2, w_3, \ldots, w_n$, where $w_1 \leq w_2 \leq w_3 \leq \cdots \leq w_n$. At the leaf with weight $w_1 + w_2$ place a (complete) binary tree of height 1 and assign the weights w_1, w_2 to the children (leaves) of this former leaf. The new binary tree T_1 so constructed is then optimal for the weights $w_1, w_2, w_3, \ldots, w_n$.

Proof: Let T_2 be an optimal tree for the weights $w_1, w_2, \ldots, w_n$, where the leaves for weights w_1, w_2 are siblings. Remove the leaves of weights w_1, w_2 and assign the weight $w_1 + w_2$ to their parent (now a leaf). This complete binary tree is denoted T_3 and $W(T_2) = W(T_3) + w_1 + w_2$. Also, $W(T_1) = W(T) + w_1 + w_2$. Since T is optimal, $W(T) \leq W(T_3)$. If $W(T) < W(T_3)$, then $W(T_1) < W(T_2)$, contradicting the choice of T_2 as optimal. Hence $W(T) = W(T_3)$ and, consequently, $W(T_1) = W(T_2)$. So T_1 is optimal for the weights $w_1, w_2, \ldots, w_n$.

Remark. The preceding proof started with an optimal tree T_2 whose existence rests on the fact that there is only a finite number of ways in which we can assign n weights to a complete binary tree with n leaves. Consequently, with a finite number of assignments there is at least one where $W(T)$ is minimal. But finite numbers can be large. This proof establishes the existence of an optimal tree for a set of weights and develops a way for constructing such a tree. To construct such a (Huffman) tree we consider the following algorithm.

Given the n (≥ 2) weights $w_1, w_2, \ldots, w_n$, proceed as follows:

Step 1: Assign the given weights, one each to a set S of n isolated vertices. [Each vertex is the root of a complete binary tree (of height 0) with a weight assigned to it.]

Step 2: While $|S| > 1$ perform the following:

a) Find two trees T, T' in S with the smallest two root weights w, w', respectively.

b) Create the new (complete binary) tree T^* with root weight $w^* = w + w'$ and having T, T' as its left and right subtrees, respectively.

c) Place T^* in S and delete T and T'. [Where $|S| = 1$, the one complete binary tree in S is a Huffman tree.]

We now use this algorithm in the following example.

EXAMPLE 12.18

Construct an optimal prefix code for the symbols a, o, q, u, y, z that occur (in a given sample) with frequencies 20, 28, 4, 17, 12, 7, respectively.

Figure 12.38 shows the construction that follows Huffman's procedure. In part (b) weights 4 and 7 are combined so that we then consider the construction for the weights 11, 12, 17, 20, 28. At each step [in parts (c)–(f) of Fig. 12.38] we create a tree with subtrees rooted at the two smallest weights. These two smallest weights belong to vertices each of which is originally either isolated (a tree with just a root) or the root of a tree obtained earlier in the construction. From the last result, a prefix code is determined as

a: 01 o: 11 q: 1000 u: 00 y: 101 z: 1001.

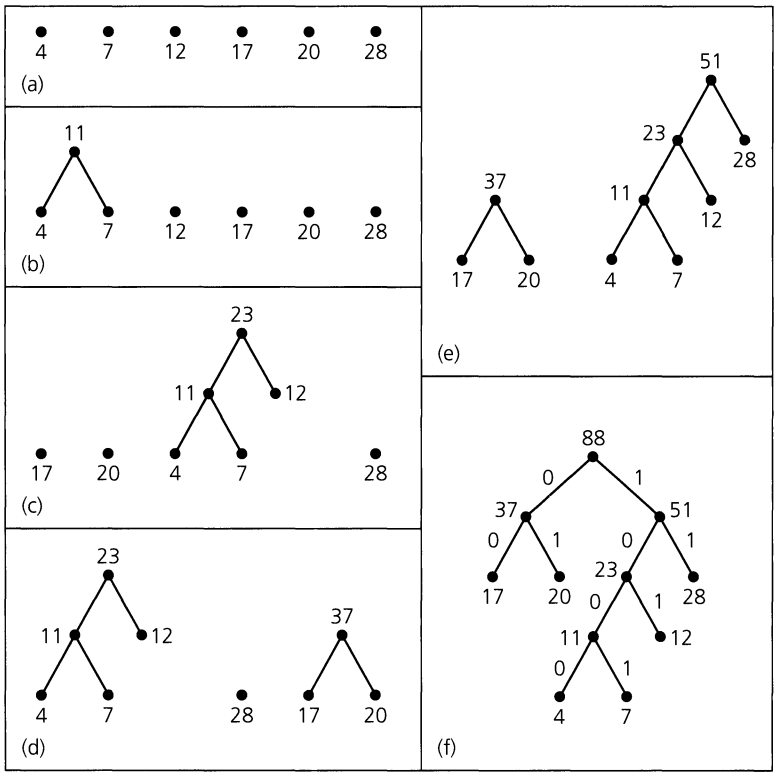


Figure 12.38

Different prefix codes may result from the way the trees T, T' are selected and assigned as the left or right subtree in steps 2(a) and 2(b) in our algorithm and from the assignment of 0 or 1 to the branches (edges) of our final (Huffman) tree.

EXERCISES 12.4

1. For the prefix code given in Fig. 12.34, decode the sequences (a) 1001111101; (b) 10111100110001101; (c) 1101111110010.

2. A code for $\{a, b, c, d, e\}$ is given by a: 00 b: 01 c: 101 d: $x10$ e: $yz1$, where $x, y, z \in \{0, 1\}$. Determine x, y, and z so that the given code is a prefix code.

3. Construct an optimal prefix code for the symbols $a, b, c, \ldots, i, j$ that occur (in a given sample) with respective frequencies 78, 16, 30, 35, 125, 31, 20, 50, 80, 3.

4. How many leaves does a full binary tree have if its height is (a) 3? (b) 7? (c) 12? (d) h?

5. Let $T = (V, E)$ be a complete m-ary tree of height h. This tree is called a *full* m-ary tree if all of its leaves are at level h. If T is a full m-ary tree with height 7 and 279,936 leaves, how many internal vertices are there in T?

6. Let T be a full m-ary tree with height h and v vertices. Determine h in terms of m and v.

7. Using the weights 2, 3, 5, 10, 10, show that the height of a Huffman tree for a given set of weights is not unique. How would you modify the algorithm so as to always produce a Huffman tree of minimal height for the given weights?

8. Let L_i, for $1 \leq i \leq 4$, be four lists of numbers, each sorted in ascending order. The numbers of entries in these lists are 75, 40, 110, and 50, respectively.

a) How many comparisons are needed to merge these four lists by merging L_1 and L_2, merging L_3 and L_4, and then merging the two resulting lists?

b) How many comparisons are needed if we first merge L_1 and L_2, then merge the result with L_3, and finally merge this result with L_4?

c) In order to minimize the total number of comparisons in this merging of the four lists, what order should the merging follow?

d) Extend the result in part (c) to n sorted lists $L_1, L_2, \ldots, L_n$.

12.5 Biconnected Components and Articulation Points

Let $G = (V, E)$ be the loop-free connected undirected graph shown in Fig. 12.39(a), where each vertex represents a communication center. Here an edge $\{x, y\}$ indicates the existence of a communication link between the centers at x and y.

