11

Trees

A connected graph that contains no simple circuits is called a tree. Trees were used as long ago as 1857, when the English mathematician Arthur Cayley used them to count certain types of chemical compounds. Since that time, trees have been employed to solve problems in a wide variety of disciplines, as the examples in this chapter will show.

Trees are particularly useful in computer science, where they are employed in a wide range of algorithms. For instance, trees are used to construct efficient algorithms for locating items in a list. They can be used in algorithms, such as Huffman coding, that construct efficient codes saving costs in data transmission and storage. Trees can be used to study games such as checkers and chess and can help determine winning strategies for playing these games. Trees can be used to model procedures carried out using a sequence of decisions. Constructing these models can help determine the computational complexity of algorithms based on a sequence of decisions, such as sorting algorithms.

Procedures for building trees containing every vertex of a graph, including depth-first search and breadth-first search, can be used to systematically explore the vertices of a graph. Exploring the vertices of a graph via depth-first search, also known as backtracking, allows for the systematic search for solutions to a wide variety of problems, such as determining how eight queens can be placed on a chessboard so that no queen can attack another.

We can assign weights to the edges of a tree to model many problems. For example, using weighted trees we can develop algorithms to construct networks containing the least expensive set of telephone lines linking different network nodes.

11.1   Introduction to Trees

Links

In Chapter 10 we showed how graphs can be used to model and solve many problems. In this chapter we will focus on a particular type of graph called a **tree**, so named because such graphs resemble trees. For example, *family trees* are graphs that represent genealogical charts. Family trees use vertices to represent the members of a family and edges to represent parent–child relationships. The family tree of the male members of the Bernoulli family of Swiss mathematicians is shown in Figure 1. The undirected graph representing a family tree (restricted to people of just one gender and with no inbreeding) is an example of a tree.
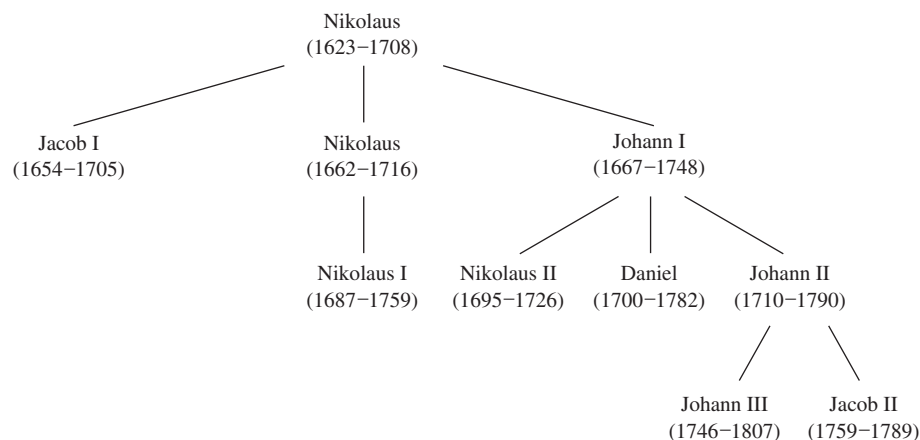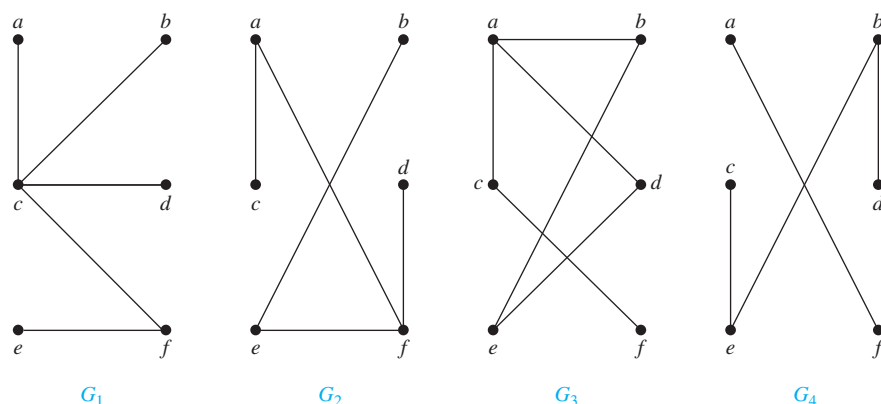

FIGURE 1   **The Bernoulli Family of Mathematicians.**

FIGURE 2   Examples of Trees and Graphs That Are Not Trees.

DEFINITION 1   A *tree* is a connected undirected graph with no simple circuits.

Because a tree cannot have a simple circuit, a tree cannot contain multiple edges or loops. Therefore any tree must be a simple graph.

**EXAMPLE 1**   Which of the graphs shown in Figure 2 are trees?

Solution: G_1 and G_2 are trees, because both are connected graphs with no simple circuits. G_3 is not a tree because e, b, a, d, e is a simple circuit in this graph. Finally, G_4 is not a tree because it is not connected.   ◄

Any connected graph that contains no simple circuits is a tree. What about graphs containing no simple circuits that are not necessarily connected? These graphs are called **forests** and have the property that each of their connected components is a tree. Figure 3 displays a forest.

Trees are often defined as undirected graphs with the property that there is a unique simple path between every pair of vertices. Theorem 1 shows that this alternative definition is equivalent to our definition.

THEOREM 1   An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

This is one graph with three connected components.
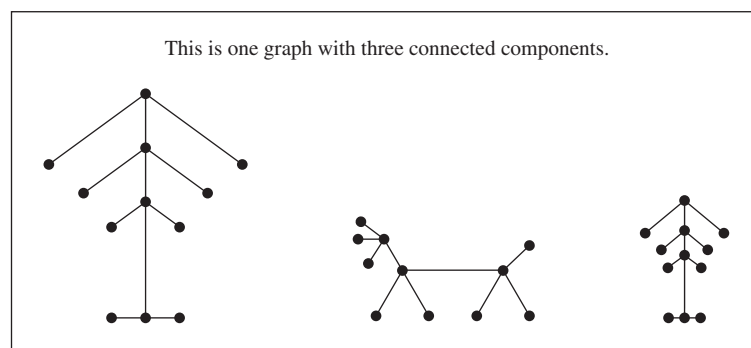
FIGURE 3   Example of a Forest.

Proof: First assume that T is a tree. Then T is a connected graph with no simple circuits. Let x and y be two vertices of T. Because T is connected, by Theorem 1 of Section 10.4 there is a simple path between x and y. Moreover, this path must be unique, for if there were a second such path, the path formed by combining the first path from x to y followed by the path from y to x obtained by reversing the order of the second path from x to y would form a circuit. This implies, using Exercise 59 of Section 10.4, that there is a simple circuit in T. Hence, there is a unique simple path between any two vertices of a tree.

Now assume that there is a unique simple path between any two vertices of a graph T. Then T is connected, because there is a path between any two of its vertices. Furthermore, T can have no simple circuits. To see that this is true, suppose T had a simple circuit that contained the vertices x and y. Then there would be two simple paths between x and y, because the simple circuit is made up of a simple path from x to y and a second simple path from y to x. Hence, a graph with a unique simple path between any two vertices is a tree.   ◁
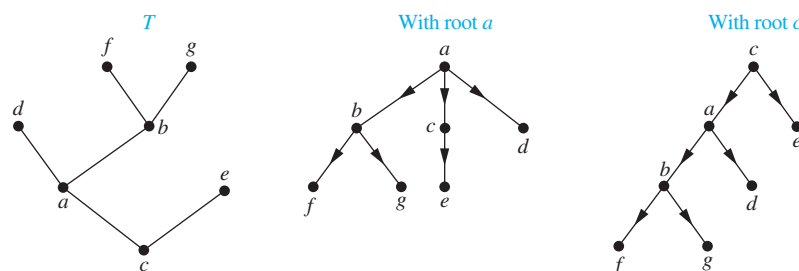
Rooted Trees

In many applications of trees, a particular vertex of a tree is designated as the **root**. Once we specify a root, we can assign a direction to each edge as follows. Because there is a unique path from the root to each vertex of the graph (by Theorem 1), we direct each edge away from the root. Thus, a tree together with its root produces a directed graph called a **rooted tree**.

DEFINITION 2

A *rooted tree* is a tree in which one vertex has been designated as the root and every edge is directed away from the root.