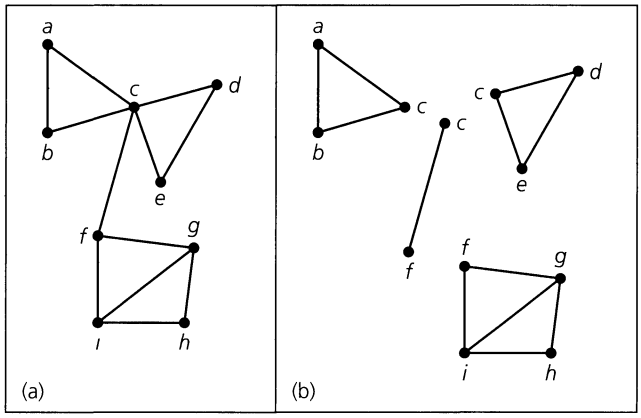


Figure 12.39

By splitting the vertices at c and f, in the suggested fashion, we obtain the collection of subgraphs in part (b) of the figure. These vertices are examples of the following.

Definition 12.9

A vertex v in a loop-free undirected graph $G = (V, E)$ is called an *articulation point* if $\kappa(G - v) > \kappa(G)$; that is, the subgraph $G - v$ has more components than the given graph G.

A loop-free connected undirected graph with no articulation points is called *biconnected.*

A *biconnected component* of a graph is a maximal biconnected subgraph—a biconnected subgraph that is not properly contained in a larger biconnected subgraph.

The graph shown in Fig. 12.39 has the two articulation points, c and f, and its four biconnected components are shown in part (b) of the figure.

In terms of communication centers and links, the articulation points of the graph indicate where the system is most vulnerable. Without articulation points, such a system is more likely to survive disruptions at a communication center, regardless of whether these disruptions are caused by the breakdown of a technical device or by external forces.

The problem of finding the articulation points in a connected graph provides an application for the depth-first spanning tree. The objective here is the development of an algorithm that determines the articulation points of a loop-free connected undirected graph. If no such points exist, then the graph is biconnected. Should such vertices exist, the resulting biconnected components can be used to provide information about such properties as the planarity and chromatic number of the given graph.

The following preliminaries are needed for developing this algorithm.

Returning to Fig. 12.39(a), we see that there are four paths from a to e—namely, (1) $a \to c \to e$; (2) $a \to c \to d \to e$; (3) $a \to b \to c \to e$; and (4) $a \to b \to c \to d \to e$. Now what do these four paths have in common? They all pass through the vertex c, one of the articulation points of G. This observation now motivates our first preliminary result.

LEMMA 12.3

Let $G = (V, E)$ be a loop-free connected undirected graph with $z \in V$. The vertex z is an articulation point of G if and only if there exist distinct $x, y \in V$ with $x \neq z$, $y \neq z$, and such that every path in G connecting x and y contains the vertex z.

Proof: This result follows from Definition 12.9. A proof is requested of the reader in the Section Exercises.

Our next lemma provides an important and useful property of the depth-first spanning tree.

LEMMA 12.4

Let $G = (V, E)$ be a loop-free connected undirected graph with $T = (V, E')$ a depth-first spanning tree of G. If $\{a, b\} \in E$ but $\{a, b\} \notin E'$, then a is either an ancestor or a descendant of b in the tree T.

Proof: From the depth-first spanning tree T, we obtain a preorder listing for the vertices in V. For all $v \in V$, let dfi(v) denote the depth-first index of vertex v—that is, the position of v in the preorder listing. Assume that dfi(a) < dfi(b). Consequently, a is encountered before b in the preorder traversal of T, so a cannot be a descendant of b. If, in addition, vertex a is not an ancestor of b, then b is not in the subtree T_a of T rooted at a. But when we backtrack (through T_a) to a, we find that because $\{a, b\} \in E$, it should have been possible for the depth-first search to go from a to b and to use the edge $\{a, b\}$ in T. This contradiction shows that b is in T_a, so a is an ancestor of b.

If $G = (V, E)$ is a loop-free connected undirected graph, let $T = (V, E')$ be a depth-first spanning tree for G, as shown in Fig. 12.40. By Lemma 12.4, the dotted edge $\{a, b\}$, which is not part of T, indicates an edge that could exist in G. Such an edge is called a *back edge* (relative to T), and here a is an ancestor of b. [Here dfi(a) = 3, whereas dfi(b) = 6.] The dotted edge $\{b, d\}$ in the figure cannot exist in G, also because of Lemma 12.4. Thus all edges of G are either edges in T or back edges (relative to T).

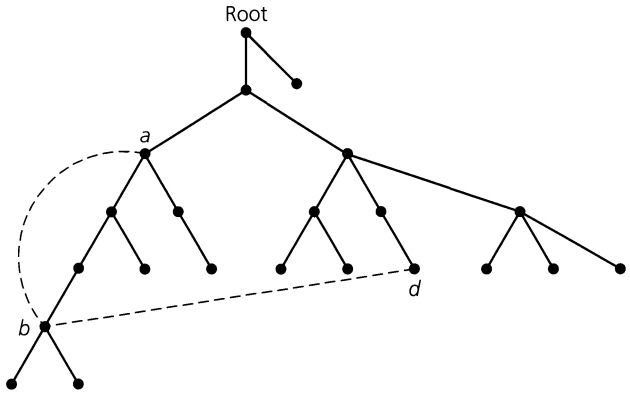


Figure 12.40

Our next example provides further insight into the relationship between the articulation points of a graph G and a depth-first spanning tree of G.

EXAMPLE 12.19

In part (1) of Fig. 12.41 we have a loop-free connected undirected graph $G = (V, E)$. Applying Lemma 12.3 to vertex a, for example, we find that the only path in G from b to i passes through a. In the case of vertex d, we apply the same lemma and consider the vertices a and h. Now we find that although there are four paths from a to h, all four pass through vertex d. Consequently, vertices a and d are two of the articulation points in G. The vertex h is the only other articulation point. Can you find two vertices in G for which all connecting paths (for these vertices) in G pass through h?

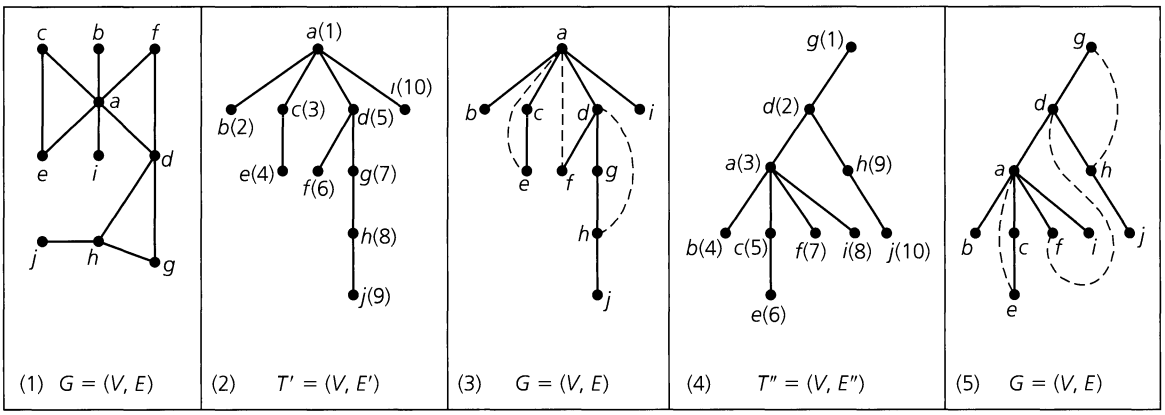


Figure 12.41

Applying the depth-first search algorithm, with the vertices of G ordered alphabetically, in part (2) of Fig. 12.41, we find the depth-first spanning tree $T' = (V, E')$ for G, where a has been chosen as the root. The parenthesized integer next to each vertex indicates the order in which that vertex is visited during the prescribed depth-first search. Part (3) of the figure incorporates the three back edges (relative to T, in G) that are missing from part (2).

For the tree T', the root a, which is an articulation point in G, has more than one child. The articulation point d has a child — namely, g — with no back edge from g or any of its descendants (h and j) to an ancestor of d [as we see in part (3) of Fig. 12.41]. The same is true for the articulation point h. Its child j has (no children and) no back edge to an ancestor of h.