Rooted trees can also be defined recursively. Refer to Section 5.3 to see how this can be done. We can change an unrooted tree into a rooted tree by choosing any vertex as the root. Note that different choices of the root produce different rooted trees. For instance, Figure 4 displays the rooted trees formed by designating a to be the root and c to be the root, respectively, in the tree T. We usually draw a rooted tree with its root at the top of the graph. The arrows indicating the directions of the edges in a rooted tree can be omitted, because the choice of root determines the directions of the edges.

The terminology for trees has botanical and genealogical origins. Suppose that T is a rooted tree. If v is a vertex in T other than the root, the **parent** of v is the unique vertex u such that there is a directed edge from u to v (the reader should show that such a vertex is unique). When u is the parent of v, v is called a **child** of u. Vertices with the same parent are called **siblings**. The **ancestors** of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root (that is, its parent, its parent's parent, and so on, until the root is reached). The **descendants** of a vertex v are those vertices that have v as

FIGURE 4   A Tree and Rooted Trees Formed by Designating Two Different Roots.
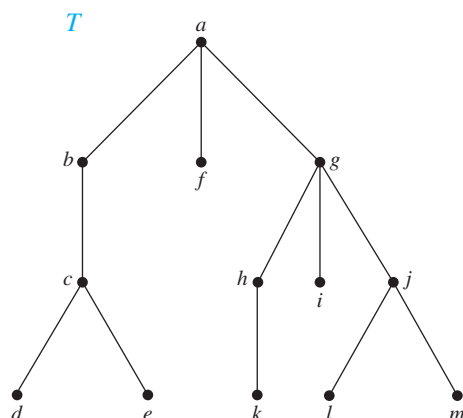
FIGURE 5 **A Rooted Tree T.**
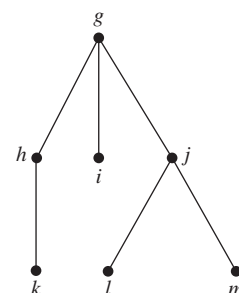
FIGURE 6 **The Subtree Rooted at g.**

an ancestor. A vertex of a rooted tree is called a **leaf** if it has no children. Vertices that have children are called **internal vertices**. The root is an internal vertex unless it is the only vertex in the graph, in which case it is a leaf.

If a is a vertex in a tree, the **subtree** with a as its root is the subgraph of the tree consisting of a and its descendants and all edges incident to these descendants.

EXAMPLE 2 In the rooted tree T (with root a) shown in Figure 5, find the parent of c, the children of g, the siblings of h, all ancestors of e, all descendants of b, all internal vertices, and all leaves. What is the subtree rooted at g?

Solution: The parent of c is b. The children of g are h, i, and j. The siblings of h are i and j. The ancestors of e are c, b, and a. The descendants of b are c, d, and e. The internal vertices are a, b, c, g, h, and j. The leaves are d, e, f, i, k, l, and m. The subtree rooted at g is shown in Figure 6. ◀

Rooted trees with the property that all of their internal vertices have the same number of children are used in many different applications. Later in this chapter we will use such trees to study problems involving searching, sorting, and coding.

DEFINITION 3 A rooted tree is called an *m-ary tree* if every internal vertex has no more than m children. The tree is called a *full m-ary tree* if every internal vertex has exactly m children. An m-ary tree with $m = 2$ is called a *binary tree*.

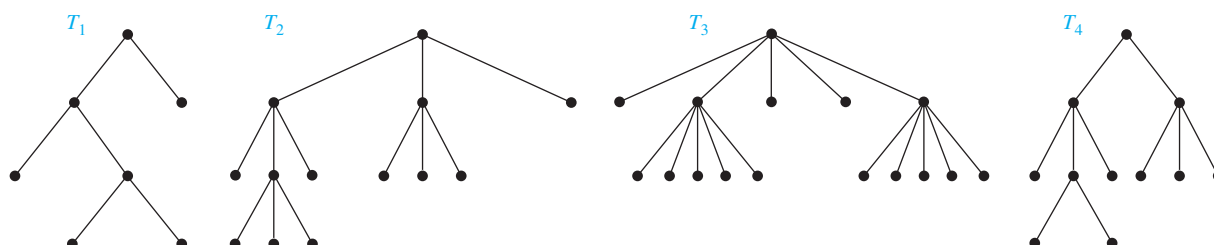**EXAMPLE 3** Are the rooted trees in Figure 7 full m-ary trees for some positive integer m?

FIGURE 7 **Four Rooted Trees.**

Solution: T_1 is a full binary tree because each of its internal vertices has two children. T_2 is a full 3-ary tree because each of its internal vertices has three children. In T_3 each internal vertex has five children, so T_3 is a full 5-ary tree. T_4 is not a full m-ary tree for any m because some of its internal vertices have two children and others have three children.   ◄
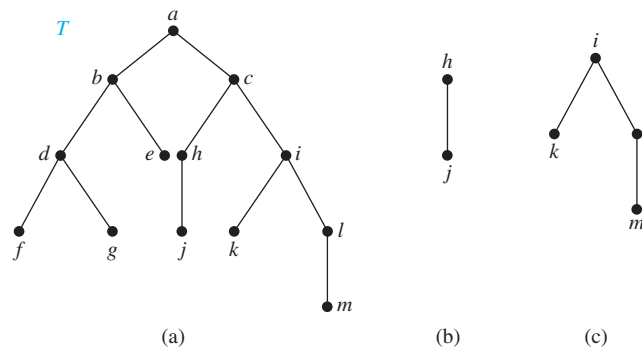
ORDERED ROOTED TREES   An **ordered rooted tree** is a rooted tree where the children of each internal vertex are ordered. Ordered rooted trees are drawn so that the children of each internal vertex are shown in order from left to right. Note that a representation of a rooted tree in the conventional way determines an ordering for its edges. We will use such orderings of edges in drawings without explicitly mentioning that we are considering a rooted tree to be ordered.

In an ordered binary tree (usually called just a **binary tree**), if an internal vertex has two children, the first child is called the **left child** and the second child is called the **right child**. The tree rooted at the left child of a vertex is called the **left subtree** of this vertex, and the tree rooted at the right child of a vertex is called the **right subtree** of the vertex. The reader should note that for some applications every vertex of a binary tree, other than the root, is designated as a right or a left child of its parent. This is done even when some vertices have only one child. We will make such designations whenever it is necessary, but not otherwise.

Ordered rooted trees can be defined recursively. Binary trees, a type of ordered rooted trees, were defined this way in Section 5.3.

EXAMPLE 4   What are the left and right children of d in the binary tree T shown in Figure 8(a) (where the order is that implied by the drawing)? What are the left and right subtrees of c?

Solution: The left child of d is f and the right child is g. We show the left and right subtrees of c in Figures 8(b) and 8(c), respectively.   ◄


FIGURE 8   **A Binary Tree T and Left and Right Subtrees of the Vertex c.**

Just as in the case of graphs, there is no standard terminology used to describe trees, rooted trees, ordered rooted trees, and binary trees. This nonstandard terminology occurs because trees are used extensively throughout computer science, which is a relatively young field. The reader should carefully check meanings given to terms dealing with trees whenever they occur.

Trees as Models

Trees are used as models in such diverse areas as computer science, chemistry, geology, botany, and psychology. We will describe a variety of such models based on trees.

Butane   Isobutane
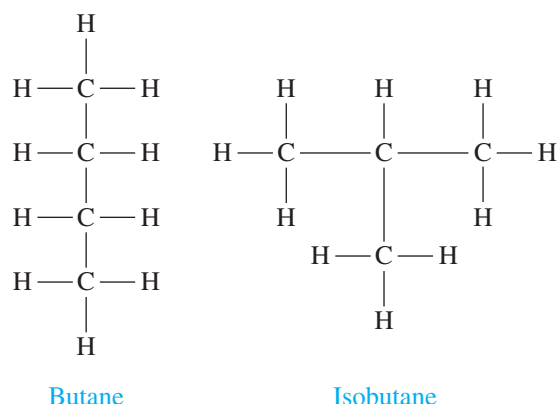
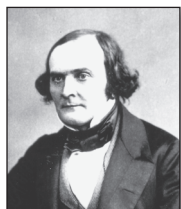**FIGURE 9**   **The Two Isomers of Butane.**

EXAMPLE 5   **Saturated Hydrocarbons and Trees**   Graphs can be used to represent molecules, where atoms are represented by vertices and bonds between them by edges. The English mathematician Arthur Cayley discovered trees in 1857 when he was trying to enumerate the isomers of compounds of the form C_nH_{2n+2}, which are called *saturated hydrocarbons*.