In part (4) of the figure, $T'' = (V, E'')$ is the depth-first spanning tree for the vertices ordered alphabetically once again, but this time vertex g has been chosen as the root. As in part (2) of the figure, the parenthesized integer next to each vertex indicates the order in which that vertex is visited during this depth-first search. The three back edges (relative to T'', in G) that are missing from T'' are shown in part (5) of the figure.

The root g of T'' has only one child and g is not an articulation point in G. Further, for each of the articulation points there is at least one child with no back edge from that child or one of its descendants to an ancestor of the articulation point. To be more specific, from part (5) of Fig. 12.41 we find that for the articulation point a we may use any of the children b, c or i, but not f; for d that child is a; and for h the child is j.

The observations made in Example 12.19 now lead us to the following.

LEMMA 12.5

Let $G = (V, E)$ be a loop-free connected undirected graph with $T = (V, E')$ a depth-first spanning tree of G. If r is the root of T, then r is an articulation point of G if and only if r has at least two children in T.

Proof: If r has only one child — say, c — then all the other vertices of G are descendants of c (and r) in T. So if x, y are two distinct vertices of T, neither of which is r, then in the subtree T_c, rooted at c, there is a path from x to y. Since r is not a vertex in T_c, r is not on this path. Consequently, r is not an articulation point in G — by virtue of Lemma 12.3. Conversely, let r be the root of the depth-first spanning tree T and let c_1, c_2 be children of r. Let x be a vertex in T_{c_1}, the subtree of T rooted at c_1. Similarly, let y be a vertex in T_{c_2}, the subtree of T rooted at c_2. Could there be a path from x to y in G that avoids r? If so, there is an edge $\{v_1, v_2\}$ in G with v_1 in T_{c_1} and v_2 in T_{c_2}. But this contradicts Lemma 12.4.

Our final preliminary result settles the issue of when a vertex, that is not the root of a depth-first spanning tree, is an articulation point of a graph.

LEMMA 12.6

Let $G = (V, E)$ be a loop-free connected undirected graph with $T = (V, E')$ a depth-first spanning tree for G. Let r be the root of T and let $v \in V$, $v \neq r$. Then v is an articulation point of G if and only if there exists a child c of v with no back edge (relative to T, in G) from a vertex in T_c, the subtree rooted at c, to an ancestor of v.

Proof: Suppose that vertex v has a child c such that there is no back edge (relative to T, in G) from a vertex in T_c to an ancestor of v. Then every path (in G) from r to c passes through v. From Lemma 12.3 it then follows that v is an articulation point of G.

To establish the converse, let the nonroot vertex v of T satisfy the following: For each child c of v there is a back edge (relative to T, in G) from a vertex in T_c, the subtree rooted at c, to an ancestor of v. Now let x, $y \in V$ with $x \neq v$, $y \neq v$. We consider the following three possibilities:

1) If neither x nor y is a descendant of v, as in part (1) of Fig. 12.42, delete from T the subtree T_v rooted at v. The resulting subtree (of T) contains x, y and a path from x to y that does not pass through v, so v is not an articulation point of G.

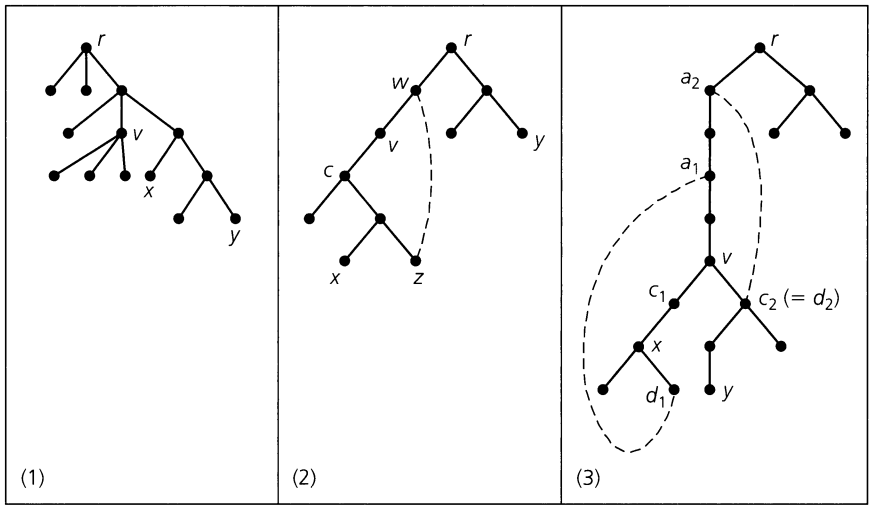


Figure 12.42

2) If one of x, y — say, x — is a descendant of v but y is not, then x is a child of v or a descendant of a child c of v [as in part (2) of Fig. 12.42]. From the hypothesis there is a back edge (relative to T, in G) from some $z \in T_c$ to an ancestor w of v. Since $x, z \in T_c$, there is a path p_1 from x to z (that does not pass through v). Then, as neither w nor y is a descendant of v, from part (1) there is a path p_2 from w to y that does not pass through v. The edges in p_1, p_2 together with the edge $\{z, w\}$ provide a path from x to y that does not pass through v — and once again, v is not an articulation point.

3) Finally, suppose that both x, y are descendants of v, as in part (3) of Fig. 12.42. Here c_1, c_2 are children of v — perhaps, with $c_1 = c_2$ — and x is a vertex in T_{c_1}, the subtree rooted at c_1, while y is a vertex in T_{c_2}, the subtree rooted at c_2. From the hypothesis, there exist back edges $\{d_1, a_1\}$ and $\{d_2, a_2\}$ (relative to T, in G), where d_1, d_2 are descendants of v and a_1, a_2 are ancestors of v. Further, there is a path p_1 from x to d_1 in T_{c_1} and a path p_2 from y to d_2 in T_{c_2}. As neither a_1 nor a_2 is a descendant of v, from part (1) we have a path p (in T) from a_1 to a_2, where p avoids v. Now we can do the following: (i) Go from x to d_1 using path p_1; (ii) Go from d_1 to a_1 on the edge $\{d_1, a_1\}$; (iii) Continue to a_2 using path p; (iv) Go from a_2 to d_2 on the edge $\{a_2, d_2\}$; and (v) Finish at y using the path p_2 from d_2 to y. This provides a path from x to y that avoids v so v is not an articulation point of G and this completes the proof.

Using the results from the preceding four lemmas, we once again start with a loop-free connected undirected graph $G = (V, E)$ with depth-first spanning tree T. For $v \in V$, where v is not the root of T, we let $T_{v,c}$ be the subtree consisting of edge $\{v, c\}$ (c a child of v) together with the tree T_c rooted at c. If there is no back edge from a descendant of v in $T_{v,c}$ to an ancestor of v (and v has at least one ancestor — the root of T), then the splitting of vertex v results in the separation of $T_{v,c}$ from G, and v is an articulation point. If no other articulation points of G occur in $T_{v,c}$, then the addition to $T_{v,c}$ of all other edges in G determined by the vertices in $T_{v,c}$ (the subgraph of G induced by the vertices in $T_{v,c}$) results in a biconnected component of G. A root has no ancestors, and it is an articulation point if and only if it has more than one child.

The depth-first spanning tree preorders the vertices of G. For $x \in V$ let $\text{dfi}(x)$ denote the depth-first index of x in that preorder. If y is a descendant of x, then $\text{dfi}(x) < \text{dfi}(y)$. For y an ancestor of x, $\text{dfi}(x) > \text{dfi}(y)$. Define $\text{low}(x) = \min\{\text{dfi}(y) | y \text{ is adjacent in } G \text{ to either } x \text{ or a descendant of } x\}$. If z is the parent of x (in T), then there are two possibilities to consider:

1) $\text{low}(x) = \text{dfi}(z)$: In this case T_x, the subtree rooted at x, contains no vertex that is adjacent to an ancestor of z by means of a back edge of T. Hence z is an articulation point of G. If T_x contains no articulation points, then T_x together with edge $\{z, x\}$ spans a biconnected component of G (that is, the subgraph of G induced by vertex z and the vertices in T_x is a biconnected component of G). Now remove T_x and the edge $\{z, x\}$ from T, and apply this idea to the remaining subtree of T.