In graph models of saturated hydrocarbons, each carbon atom is represented by a vertex of degree 4, and each hydrogen atom is represented by a vertex of degree 1. There are $3n + 2$ vertices in a graph representing a compound of the form C_nH_{2n+2}. The number of edges in such a graph is half the sum of the degrees of the vertices. Hence, there are $(4n + 2n + 2)/2 = 3n + 1$ edges in this graph. Because the graph is connected and the number of edges is one less than the number of vertices, it must be a tree (see Exercise 15).
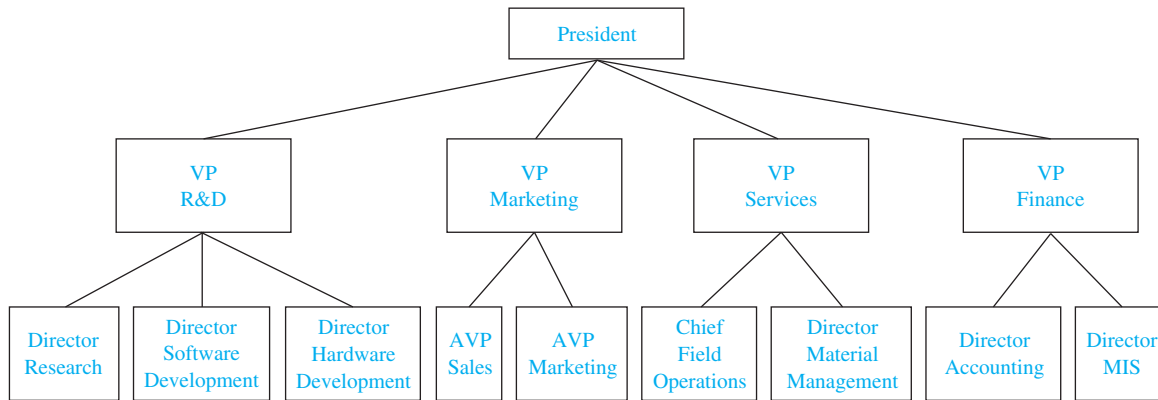
The nonisomorphic trees with n vertices of degree 4 and $2n + 2$ of degree 1 represent the different isomers of C_nH_{2n+2}. For instance, when $n = 4$, there are exactly two nonisomorphic trees of this type (the reader should verify this). Hence, there are exactly two different isomers of C_4H_{10}. Their structures are displayed in Figure 9. These two isomers are called butane and isobutane.   ◀

EXAMPLE 6   **Representing Organizations**   The structure of a large organization can be modeled using a rooted tree. Each vertex in this tree represents a position in the organization. An edge from one vertex to another indicates that the person represented by the initial vertex is the (direct) boss of the person represented by the terminal vertex. The graph shown in Figure 10 displays such a tree. In the organization represented by this tree, the Director of Hardware Development works directly for the Vice President of R&D.   ◀

EXAMPLE 7   **Computer File Systems**   Files in computer memory can be organized into directories. A directory can contain both files and subdirectories. The root directory contains the entire file

Links

FIGURE 10 **An Organizational Tree for a Computer Company.**

system. Thus, a file system may be represented by a rooted tree, where the root represents the root directory, internal vertices represent subdirectories, and leaves represent ordinary files or empty directories. One such file system is shown in Figure 11. In this system, the file khr is in the directory rje. (Note that links to files where the same file may have more than one pathname can lead to circuits in computer file systems.) ◀

The root is the root directory /
Internal vertices are directories
Leaves are files
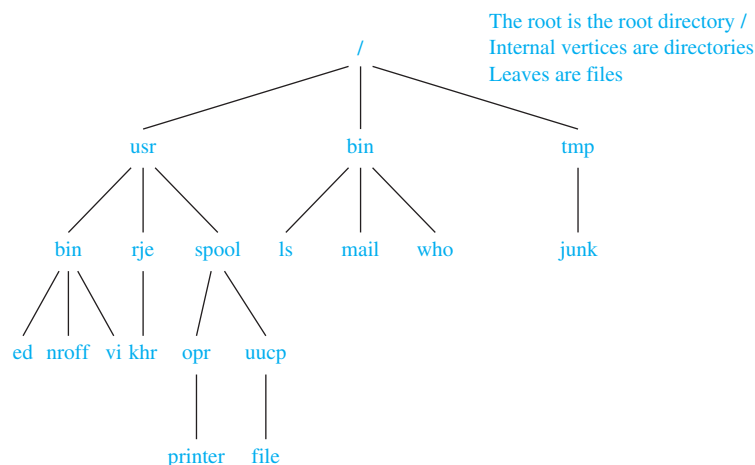
FIGURE 11 **A Computer File System.**

EXAMPLE 8 **Tree-Connected Parallel Processors** In Example 17 of Section 10.2 we described several interconnection networks for parallel processing. A **tree-connected network** is another important way to interconnect processors. The graph representing such a network is a complete binary tree, that is, a full binary tree where every root is at the same level. Such a network interconnects $n = 2^k - 1$ processors, where k is a positive integer. A processor represented by the vertex v that is not a root or a leaf has three two-way connections—one to the processor represented by the parent of v and two to the processors represented by the two children of v. The processor represented by the root has two two-way connections to the processors represented by its two children. A processor represented by a leaf v has a single two-way connection to the parent of v. We display a tree-connected network with seven processors in Figure 12.
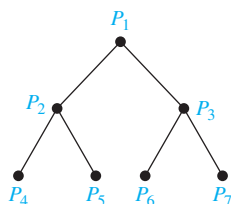
FIGURE 12 **A Tree-Connected Network of Seven Processors.**

We now illustrate how a tree-connected network can be used for parallel computation. In particular, we show how the processors in Figure 12 can be used to add eight numbers, using three steps. In the first step, we add x_1 and x_2 using P_4, x_3 and x_4 using P_5, x_5 and x_6 using P_6,

and x_7 and x_8 using P_7. In the second step, we add $x_1 + x_2$ and $x_3 + x_4$ using P_2 and $x_5 + x_6$ and $x_7 + x_8$ using P_3. Finally, in the third step, we add $x_1 + x_2 + x_3 + x_4$ and $x_5 + x_6 + x_7 + x_8$ using P_1. The three steps used to add eight numbers compares favorably to the seven steps required to add eight numbers serially, where the steps are the addition of one number to the sum of the previous numbers in the list.   ◀

Properties of Trees

We will often need results relating the numbers of edges and vertices of various types in trees.

THEOREM 2   A tree with n vertices has $n - 1$ edges.

Links

Proof: We will use mathematical induction to prove this theorem. Note that for all the trees here we can choose a root and consider the tree rooted.

BASIS STEP: When $n = 1$, a tree with $n = 1$ vertex has no edges. It follows that the theorem is true for $n = 1$.

INDUCTIVE STEP: The inductive hypothesis states that every tree with k vertices has $k - 1$ edges, where k is a positive integer. Suppose that a tree T has $k + 1$ vertices and that v is a leaf of T (which must exist because the tree is finite), and let w be the parent of v. Removing from T the vertex v and the edge connecting w to v produces a tree T' with k vertices, because the resulting graph is still connected and has no simple circuits. By the inductive hypothesis, T' has $k - 1$ edges. It follows that T has k edges because it has one more edge than T', the edge connecting v and w. This completes the inductive step.   ◀

Recall that a tree is a connected undirected graph with no simple circuits. So, when G is an undirected graph with n vertices, Theorem 2 tells us that the two conditions *(i)* G is connected and *(ii)* G has no simple circuits, imply *(iii)* G has $n - 1$ edges. Also, when *(i)* and *(iii)* hold, then *(ii)* must also hold, and when *(ii)* and *(iii)* hold, *(i)* must also hold. That is, if G is connected and G has $n - 1$ edges, then G has no simple circuits, so that G is a tree (see Exercise 15(a)), and if G has no simple circuits and G has $n - 1$ edges, then G is connected, and so is a tree (see Exercise 15(b)). Consequently, when two of *(i)*, *(ii)*, and *(iii)* hold, the third condition must also hold, and G must be a tree.

COUNTING VERTICES IN FULL m-ARY TREES   The number of vertices in a full m-ary tree with a specified number of internal vertices is determined, as Theorem 3 shows. As in Theorem 2, we will use n to denote the number of vertices in a tree.

THEOREM 3   A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices.

Proof: Every vertex, except the root, is the child of an internal vertex. Because each of the i internal vertices has m children, there are mi vertices in the tree other than the root. Therefore, the tree contains $n = mi + 1$ vertices.   ◀

Suppose that T is a full m-ary tree. Let i be the number of internal vertices and l the number of leaves in this tree. Once one of n, i, and l is known, the other two quantities are determined. Theorem 4 explains how to find the other two quantities from the one that is known.