2) $\text{low}(x) < \text{dfi}(z)$: Here there is a descendant of z in T_x that is joined [by a back edge (relative to T, in G)] to an ancestor of z.

To deal in an efficient manner with these ideas, we develop the following algorithm. Let $G = (V, E)$ be a loop-free connected undirected graph.

Step 1: Find the depth-first spanning tree T for G according to a prescribed order. Let $x_1, x_2, \ldots, x_n$ be the vertices of G preordered by T. Then $\text{dfi}(x_j) = j$ for all $1 \leq j \leq n$.

Step 2: Start with x_n and continue back to $x_{n-1}, x_{n-2}, \ldots, x_3, x_2, x_1$, determining $\text{low}(x_j)$, for $j = n, n-1, n-2, \ldots, 3, 2, 1$, recursively, as follows:

a) $\text{low}'(x_j) = \min\{\text{dfi}(z) | z \text{ is adjacent in } G \text{ to } x_j\}$.

b) If $c_1, c_2, \ldots, c_m$ are the children of x_j, then $\text{low}(x_j) = \min\{\text{low}'(x_j), \text{low}(c_1), \text{low}(c_2), \ldots, \text{low}(c_m)\}$. [No problem arises here, for the vertices are examined in the reverse order to the given preorder. Consequently, if c is a child of p, then $\text{low}(c)$ is determined before $\text{low}(p)$.]

Step 3: Let w_j be the parent of x_j in T. If $\text{low}(x_j) = \text{dfi}(w_j)$, then w_j is an articulation point of G, unless w_j is the root of T and w_j has no child in T other than x_j. Moreover, in either situation the subtree rooted at x_j together with the edge $\{w_j, x_j\}$ is part of a biconnected component of G.

EXAMPLE 12.20

We apply this algorithm to the graph $G = (V, E)$ shown in part (i) of Fig. 12.43.

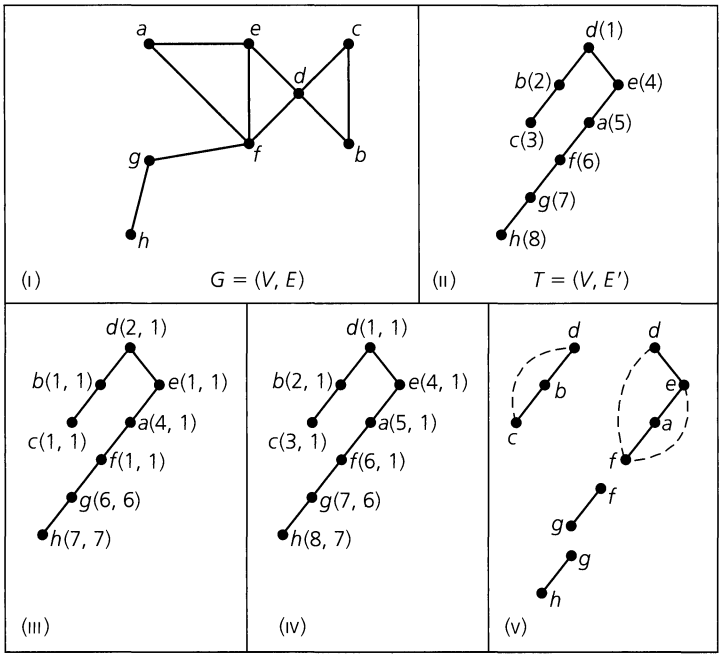


Figure 12.43

In part (ii) of the figure we have the depth-first spanning tree $T = (V, E')$ for G with d as the root. (Here the order followed for the vertices of G is alphabetic.) Next to each vertex v of T [in part (ii)] is the $\text{dfi}(v)$. These labels tell us the order in which the vertices of G are first visited.

For step (2) of the algorithm we go in the reverse order from the depth-first search and start with vertex $h(= x_8)$. Since $\{g, h\} \in E$ and h is not adjacent to any other vertex of G we have $\text{low}'(h) = \text{dfi}(g)\ [= \text{dfi}(x_7)] = 7$. Further, as h has no children, it follows that $\text{low}(h) = \text{low}'(h) = 7$. This accounts for the label (7, 7) $[= (\text{low}'(h), \text{low}(h))]$ next

to h in part (iii) of Fig. 12.43. Continuing next with g, and then f, we obtain the labels (6, 6) for g, and (1, 1) for f, since $\text{low}'(g) = \text{low}(g) = 6$ and $\text{low}'(f) = \text{low}(f) = 1$. Since $\{a, e\}, \{a, f\} \in E$ with $\text{dfi}(e) = 4$ and $\text{dfi}(f) = 6$, for vertex a we have $\text{low}'(a) = \min\{4, 6\} = 4$. Then we find that $\text{low}(a) = \min\{4, \text{low}(f)\} = \min\{4, 1\} = 1$. Hence the label (4, 1) for vertex a. Continuing back through e, c, b, and d, we obtain the labels $(\text{low}'(x_i), \text{low}(x_i))$ for $i = 4, 3, 2, 1$. Consequently, by applying step (2) of the algorithm we arrive at the tree in Fig. 12.43 (iii).

In part (iv) of Fig. 12.43 the ordered pair next to each vertex v is $(\text{dfi}(v), \text{low}(v))$. Applying step (3) of the algorithm to the tree in part (iv), at this point we go in reverse order once again. First we deal with vertex h $(= x_8)$. Since g is the parent of h (in T) and $\text{low}(h) = 7 = \text{dfi}(g)$, g is an articulation point of G and the edge $\{h, g\}$ is a biconnected component of G. Deleting the subtree rooted at g from T, we continue with vertex g $(= x_7)$. Here f is the parent of g (in the tree $T - h$) and $\text{low}(g) = 6 = \text{dfi}(f)$, so f is another articulation point — with edge $\{g, f\}$ the corresponding biconnected component.

Continuing now with the tree $(T - h) - g$, as we go from f to a to e, and then from c to b, we find no new articulation points among the four vertices a, e, c, and b. Since vertex d is the root of T and d has two children — namely, the vertices b and e, it then follows from Lemma 12.5 that d is an articulation point of G. The vertices d, e, a, f induce the biconnected component consisting of the tree edges $\{f, a\}, \{a, e\}, \{e, d\}$ and the back edges (relative to T, in G) $\{f, e\}$ and $\{f, d\}$. Finally, the cycle induced (in G) by the vertices b, c and d provides the fourth biconnected component.

Part (v) of Fig. 12.43 shows the three articulation points g, f, and d, and the four biconnected components of G.

EXERCISES 12.5

1. Find the articulation points and biconnected components for the graph shown in Fig. 12.44.

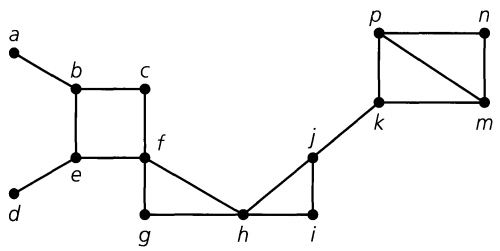


Figure 12.44

2. Prove Lemma 12.3.

3. Let $T = (V, E)$ be a tree with $|V| = n \geq 3$.

a) What are the smallest and the largest numbers of articulation points that T can have? Describe the trees for each of these cases.

b) How many biconnected components does T have in each of the cases in part (a)?