THEOREM 4 A full m-ary tree with

  (*i*) n vertices has $i = (n-1)/m$ internal vertices and $l = [(m-1)n+1]/m$ leaves,
  (*ii*) i internal vertices has $n = mi + 1$ vertices and $l = (m-1)i + 1$ leaves,
 (*iii*) l leaves has $n = (ml - 1)/(m-1)$ vertices and $i = (l-1)/(m-1)$ internal vertices.

Proof: Let n represent the number of vertices, i the number of internal vertices, and l the number of leaves. The three parts of the theorem can all be proved using the equality given in Theorem 3, that is, $n = mi + 1$, together with the equality $n = l + i$, which is true because each vertex is either a leaf or an internal vertex. We will prove part (*i*) here. The proofs of parts (*ii*) and (*iii*) are left as exercises for the reader.

  Solving for i in $n = mi + 1$ gives $i = (n-1)/m$. Then inserting this expression for i into the equation $n = l + i$ shows that $l = n - i = n - (n-1)/m = [(m-1)n+1]/m$. ◁

  Example 9 illustrates how Theorem 4 can be used.

EXAMPLE 9 Suppose that someone starts a chain letter. Each person who receives the letter is asked to send it on to four other people. Some people do this, but others do not send any letters. How many people have seen the letter, including the first person, if no one receives more than one letter and if the chain letter ends after there have been 100 people who read it but did not send it out? How many people sent out the letter?

FIGURE 13 **A Rooted Tree.**
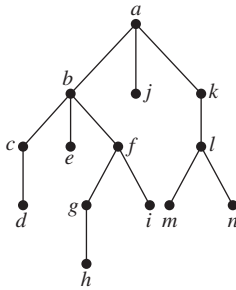
Solution: The chain letter can be represented using a 4-ary tree. The internal vertices correspond to people who sent out the letter, and the leaves correspond to people who did not send it out. Because 100 people did not send out the letter, the number of leaves in this rooted tree is $l = 100$. Hence, part (*iii*) of Theorem 4 shows that the number of people who have seen the letter is $n = (4 \cdot 100 - 1)/(4 - 1) = 133$. Also, the number of internal vertices is $133 - 100 = 33$, so 33 people sent out the letter. ◀

BALANCED m-ARY TREES It is often desirable to use rooted trees that are "balanced" so that the subtrees at each vertex contain paths of approximately the same length. Some definitions will make this concept clear. The **level** of a vertex v in a rooted tree is the length of the unique path from the root to this vertex. The level of the root is defined to be zero. The **height** of a rooted tree is the maximum of the levels of vertices. In other words, the height of a rooted tree is the length of the longest path from the root to any vertex.

EXAMPLE 10 Find the level of each vertex in the rooted tree shown in Figure 13. What is the height of this tree?

Solution: The root a is at level 0. Vertices b, j, and k are at level 1. Vertices c, e, f, and l are at level 2. Vertices d, g, i, m, and n are at level 3. Finally, vertex h is at level 4. Because the largest level of any vertex is 4, this tree has height 4. ◀

  A rooted m-ary tree of height h is **balanced** if all leaves are at levels h or $h - 1$.

EXAMPLE 11 Which of the rooted trees shown in Figure 14 are balanced?

Solution: T_1 is balanced, because all its leaves are at levels 3 and 4. However, T_2 is not balanced, because it has leaves at levels 2, 3, and 4. Finally, T_3 is balanced, because all its leaves are at level 3. ◀
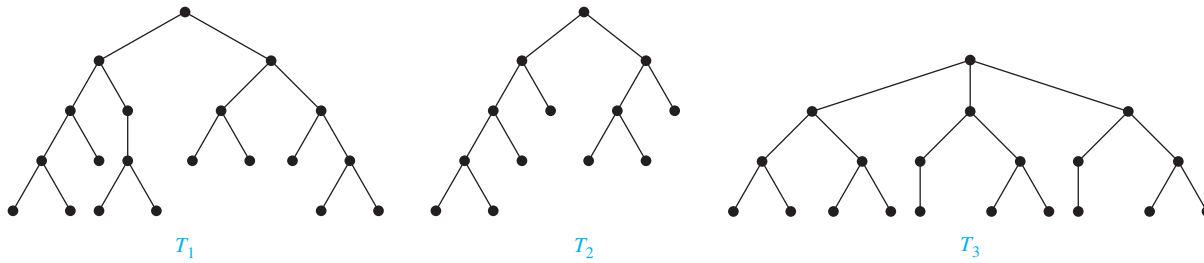
FIGURE 14   **Some Rooted Trees.**

A BOUND FOR THE NUMBER OF LEAVES IN AN m-ARY TREE   It is often useful to have an upper bound for the number of leaves in an m-ary tree. Theorem 5 provides such a bound in terms of the height of the m-ary tree.

THEOREM 5   There are at most m^h leaves in an m-ary tree of height h.

Proof: The proof uses mathematical induction on the height. First, consider m-ary trees of height 1. These trees consist of a root with no more than m children, each of which is a leaf. Hence, there are no more than $m^1 = m$ leaves in an m-ary tree of height 1. This is the basis step of the inductive argument.

Now assume that the result is true for all m-ary trees of height less than h; this is the inductive hypothesis. Let T be an m-ary tree of height h. The leaves of T are the leaves of the subtrees of T obtained by deleting the edges from the root to each of the vertices at level 1, as shown in Figure 15.

Each of these subtrees has height less than or equal to $h - 1$. So by the inductive hypothesis, each of these rooted trees has at most m^{h-1} leaves. Because there are at most m such subtrees, each with a maximum of m^{h-1} leaves, there are at most $m \cdot m^{h-1} = m^h$ leaves in the rooted tree. This finishes the inductive argument. ◁

COROLLARY 1   If an m-ary tree of height h has l leaves, then $h \geq \lceil \log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil \log_m l \rceil$. (We are using the ceiling function here. Recall that $\lceil x \rceil$ is the smallest integer greater than or equal to x.)

Proof: We know that $l \leq m^h$ from Theorem 5. Taking logarithms to the base m shows that $\log_m l \leq h$. Because h is an integer, we have $h \geq \lceil \log_m l \rceil$. Now suppose that the tree is balanced.
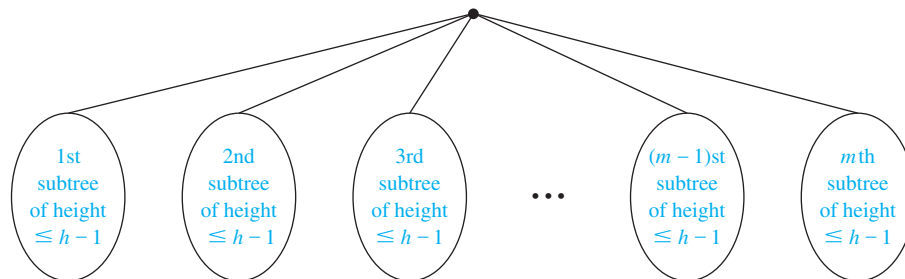
FIGURE 15   **The Inductive Step of the Proof.**

Then each leaf is at level h or $h - 1$, and because the height is h, there is at least one leaf at level h. It follows that there must be more than $m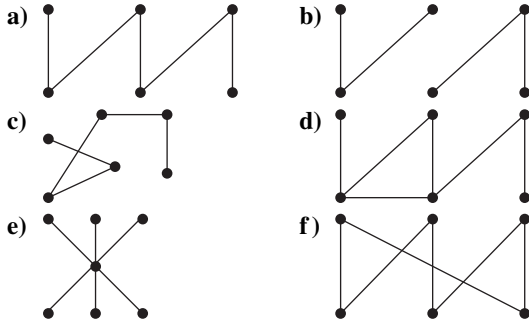^{h-1}$ leaves (see Exercise 30). Because $l \leq m^h$, we have $m^{h-1} < l \leq m^h$. Taking logarithms to the base m in this inequality gives $h - 1 < \log_m l \leq h$. Hence, $h = \lceil \log_m l \rceil$.   ◁

Exercises

1. Which of these graphs are trees?

2. Which of these graphs are trees?
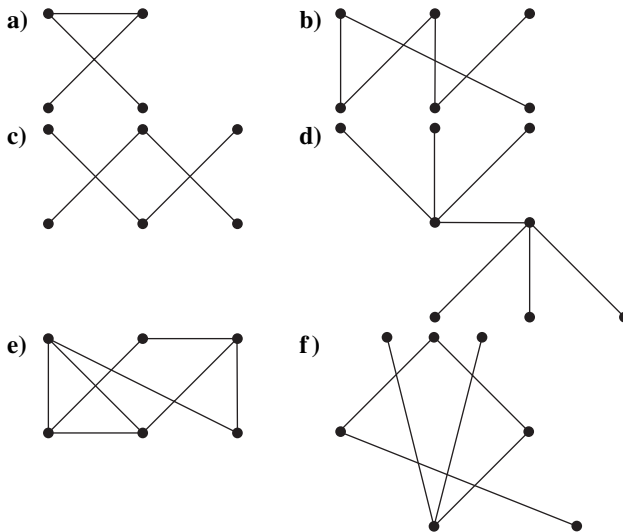
3. Answer these questions about the rooted tree illustrated.
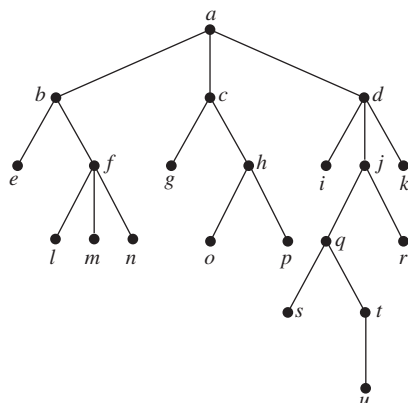
a) Which vertex is the root?
b) Which vertices are internal?
c) Which vertices are leaves?
d) Which vertices are children of j?
e) Which vertex is the parent of h?
f) Which vertices are siblings of o?
g) Which vertices are ancestors of m?
h) Which vertices are descendants of b?

4. Answer the same questions as listed in Exercise 3 for the rooted tree illustrated.
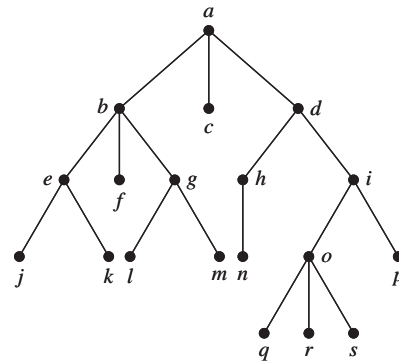
5. Is the rooted tree in Exercise 3 a full m-ary tree for some positive integer m?

6. Is the rooted tree in Exercise 4 a full m-ary tree for some positive integer m?

7. What is the level of each vertex of the rooted tree in Exercise 3?

8. What is the level of each vertex of the rooted tree in Exercise 4?

9. Draw the subtree of the tree in Exercise 3 that is rooted at

    a) a.   b) c.   c) e.

10. Draw the subtree of the tree in Exercise 4 that is rooted at

    a) a.   b) c.   c) e.

11. a) How many nonisomorphic unrooted trees are there with three vertices?

    b) How many nonisomorphic rooted trees are there with three vertices (using isomorphism for directed graphs)?

*12. a) How many nonisomorphic unrooted trees are there with four vertices?

    b) How many nonisomorphic rooted trees are there with four vertices (using isomorphism for directed graphs)?

*13. a) How many nonisomorphic unrooted trees are there with five vertices?

b) How many nonisomorphic rooted trees are there with five vertices (using isomorphism for directed graphs)?

*14. Show that a simple graph is a tree if and only if it is connected but the deletion of any of its edges produces a graph that is not connected.

☞ ***15.** Let G be a simple graph with n vertices. Show that

a) G is a tree if and only if it is connected and has $n - 1$ edges.

b) G is a tree if and only if G has no simple circuits and has $n - 1$ edges. [*Hint:* To show that G is connected if it has no simple circuits and $n - 1$ edges, show that G cannot have more than one connected component.]

16. Which complete bipartite graphs $K_{m,n}$, where m and n are positive integers, are trees?

17. How many edges does a tree with 10,000 vertices have?

18. How many vertices does a full 5-ary tree with 100 internal vertices have?

19. How many edges does a full binary tree with 1000 internal vertices have?

20. How many leaves does a full 3-ary tree with 100 vertices have?

21. Suppose 1000 people enter a chess tournament. Use a rooted tree model of the tournament to determine how many games must be played to determine a champion, if a player is eliminated after one loss and games are played until only one entrant has not lost. (Assume there are no ties.)

22. A chain letter starts when a person sends a letter to five others. Each person who receives the letter either sends it to five other people who have never received it or does not send it to anyone. Suppose that 10,000 people send out the letter before the chain ends and that no one receives more than one letter. How many people receive the letter, and how many do not send it out?

23. A chain letter starts with a person sending a letter out to 10 others. Each person is asked to send the letter out to 10 others, and each letter contains a list of the previous six people in the chain. Unless there are fewer than six names in the list, each person sends one dollar to the first person in this list, removes the name of this person from the list, moves up each of the other five names one position, and inserts his or her name at the end of this list. If no person breaks the chain and no one receives more than one letter, how much money will a person in the chain ultimately receive?

*24. Either draw a full m-ary tree with 76 leaves and height 3, where m is a positive integer, or show that no such tree exists.

*25. Either draw a full m-ary tree with 84 leaves and height 3, where m is a positive integer, or show that no such tree exists.

*26. A full m-ary tree T has 81 leaves and height 4.

a) Give the upper and lower bounds for m.

b) What is m if T is also balanced?

A **complete m-ary tree** is a full m-ary tree in which every leaf is at the same level.

27. Construct a complete binary tree of height 4 and a complete 3-ary tree of height 3.

28. How many vertices and how many leaves does a complete m-ary tree of height h have?

29. Prove

a) part (*ii*) of Theorem 4.

b) part (*iii*) of Theorem 4.

☞ **30.** Show that a full m-ary balanced tree of height h has more than m^{h-1} leaves.

31. How many edges are there in a forest of t trees containing a total of n vertices?

32. Explain how a tree can be used to represent the table of contents of a book organized into chapters, where each chapter is organized into sections, and each section is organized into subsections.

33. How many different isomers do these saturated hydrocarbons have?

a) C_3H_8 **b)** C_5H_{12} **c)** C_6H_{14}

34. What does each of these represent in an organizational tree?

a) the parent of a vertex

b) a child of a vertex

c) a sibling of a vertex

d) the ancestors of a vertex

e) the descendants of a vertex

f) the level of a vertex

g) the height of the tree

35. Answer the same questions as those given in Exercise 34 for a rooted tree representing a computer file system.

36. a) Draw the complete binary tree with 15 vertices that represents a tree-connected network of 15 processors.

b) Show how 16 numbers can be added using the 15 processors in part (a) using four steps.

37. Let n be a power of 2. Show that n numbers can be added in $\log n$ steps using a tree-connected network of $n - 1$ processors.

*38. A **labeled tree** is a tree where each vertex is assigned a label. Two labeled trees are considered isomorphic when there is an isomorphism between them that preserves the labels of vertices. How many nonisomorphic trees are there with three vertices labeled with different integers from the set $\{1, 2, 3\}$? How many nonisomorphic trees are there with four vertices labeled with different integers from the set $\{1, 2, 3, 4\}$?

The **eccentricity** of a vertex in an unrooted tree is the length of the longest simple path beginning at this vertex. A vertex is called a **center** if no vertex in the tree has smaller eccentricity than this vertex. In Exercises 39–41 find every vertex that is a center in the given tree.
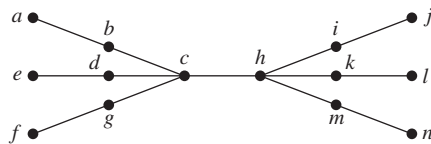
39.
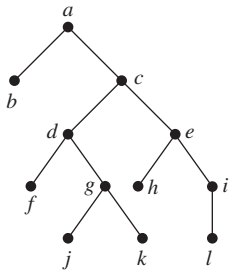
40.
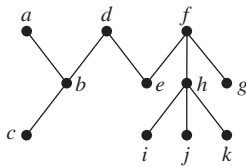
41.

42. Show that a center should be chosen as the root to produce a rooted tree of minimal height from an unrooted tree.

*43. Show that a tree has either one center or two centers that are adjacent.

44. Show that every tree can be colored using two colors.

The **rooted Fibonacci trees** T_n are defined recursively in the following way. T_1 and T_2 are both the rooted tree consisting of a single vertex, and for $n = 3, 4, \ldots$, the rooted tree T_n is constructed from a root with T_{n-1} as its left subtree and T_{n-2} as its right subtree.