4. a) Let $T = (V, E)$ be a tree. If $v \in V$, prove that v is an articulation point of T if and only if $\deg(v) > 1$.

b) Let $G = (V, E)$ be a loop-free connected undirected graph with $|E| \geq 1$. Prove that G has at least two vertices that are not articulation points.

5. If $B_1, B_2, \ldots, B_k$ are the biconnected components of a loop-free connected undirected graph G, how is $\chi(G)$ related to $\chi(B_i)$, $1 \leq i \leq k$? [Recall that $\chi(G)$ denotes the chromatic number of G, as defined in Section 11.6.]

6. Let $G = (V, E)$ be a loop-free connected undirected graph with biconnected components $B_1, B_2, \ldots, B_8$. For $1 \leq i \leq 8$, the number of distinct spanning trees for B_i is n_i. How many distinct spanning trees exist for G?

7. Let $G = (V, E)$ be a loop-free connected undirected graph with $|V| \geq 3$. If G has no articulation points, prove that G has no pendant vertices.

8. For the loop-free connected undirected graph G in Fig. 12.43(i), order the vertices alphabetically.

a) Determine the depth-first spanning tree T for G with e as the root.

b) Apply the algorithm developed in this section to the tree T in part (a) to find the articulation points and biconnected components of G.

9. Answer the questions posed in the previous exercise but this time order the vertices as h, g, f, e, d, c, b, a and let c be the root of T.

10. Let $G = (V, E)$ be a loop-free connected undirected graph, where $V = \{a, b, c, \ldots, h, i, j\}$. Ordering the vertices alphabetically, the depth-first spanning tree T for G — with a as the root — is given in Fig. 12.45(i). In part (ii) of the figure the ordered pair next to each vertex v provides $(\text{low}'(v), \text{low}(v))$. Determine the articulation points and the spanning trees for the biconnected components of G.

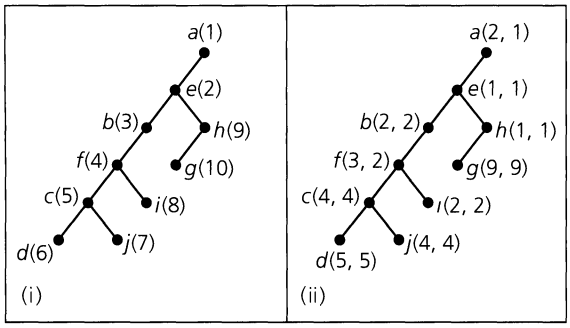


Figure 12.45

11. In step (2) of the algorithm for articulation points, is it really necessary to compute $\text{low}(x_1)$ and $\text{low}(x_2)$?

12. Let $G = (V, E)$ be a loop-free connected undirected graph with $v \in V$.

a) Prove that $\overline{G - v} = \overline{G} - v$.

b) If v is an articulation point of G, prove that v cannot be an articulation point of $\overline{G}$.

13. If $G = (V, E)$ is a loop-free undirected graph, we call G *color-critical* if $\chi(G - v) < \chi(G)$ for all $v \in V$. (We examined such graphs earlier, in Exercise 19 of Section 11.6.) Prove that a color-critical graph has no articulation points.

14. Does the result in Lemma 12.4 remain true if $T = (V, E')$ is a breadth-first spanning tree for $G = (V, E)$?

12.6 Summary and Historical Review

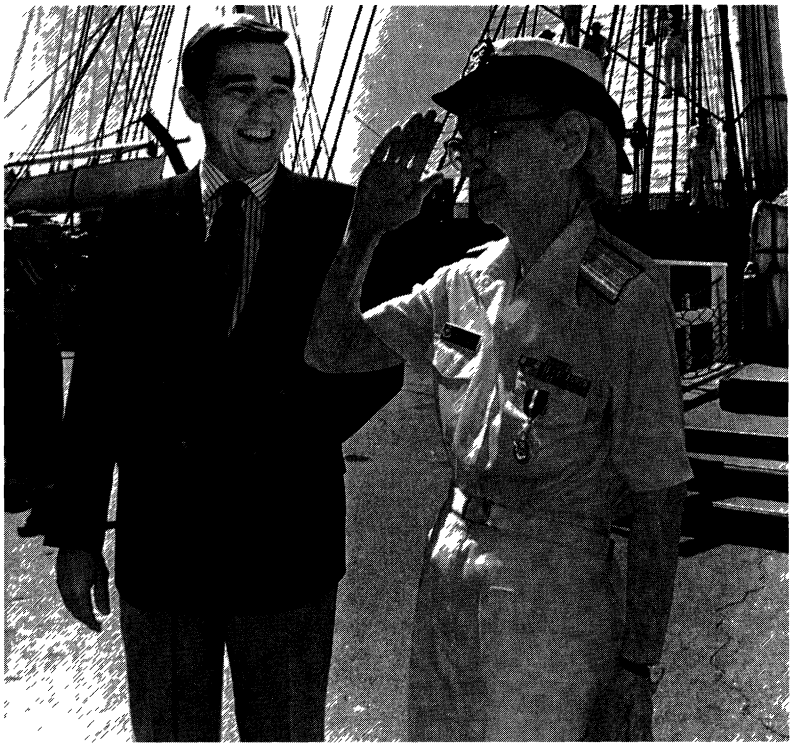
The structure now called a tree first appeared in 1847 in the work of Gustav Kirchhoff (1824–1887) on electrical networks. The concept also appeared at this time in *Geometrie die Lage*, by Karl von Staudt (1798–1867). In 1857 trees were rediscovered by Arthur Cayley (1821–1895), who was unaware of these earlier developments. The first to call the structure a "tree," Cayley used it in applications dealing with chemical isomers. He also investigated the enumeration of certain classes of trees. In his first work on trees, Cayley enumerated unlabeled rooted trees. This was then followed by the enumeration of unlabeled ordered trees. Two of Cayley's contemporaries who also studied trees were Carl Borchardt (1817–1880) and Marie Ennemond Jordan (1838–1922).

Arthur Cayley (1821–1895)

The formula n^{n-2} for the number of labeled trees on n vertices (Exercise 21 at the end of Section 12.1) was discovered in 1860 by Carl Borchardt. Cayley later gave an independent development of the formula, in 1889. Since then, there have been other derivations. These are surveyed in the book by J. W. Moon [10].

The paper by G. Polya [11] is a pioneering work on the enumeration of trees and other combinatorial structures. Polya's theory of enumeration, which we shall see in Chapter 16, was developed in this work. For more on the enumeration of trees, the reader should see Chapter 15 of F. Harary [5]. The article by D. R. Shier [12] provides a labyrinth of several different techniques for calculating the number of spanning trees for $K_{2,n}$.

The high-speed digital computer has proved to be a constant impetus for the discovery of new applications of trees. The first application of these structures was in the manipulation of algebraic formulae. This dates back to 1951 in the work of Grace Murray Hopper. Since then, computer applications of trees have been widely investigated. In the beginning, particular results appeared only in the documentation of specific algorithms. The first general survey of the applications of trees was made in 1961 by Kenneth Iverson as part of a broader survey on data structures. Such ideas as preorder and postorder can be traced to the early 1960s, as evidenced in the work of Zdzislaw Pawlak, Lyle Johnson, and Kenneth Iverson. At this time Kenneth Iverson also introduced the name and the notation, namely $\lceil x \rceil$, for the ceiling of a real number x. Additional material on these orders and the procedures for their implementation on a computer can be found in Chapter 3 of the text by A. V. Aho, J. E. Hopcroft, and J. D. Ullman [1]. In the article by J. E. Atkins, J. S. Dierckman, and K. O'Bryant [2], the notion of preorder is used to develop an optimal route for snow removal.



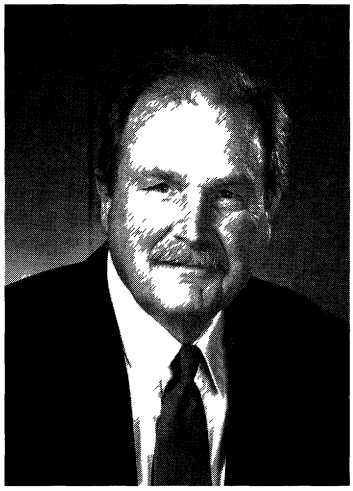
Rear Admiral Grace Murray Hopper (1906–1992) salutes as she and Navy Secretary John Lehman leave the U.S.S *Constitution*.

AP/World Wide Photos

If $G = (V, E)$ is a loop-free undirected graph, then the depth-first search and the breadth-first search (given in Section 12.2) provide ways to determine whether the given graph is connected. The algorithms developed for these searching procedures are also important in developing other algorithms. For example, the depth-first search arises in the algorithm for finding the articulation points and biconnected components of a loop-free connected undirected graph. If $|V| = n$ and $|E| = e$, then it can be shown that both the depth-first search and the breadth-first search have time-complexity $O(\max\{n, e\})$. For most graphs $e > n$, so the algorithms are generally considered to have time-complexity $O(e)$. These ideas are developed in great detail in Chapter 7 of S. Baase and A. Van Gelder [3], where the coverage also includes an analysis of the time-complexity function for the algorithm (of Section 12.5) that determines articulation points (and biconnected components). Chapter 6 of the text by A. V. Aho, J. E. Hopcroft, and J. D. Ullman [1] also deals with the depth-first search, whereas Chapter 7 covers the breadth-first search and the algorithm for articulation points.

More on the properties and computer applications of trees is given in Section 3 of Chapter 2 in the work by D. E. Knuth [7]. Sorting techniques and their use of trees can be further studied in Chapter 11 of A. V. Aho, J. E. Hopcroft, and J. D. Ullman [1] and in Chapter 7 of T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein [4]. An extensive investigation will warrant the coverage found in the text by D. E. Knuth [8].

The technique in Section 12.4 for designing prefix codes is based on methods developed by D. A. Huffman [6].



David A. Huffman

University of Florida, Department of Computer and Information Science and Engineering

Finally, Chapter 7 of C. L. Liu [9] deals with trees, cycles, cut-sets, and the vector spaces associated with these ideas. The reader with a background in linear or abstract algebra should find this material of interest.

REFERENCES

1. Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D. *Data Structures and Algorithms.* Reading, Mass.: Addison-Wesley, 1983.
2. Atkins, Joel E., Dierckman, Jeffrey S., and O'Bryant, Kevin. "A Real Snow Job." *The UMAP Journal*, Fall no. 3 (1990): pp. 231–239.

3. Baase, Sara, and Van Gelder, Allen. *Computer Algorithms: Introduction to Design and Analysis*, 3rd ed. Reading, Mass.: Addison-Wesley, 2000.
4. Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms*, 2nd ed. Boston, Mass.: McGraw-Hill, 2001.
5. Harary, Frank. *Graph Theory*. Reading, Mass.: Addison-Wesley, 1969.
6. Huffman, David A. "A Method for the Construction of Minimum Redundancy Codes." *Proceedings of the IRE* 40 (1952): pp. 1098–1101.
7. Knuth, Donald E. *The Art of Computer Programming*, Vol. 1, 2nd ed. Reading, Mass.: Addison-Wesley, 1973.
8. Knuth, Donald E. *The Art of Computer Programming*, Vol. 3. Reading, Mass.: Addison-Wesley, 1973.
9. Liu, C. L. *Introduction to Combinatorial Mathematics*. New York: McGraw-Hill, 1968.
10. Moon, John Wesley. *Counting Labelled Trees*. Canadian Mathematical Congress, Montreal, Canada, 1970.
11. Polya, George. "Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und Chemische Verbindungen." *Acta Mathematica* 68 (1937): pp. 145–234.
12. Shier, Douglas R. "Spanning Trees: Let Me Count the Ways." *Mathematics Magazine* 73 (2000): pp. 376–381.

SUPPLEMENTARY EXERCISES

1. Let $G = (V, E)$ be a loop-free undirected graph with $|V| = n$. Prove that G is a tree if and only if $P(G, \lambda) = \lambda(\lambda - 1)^{n-1}$.

2. A telephone communication system is set up at a company where 125 executives are employed. The system is initialized by the president, who calls her four vice presidents. Each vice president then calls four other executives, some of whom in turn call four others, and so on. (Each executive who does make a call will actually make four calls.)

a) How many calls are made in reaching all 125 executives?

b) How many executives, aside from the president, are required to make calls?

3. Let T be a complete binary tree with the vertices of T ordered by a preorder traversal. This traversal assigns the label 1 to all internal vertices of T and the label 0 to each leaf. The sequence of 0's and 1's that results from the preorder traversal of T is called the tree's *characteristic sequence*.

a) Find the characteristic sequence for the complete binary tree shown in Fig. 12.17.

b) Determine the complete binary trees for the characteristic sequences

i) 1011001010100 and

ii) 1011110000101011000.

c) What are the last two symbols in the characteristic sequence for all complete binary trees? Why?

4. For $k \in \mathbf{Z}^+$, let $n = 2^k$, and consider the list $L: a_1, a_2, a_3, \ldots, a_n$. To sort L in ascending order, first compare the entries a_i and $a_{i+(n/2)}$, for each $1 \leq i \leq n/2$. For the resulting 2^{k-1} ordered pairs, merge sort the ith and $(i + (n/4))$-th ordered pairs, for each $1 \leq i \leq n/4$. Now do a merge sort on the ith and $(i + (n/8))$-th ordered quadruples, for each $1 \leq i \leq n/8$. Continue the process until the elements of L are in ascending order.

a) Apply this sorting procedure to the list

$$L: 11, 3, 4, 6, -5, 7, 35, -2, 1, 23, 9, 15, 18, 2, -10, 5.$$

b) If $n = 2^k$, how many comparisons at most does this procedure require?

5. Let $G = (V, E)$ be a loop-free undirected graph. If $\deg(v) \geq 2$ for all $v \in V$, prove that G contains a cycle.

6. Let $T = (V, E)$ be a rooted tree with root r. Define the relation $\mathcal{R}$ on V by $x \mathcal{R} y$, for $x, y \in V$, if $x = y$ or if x is on the path from r to y. Prove that $\mathcal{R}$ is a partial order.

7. Let $T = (V, E)$ be a tree with $V = \{v_1, v_2, \ldots, v_n\}$, for $n \geq 2$. Prove that the number of pendant vertices in T is equal to

$$2 + \sum_{\deg(v_i) \geq 3} (\deg(v_i) - 2).$$

8. Let $G = (V, E)$ be a loop-free undirected graph. Define the relation $\mathcal{R}$ on E as follows: If $e_1, e_2 \in E$, then $e_1 \mathcal{R} e_2$ if $e_1 = e_2$ or if e_1 and e_2 are edges of a cycle C in G.

a) Verify that $\mathcal{R}$ is an equivalence relation on E.

b) Describe the partition of E induced by $\mathcal{R}$.

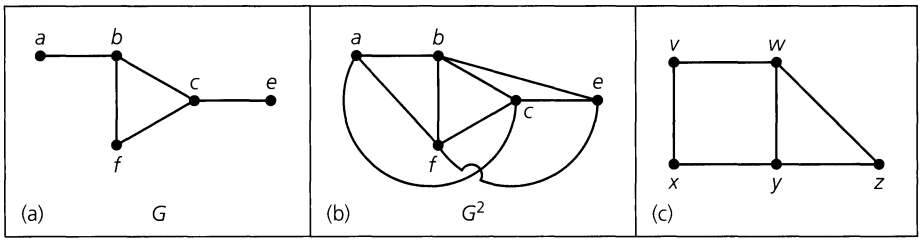


Figure 12.46

9. If $G = (V, E)$ is a loop-free connected undirected graph and $a, b \in V$, then we define the *distance* from a to b (or from b to a), denoted $d(a, b)$, as the length of a shortest path (in G) connecting a and b. (This is the number of edges in a shortest path connecting a and b and is 0 when $a = b$.)