45. Draw the first seven rooted Fibonacci trees.

*46. How many vertices, leaves, and internal vertices does the rooted Fibonacci tree T_n have, where n is a positive integer? What is its height?

47. What is wrong with the following "proof" using mathematical induction of the statement that every tree with n vertices has a path of length $n - 1$. *Basis step:* Every tree with one vertex clearly has a path of length 0. *Inductive step:* Assume that a tree with n vertices has a path of length $n - 1$, which has u as its terminal vertex. Add a vertex v and the edge from u to v. The resulting tree has $n + 1$ vertices and has a path of length n. This completes the inductive step.

*48. Show that the average depth of a leaf in a binary tree with n vertices is $\Omega(\log n)$.

11.2 Applications of Trees

Introduction

We will discuss three problems that can be studied using trees. The first problem is: How should items in a list be stored so that an item can be easily located? The second problem is: What series of decisions should be made to find an object with a certain property in a collection of objects of a certain type? The third problem is: How should a set of characters be efficiently coded by bit strings?

Binary Search Trees

Links

Searching for items in a list is one of the most important tasks that arises in computer science. Our primary goal is to implement a searching algorithm that finds items efficiently when the items are totally ordered. This can be accomplished through the use of a **binary search tree**, which is a binary tree in which each child of a vertex is designated as a right or left child, no vertex has more than one right child or left child, and each vertex is labeled with a key, which is one of the items. Furthermore, vertices are assigned keys so that the key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.

This recursive procedure is used to form the binary search tree for a list of items. Start with a tree containing just one vertex, namely, the root. The first item in the list is assigned as the key of the root. To add a new item, first compare it with the keys of vertices already in the tree, starting at the root and moving to the left if the item is less than the key of the respective vertex if this vertex has a left child, or moving to the right if the item is greater than the key of the

respective vertex if this vertex has a right child. When the item is less than the respective vertex and this vertex has no left child, then a new vertex with this item as its key is inserted as a new left child. Similarly, when the item is greater than the respective vertex and this vertex has no right child, then a new vertex with this item as its key is inserted as a new right child. We illustrate this procedure with Example 1.

EXAMPLE 1 Form a binary search tree for the words *mathematics, physics, geography, zoology, meteorology, geology, psychology*, and *chemistry* (using alphabetical order).

Solution: Figure 1 displays the steps used to construct this binary search tree. The word *mathematics* is the key of the root. Because *physics* comes after *mathematics* (in alphabetical order), add a right child of the root with key *physics*. Because *geography* comes before *mathematics*, add a left child of the root with key *geography*. Next, add a right child of the vertex with key *physics*, and assign it the key *zoology*, because *zoology* comes after *mathematics* and after *physics*. Similarly, add a left child of the vertex with key *physics* and assign this new vertex the key *meteorology*. Add a right child of the vertex with key *geography* and assign this new vertex the key *geology*. Add a left child of the vertex with key *zoology* and assign it the key *psychology*. Add a left child of the vertex with key *geography* and assign it the key *chemistry*. (The reader should work through all the comparisons needed at each step.) ◀

Once we have a binary search tree, we need a way to locate items in the binary search tree, as well as a way to add new items. Algorithm 1, an insertion algorithm, actually does both of these tasks, even though it may appear that it is only designed to add vertices to a binary search tree. That is, Algorithm 1 is a procedure that locates an item x in a binary search tree if it is present, and adds a new vertex with x as its key if x is not present. In the pseudocode, v is the vertex currently under examination and $label(v)$ represents the key of this vertex. The algorithm begins by examining the root. If x equals the key of v, then the algorithm has found the location of x and terminates; if x is less than the key of v, we move to the left child of v and repeat the procedure; and if x is greater than the key of v, we move to the right child of v and repeat the procedure. If at any step we attempt to move to a child that is not present, we know that x is not present in the tree, and we add a new vertex as this child with x as its key.
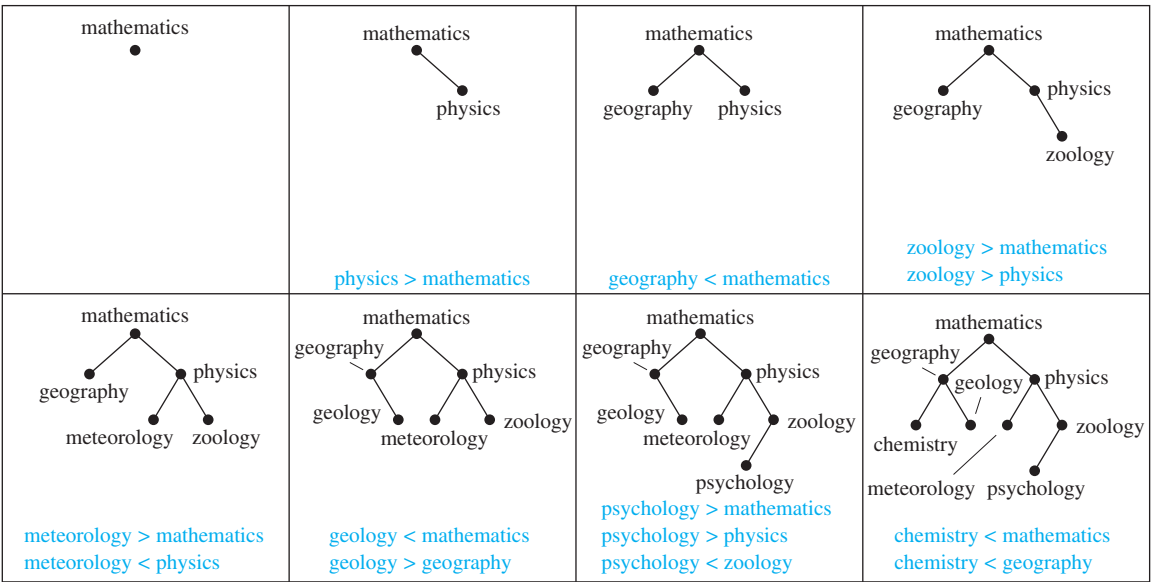
FIGURE 1 Constructing a Binary Search Tree.

ALGORITHM 1 Locating an Item in or Adding an Item to a Binary Search Tree.

procedure *insertion*(*T*: binary search tree, *x*: item)
v := root of *T*
{a vertex not present in *T* has the value *null* }
while *v* ≠ *null* and *label*(*v*) ≠ *x*
  if *x* < *label*(*v*) **then**
    if left child of *v* ≠ *null* **then** *v* := left child of *v*
    else add *new vertex* as a left child of *v* and set *v* := *null*
  else
    if right child of *v* ≠ *null* **then** *v* := right child of *v*
    else add *new vertex* as a right child of *v* and set *v* := *null*
if root of *T* = *null* **then** add a vertex *v* to the tree and label it with *x*
else if *v* is null or *label*(*v*) ≠ *x* **then** label *new vertex* with *x* and let *v* be this new vertex
return *v* {*v* = location of *x*}

Example 2 illustrates the use of Algorithm 1 to insert a new item into a binary search tree.

EXAMPLE 2 Use Algorithm 1 to insert the word *oceanography* into the binary search tree in Example 1.

Solution: Algorithm 1 begins with *v*, the vertex under examination, equal to the root of *T*, so *label*(*v*) = *mathematics*. Because *v* ≠ *null* and *label*(*v*) = *mathematics* < *oceanography*, we next examine the right child of the root. This right child exists, so we set *v*, the vertex under examination, to be this right child. At this step we have *v* ≠ *null* and *label*(*v*) = *physics* > *oceanography*, so we examine the left child of *v*. This left child exists, so we set *v*, the vertex under examination, to this left child. At this step, we also have *v* ≠ *null* and *label*(*v*) = *metereology* < *oceanography*, so we try to examine the right child of *v*. However, this right child does not exist, so we add a new vertex as the right child of *v* (which at this point is the vertex with the key *metereology*) and we set *v* := *null*. We now exit the **while** loop because *v* = *null*. Because the root of *T* is not *null* and *v* = *null*, we use the **else if** statement at the end of the algorithm to label our new vertex with the key *oceanography*. ◄

We will now determine the computational complexity of this procedure. Suppose we have a binary search tree *T* for a list of *n* items. We can form a full binary tree *U* from *T* by adding unlabeled vertices whenever necessary so that every vertex with a key has two children. This is illustrated in Figure 2. Once we have done this, we can easily locate or add a new item as a key without adding a vertex.