For any loop-free connected undirected graph $G = (V, E)$, the *square* of G, denoted G^2, is the graph with vertex set V (the same as G) and edge set defined as follows: For distinct $a, b \in V$, $\{a, b\}$ is an edge in G^2 if $d(a, b) \leq 2$ (in G). In parts (a) and (b) of Fig. 12.46, we have a graph G and its square.

a) Find the square of the graph in part (c) of the figure.

b) Find G^2 if G is the graph $K_{1,3}$.

c) If G is the graph $K_{1,n}$, for $n \geq 4$, how many edges are added to G in order to construct G^2?

d) For any loop-free connected undirected graph G, prove that G^2 has no articulation points.

10. a) Let $T = (V, E)$ be a complete 6-ary tree of height 8. If T is balanced, but not full, determine the minimum and maximum values for $|V|$.

b) Answer part (a) if $T = (V, E)$ is a complete m-ary tree of height h.

11. The rooted *Fibonacci trees* T_n, $n \geq 1$, are defined recursively as follows:

1) T_1 is the rooted tree consisting of only the root;

2) T_2 is the same as T_1 — it too is a rooted tree that consists of a single vertex; and

3) For $n \geq 3$, T_n is the rooted binary tree with T_{n-1} as its left subtree and T_{n-2} as its right subtree.

The first six rooted Fibonacci trees are shown in Fig. 12.47:

a) For $n \geq 1$, let ℓ_n count the number of leaves in T_n. Find and solve a recurrence relation for ℓ_n.

b) Let i_n count the number of internal vertices for the tree T_n, where $n \geq 1$. Find and solve a recurrence relation for i_n.

c) Determine a formula for v_n, the total number of vertices in T_n, where $n \geq 1$.

12. a) The graph in part (a) of Fig. 12.48 has exactly one spanning tree — namely, the graph itself. The graph in Fig. 12.48(b) has four nonidentical, though isomorphic, spanning trees. In part (c) of the figure we find three of the nonidentical spanning trees for the graph in part (d). Note that T_2 and T_3 are isomorphic, but T_1 is not isomorphic to T_2 (or T_3). How many nonidentical spanning trees exist for the graph in Fig. 12.48(d)?

b) In Fig. 12.48(e) we generalize the graphs in parts (a), (b), and (d) of the figure. For each $n \in \mathbf{Z}^+$, the graph G_n is $K_{2,n}$.

If t_n counts the number of nonidentical spanning trees for G_n, find and solve a recurrence relation for t_n.

13. Let $G = (V, E)$ be the undirected connected "ladder graph" shown in Fig. 12.49. For $n \geq 0$, let a_n count the number of spanning trees of G, whereas b_n counts the number of these spanning trees that contain the edge $\{x_1, y_1\}$.

a) Explain why $a_n = a_{n-1} + b_n$.

b) Find an equation that expresses b_n in terms of a_{n-1} and b_{n-1}.

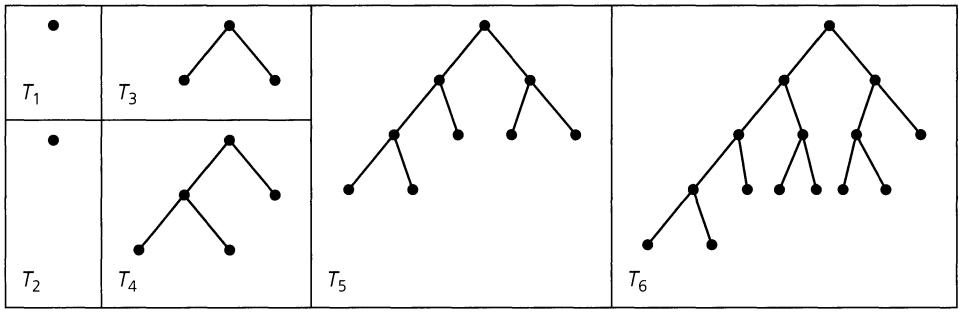


Figure 12.47

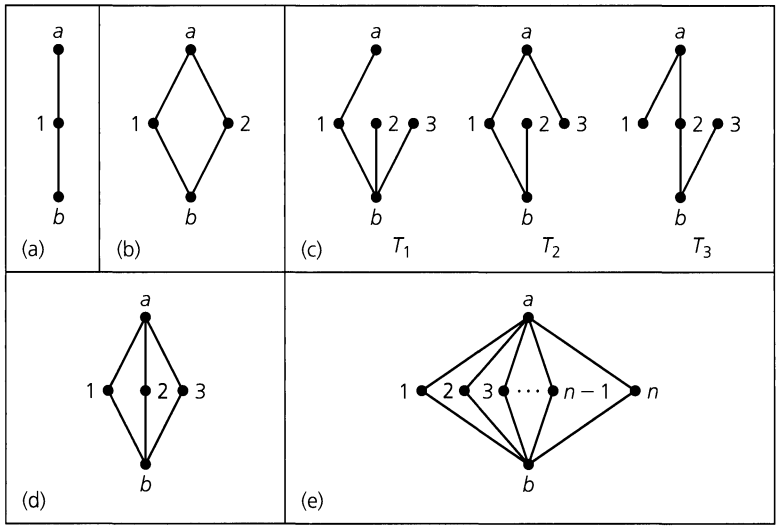


Figure 12.48

c) Use the results in parts (a) and (b) to set up and solve a recurrence relation for a_n.

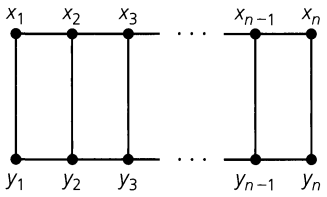


Figure 12.49

14. Let $T = (V, E)$ be a tree where $|V| = v$ and $|E| = e$. The tree T is called *graceful* if it is possible to assign the labels $\{1, 2, 3, \ldots, v\}$ to the vertices of T in such a manner that the induced edge labeling — where each edge $\{i, j\}$ is assigned the label $|i - j|$, for $i, j \in \{1, 2, 3, \ldots, v\}$, $i \neq j$ — results in the e edges being labeled by $1, 2, 3, \ldots, e$.

a) Prove that every path on n vertices, $n \geq 2$, is graceful.

b) For $n \in \mathbf{Z}^+$, $n \geq 2$, show that $K_{1,n}$ is graceful.

c) If $T = (V, E)$ is a tree with $4 \leq |V| \leq 6$, show that T is graceful. (It has been conjectured that every tree is graceful.)

15. For an undirected graph $G = (V, E)$ a subset of I of V is called *independent* when no two vertices in I are adjacent. If, in addition, $I \cup \{x\}$ is not independent for each $x \in V - I$, then we say that I is a *maximal independent* set (of vertices).

The two graphs in Fig. 12.50 are examples of special kinds of trees called caterpillars. In general, a tree $T = (V, E)$ is a *caterpillar* when there is a (maximal) path p such that, for all $v \in V$, either v is on the path p or v is adjacent to a vertex on the path p. This path p is called the *spine* of the caterpillar.

a) How many maximal independent sets of vertices are there for each of the caterpillars in parts (i) and (ii) of Fig. 12.50?

b) For $n \in \mathbf{Z}^+$, with $n \geq 3$, let a_n count the number of maximal independent sets in a caterpillar T whose spine contains n vertices. Find and solve a recurrence relation for a_n. [The reader may wish to reexamine part (a) of Supplementary Exercise 21 in Chapter 11.]