The most comparisons needed to add a new item is the length of the longest path in *U* from the root to a leaf. The internal vertices of *U* are the vertices of *T*. It follows that *U* has *n* internal vertices. We can now use part (*ii*) of Theorem 4 in Section 11.1 to conclude that *U* has *n* + 1 leaves. Using Corollary 1 of Section 11.1, we see that the height of *U* is greater than or equal to $h = \lceil \log(n + 1) \rceil$. Consequently, it is necessary to perform at least $\lceil \log(n + 1) \rceil$ comparisons to add some item. Note that if *U* is balanced, its height is $\lceil \log(n + 1) \rceil$ (by Corollary 1 of Section 11.1). Thus, if a binary search tree is balanced, locating or adding an item requires no more than $\lceil \log(n + 1) \rceil$ comparisons. A binary search tree can become unbalanced as items are added to it. Because balanced binary search trees give optimal worst-case complexity for binary searching, algorithms have been devised that rebalance binary search trees as items are added. The interested reader can consult references on data structures for the description of such algorithms.

Unlabeled vertices circled
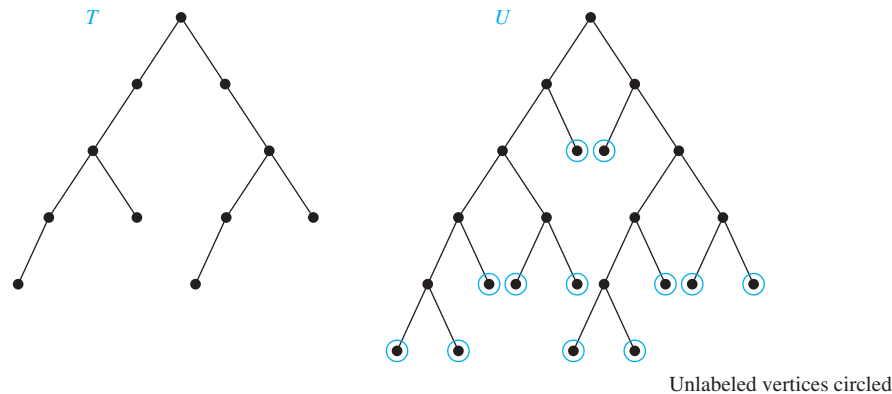
FIGURE 2 **Adding Unlabeled Vertices to Make a Binary Search Tree Full.**

Decision Trees

Links

Rooted trees can be used to model problems in which a series of decisions leads to a solution. For instance, a binary search tree can be used to locate items based on a series of comparisons, where each comparison tells us whether we have located the item, or whether we should go right or left in a subtree. A rooted tree in which each internal vertex corresponds to a decision, with a subtree at these vertices for each possible outcome of the decision, is called a **decision tree**. The possible solutions of the problem correspond to the paths to the leaves of this rooted tree. Example 3 illustrates an application of decision trees.

EXAMPLE 3   Suppose there are seven coins, all with the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which of the eight coins is the counterfeit one? Give an algorithm for finding this counterfeit coin.

Extra Examples

Solution: There are three possibilities for each weighing on a balance scale. The two pans can have equal weight, the first pan can be heavier, or the second pan can be heavier. Consequently, the decision tree for the sequence of weighings is a 3-ary tree. There are at least eight leaves in the decision tree because there are eight possible outcomes (because each of the eight coins can be the counterfeit lighter coin), and each possible outcome must be represented by at least one leaf. The largest number of weighings needed to determine the counterfeit coin is the height of the decision tree. From Corollary 1 of Section 11.1 it follows that the height of the decision tree is at least $\lceil \log_3 8 \rceil = 2$. Hence, at least two weighings are needed.

It is possible to determine the counterfeit coin using two weighings. The decision tree that illustrates how this is done is shown in Figure 3.   ◀

THE COMPLEXITY OF COMPARISON-BASED SORTING ALGORITHMS   Many different sorting algorithms have been developed. To decide whether a particular sorting algorithm is efficient, its complexity is determined. Using decision trees as models, a lower bound for the worst-case complexity of sorting algorithms that are based on binary comparisons can be found.

We can use decision trees to model sorting algorithms and to determine an estimate for the worst-case complexity of these algorithms. Note that given n elements, there are $n!$ possible orderings of these elements, because each of the $n!$ permutations of these elements can be the correct order. The sorting algorithms studied in this book, and most commonly used sorting algorithms, are based on binary comparisons, that is, the comparison of two elements at a time. The result of each such comparison narrows down the set of possible orderings. Thus, a sorting algorithm based on binary comparisons can be represented by a binary decision tree in which each internal vertex represents a comparison of two elements. Each leaf represents one of the $n!$ permutations of n elements.
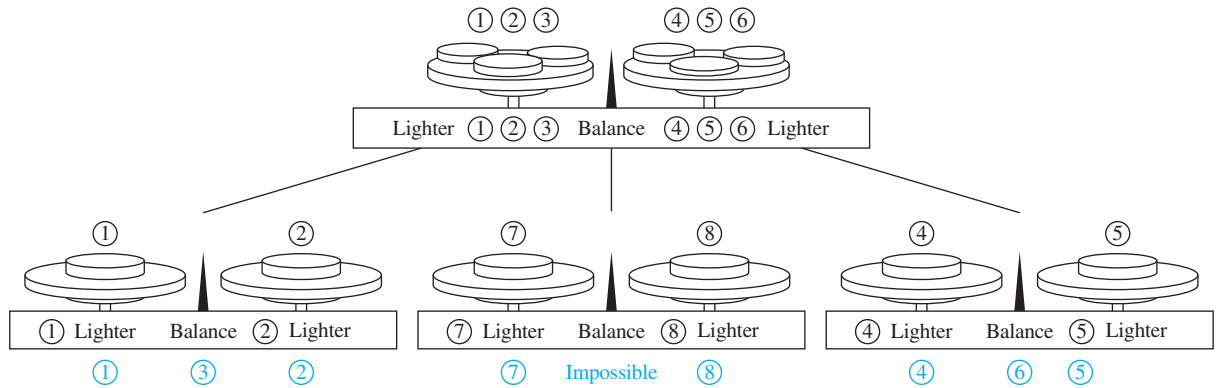
FIGURE 3   **A Decision Tree for Locating a Counterfeit Coin. The counterfeit coin is shown in color below each final weighing.**

EXAMPLE 4   We display in Figure 4 a decision tree that orders the elements of the list a, b, c.   ◀
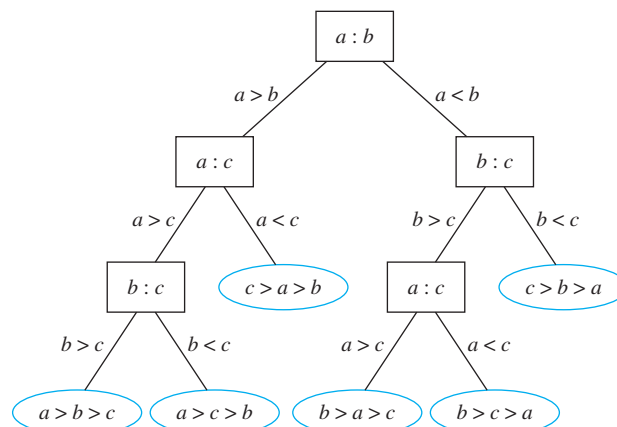
FIGURE 4   **A Decision Tree for Sorting Three Distinct Elements.**

The complexity of a sort based on binary comparisons is measured in terms of the number of such comparisons used. The largest number of binary comparisons ever needed to sort a list with n elements gives the worst-case performance of the algorithm. The most comparisons used equals the longest path length in the decision tree representing the sorting procedure. In other words, the largest number of comparisons ever needed is equal to the height of the decision tree. Because the height of a binary tree with $n!$ leaves is at least $\lceil \log n! \rceil$ (using Corollary 1 in Section 11.1), at least $\lceil \log n! \rceil$ comparisons are needed, as stated in Theorem 1.

THEOREM 1   A sorting algorithm based on binary comparisons requires at least $\lceil \log n! \rceil$ comparisons.

We can use Theorem 1 to provide a big-Omega estimate for the number of comparisons used by a sorting algorithm based on binary comparison. We need only note that by Exercise 72 in Section 3.2 we know that $\lceil \log n! \rceil$ is $\Theta(n \log n)$, one of the commonly used reference functions for the computational complexity of algorithms. Corollary 1 is a consequence of this estimate.

COROLLARY 1 The number of comparisons used by a sorting algorithm to sort n elements based on binary comparisons is $\Omega(n \log n)$.