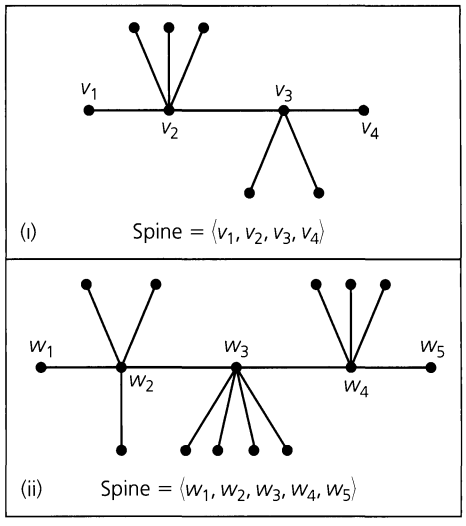


Figure 12.50

16. In part (i) of Fig. 12.51 we find a graceful labeling of the caterpillar shown in part (i) of Fig. 12.50. Find a graceful labeling for the caterpillars in part (ii) of Figs. 12.50 and 12.51.

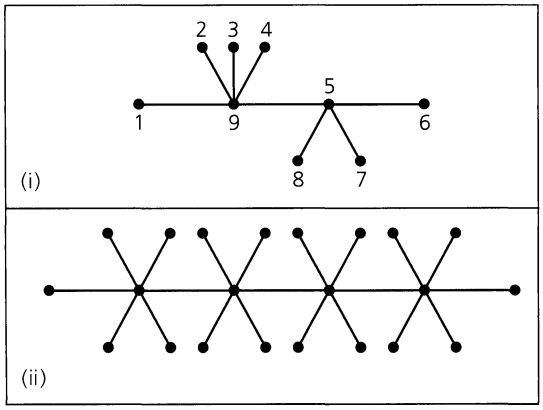


Figure 12.51

17. Develop an algorithm to gracefully label the vertices of a caterpillar with at least two edges.

18. Consider the caterpillar in part (i) of Fig. 12.50. If we label each edge of the spine with a 1 and each of the other edges with a 0, the caterpillar can be represented by a binary string. Here that binary string is 10001001 where the first 1 is for the first (left-most) edge of the spine, the next three 0's are for the (nonspine) edges at v_2, the second 1 is for edge $\{v_2, v_3\}$, the two 0's are for the (nonspine) leaves at v_3, and the final 1 accounts for the third (right-most) edge of the spine.

We also note that the reversal of the binary string 10001001 — namely, 10010001 — corresponds with a second caterpillar that is isomorphic to the one in part (i) of Fig. 12.50.

a) Find the binary strings for each of the caterpillars in part (ii) of Figs. 12.50 and 12.51.

b) Can a caterpillar have a binary string of all 1's?

c) Can the binary string for a caterpillar have only two 1's?

d) Draw all the nonisomorphic caterpillars on five vertices. For each caterpillar determine its binary string. How many of these binary strings are palindromes?

e) Answer the question posed in part (d) upon replacing "five" by "six."

f) For $n \geq 3$, prove that the number of nonisomorphic caterpillars on n vertices is $(1/2)(2^{n-3} + 2^{\lceil (n-3)/2 \rceil}) = 2^{n-4} + 2^{\lfloor (n-4)/2 \rfloor} = 2^{n-4} + 2^{\lfloor n/2 \rfloor - 2}$. (This was first established in 1973 by F. Harary and A. J. Schwenk.)

19. For $n \geq 0$, we want to count the number of ordered rooted trees on $n + 1$ vertices. The five trees in Fig. 12.52(a) cover the case for $n = 3$.

[*Note*: Although the two trees in Fig. 12.52(b) are distinct as binary rooted trees, as ordered rooted trees they are considered the same tree and each is accounted for by the fourth tree in Fig. 12.52(a).]

a) Performing a postorder traversal of each tree in Fig. 12.52(a), we traverse each edge twice — once going down and once coming back up. When we traverse an edge going down, we shall write "1" and when we traverse one coming back up, we shall write "−1." Hence the postorder traversal for the first tree in Fig. 12.52(a) generates the list 1, 1, 1, −1, −1, −1. The list 1, 1, −1, −1, 1, −1 arises for the second tree in part (a) of the figure. Find the corresponding lists for the other three trees in Fig. 12.52(a).

b) Determine the ordered rooted trees on five vertices that generate the lists: (i) 1, −1, 1, 1, −1, 1, −1, −1; (ii) 1, 1, −1, −1, 1, 1, −1, −1; and (iii) 1, −1, 1, −1, 1, 1, −1, −1. How many such trees are there on five vertices?

c) For $n \geq 0$, how many ordered rooted trees are there for $n + 1$ vertices?

20. For $n \geq 1$, let t_n count the number of spanning trees for the *fan* on $n + 1$ vertices. The fan for $n = 4$ is shown in Fig. 12.53.

a) Show that $t_{n+1} = t_n + \sum_{i=0}^{n} t_i$, where $n \geq 1$ and $t_0 = 1$.

b) For $n \geq 2$, show that $t_{n+1} = 3t_n - t_{n-1}$.

c) Solve the recurrence relation in part (b) and show that for $n \geq 1$, $t_n = F_{2n}$, the $2n$th Fibonacci number.

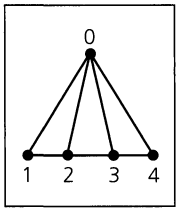


Figure 12.53

21. a) Consider the subgraph of G (in Fig. 12.54) induced by the vertices a, b, c, d. This graph is called a *kite*. How many nonidentical (though some may be isomorphic) spanning trees are there for this kite?

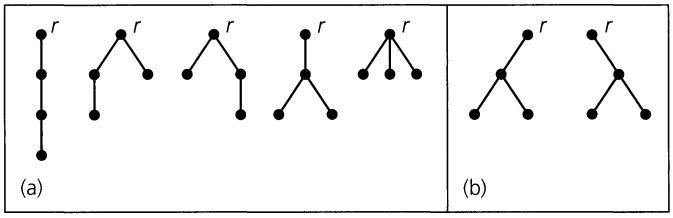


Figure 12.52

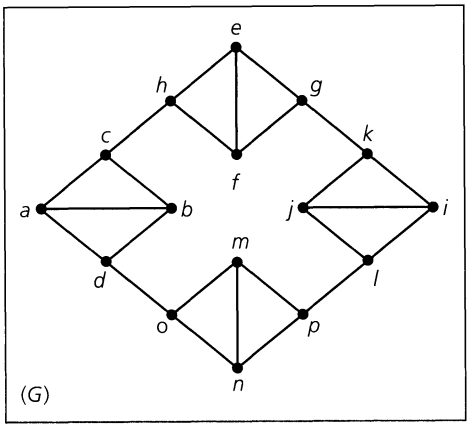


Figure 12.54

b) How many nonidentical (though some may be isomorphic) spanning trees of G do not contain the edge $\{c, h\}$?

c) How many nonidentical (though some may be isomorphic) spanning trees of G contain all four of the edges $\{c, h\}$, $\{g, k\}$, $\{l, p\}$, and $\{d, o\}$?

d) How many nonidentical (though some may be isomorphic) spanning trees exist for G?

e) We generalize the graph G as follows. For $n \geq 2$, start with a cycle on the $2n$ vertices $v_1, v_2, \ldots, v_{2n-1}, v_{2n}$. Replace each of the n edges $\{v_1, v_2\}$, $\{v_3, v_4\}, \ldots,$ $\{v_{2n-1}, v_{2n}\}$ with a (labeled) kite so that the resulting graph is 3-regular. (The case for $n = 4$ appears in Fig. 12.54.) How many nonidentical (though some may be isomorphic) spanning trees are there for this graph?