A consequence of Corollary 1 is that a sorting algorithm based on binary comparisons that uses $\Theta(n \log n)$ comparisons, in the worst case, to sort n elements is optimal, in the sense that no other such algorithm has better worst-case complexity. Note that by Theorem 1 in Section 5.4 we see that the merge sort algorithm is optimal in this sense.

We can also establish a similar result for the average-case complexity of sorting algorithms. The average number of comparisons used by a sorting algorithm based on binary comparisons is the average depth of a leaf in the decision tree representing the sorting algorithm. By Exercise 48 in Section 11.1 we know that the average depth of a leaf in a binary tree with N vertices is $\Omega(\log N)$. We obtain the following estimate when we let $N = n!$ and note that a function that is $\Omega(\log n!)$ is also $\Omega(n \log n)$ because $\log n!$ is $\Theta(n \log n)$.

THEOREM 2 The average number of comparisons used by a sorting algorithm to sort n elements based on binary comparisons is $\Omega(n \log n)$.
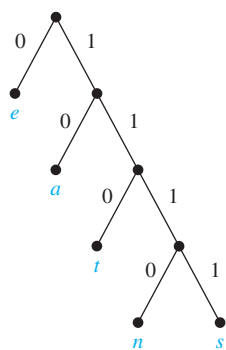
Prefix Codes

Consider the problem of using bit strings to encode the letters of the English alphabet (where no distinction is made between lowercase and uppercase letters). We can represent each letter with a bit string of length five, because there are only 26 letters and there are 32 bit strings of length five. The total number of bits used to encode data is five times the number of characters in the text when each character is encoded with five bits. Is it possible to find a coding scheme of these letters such that, when data are coded, fewer bits are used? We can save memory and reduce transmittal time if this can be done.

Consider using bit strings of different lengths to encode letters. Letters that occur more frequently should be encoded using short bit strings, and longer bit strings should be used to encode rarely occurring letters. When letters are encoded using varying numbers of bits, some method must be used to determine where the bits for each character start and end. For instance, if e were encoded with 0, a with 1, and t with 01, then the bit string 0101 could correspond to *eat, tea, eaea*, or *tt*.

One way to ensure that no bit string corresponds to more than one sequence of letters is to encode letters so that the bit string for a letter never occurs as the first part of the bit string for another letter. Codes with this property are called **prefix codes**. For instance, the encoding of e as 0, a as 10, and t as 11 is a prefix code. A word can be recovered from the unique bit string that encodes its letters. For example, the string 10110 is the encoding of *ate*. To see this, note that the initial 1 does not represent a character, but 10 does represent a (and could not be the first part of the bit string of another letter). Then, the next 1 does not represent a character, but 11 does represent t. The final bit, 0, represents e.

A prefix code can be represented using a binary tree, where the characters are the labels of the leaves in the tree. The edges of the tree are labeled so that an edge leading to a left child is assigned a 0 and an edge leading to a right child is assigned a 1. The bit string used to encode a character is the sequence of labels of the edges in the unique path from the root to the leaf that has this character as its label. For instance, the tree in Figure 5 represents the encoding of e by 0, a by 10, t by 110, n by 1110, and s by 1111.

The tree representing a code can be used to decode a bit string. For instance, consider the word encoded by 11111011100 using the code in Figure 5. This bit string can be decoded by starting at the root, using the sequence of bits to form a path that stops when a leaf is reached.

FIGURE 5 A Binary Tree with a Prefix Code.

Each 0 bit takes the path down the edge leading to the left child of the last vertex in the path, and each 1 bit corresponds to the right child of this vertex. Consequently, the initial 1111 corresponds to the path starting at the root, going right four times, leading to a leaf in the graph that has *s* as its label, because the string 1111 is the code for *s*. Continuing with the fifth bit, we reach a leaf next after going right then left, when the vertex labeled with *a*, which is encoded by 10, is visited. Starting with the seventh bit, we reach a leaf next after going right three times and then left, when the vertex labeled with *n*, which is encoded by 1110, is visited. Finally, the last bit, 0, leads to the leaf that is labeled with *e*. Therefore, the original word is *sane*.

We can construct a prefix code from any binary tree where the left edge at each internal vertex is labeled by 0 and the right edge by a 1 and where the leaves are labeled by characters. Characters are encoded with the bit string constructed using the labels of the edges in the unique path from the root to the leaves.

Links

HUFFMAN CODING   We now introduce an algorithm that takes as input the frequencies (which are the probabilities of occurrences) of symbols in a string and produces as output a prefix code that encodes the string using the fewest possible bits, among all possible binary prefix codes for these symbols. This algorithm, known as **Huffman coding**, was developed by David Huffman in a term paper he wrote in 1951 while a graduate student at MIT. (Note that this algorithm assumes that we already know how many times each symbol occurs in the string, so we can compute the frequency of each symbol by dividing the number of times this symbol occurs by the length of the string.) Huffman coding is a fundamental algorithm in *data compression*, the subject devoted to reducing the number of bits required to represent information. Huffman coding is extensively used to compress bit strings representing text and it also plays an important role in compressing audio and image files.

Algorithm 2 presents the Huffman coding algorithm. Given symbols and their frequencies, our goal is to construct a rooted binary tree where the symbols are the labels of the leaves. The algorithm begins with a forest of trees each consisting of one vertex, where each vertex has a symbol as its label and where the weight of this vertex equals the frequency of the symbol that is its label. At each step, we combine two trees having the least total weight into a single tree by introducing a new root and placing the tree with larger weight as its left subtree and the tree with smaller weight as its right subtree. Furthermore, we assign the sum of the weights of the two subtrees of this tree as the total weight of the tree. (Although procedures for breaking ties by choosing between trees with equal weights can be specified, we will not specify such procedures here.) The algorithm is finished when it has constructed a tree, that is, when the forest is reduced to a single tree.

Demo

Links
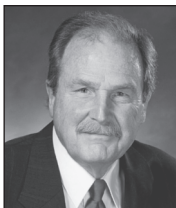
DAVID A. HUFFMAN (1925–1999)   David Huffman grew up in Ohio. At the age of 18 he received his B.S. in electrical engineering from The Ohio State University. Afterward he served in the U.S. Navy as a radar maintenance officer on a destroyer that had the mission of clearing mines in Asian waters after World War II. Later, he earned his M.S. from Ohio State and his Ph.D. in electrical engineering from MIT. Huffman joined the MIT faculty in 1953, where he remained until 1967 when he became the founding member of the computer science department at the University of California at Santa Cruz. He played an important role in developing this department and spent the remainder of his career there, retiring in 1994.

Huffman is noted for his contributions to information theory and coding, signal designs for radar and for communications, and design procedures for asynchronous logical circuits. His work on surfaces with zero curvature led him to develop original techniques for folding paper and vinyl into unusual shapes considered works of art by many and publicly displayed in several exhibits. However, Huffman is best known for his development of what is now called Huffman coding, a result of a term paper he wrote during his graduate work at MIT.

Huffman enjoyed exploring the outdoors, hiking, and traveling extensively. He became certified as a scuba diver when he was in his late 60s. He kept poisonous snakes as pets.

ALGORITHM 2 Huffman Coding.

procedure *Huffman*(C: symbols a_i with frequencies w_i, $i = 1, \ldots, n$)
$F :=$ forest of n rooted trees, each consisting of the single vertex a_i and assigned weight w_i
while F is not a tree
    Replace the rooted trees T and T' of least weights from F with $w(T) \geq w(T')$ with a tree
    having a new root that has T as its left subtree and T' as its right subtree. Label the new
    edge to T with 0 and the new edge to T' with 1.
    Assign $w(T) + w(T')$ as the weight of the new tree.
{the Huffman coding for the symbol a_i is the concatenation of the labels of the edges in the
unique path from the root to the vertex a_i}

Example 5 illustrates how Algorithm 2 is used to encode a set of five symbols.

EXAMPLE 5

Use Huffman coding to encode the following symbols with the frequencies listed: A: 0.08, B: 0.10, C: 0.12, D: 0.15, E: 0.20, F: 0.35. What is the average number of bits used to encode a character?

Solution: Figure 6 displays the steps used to encode these symbols. The encoding produced encodes A by 111, B by 110, C by 011, D by 010, E by 10, and F by 00. The average number of bits used to encode a symbol using this encoding is

$$3 \cdot 0.08 + 3 \cdot 0.10 + 3 \cdot 0.12 + 3 \cdot 0.15 + 2 \cdot 0.20 + 2 \cdot 0.35 = 2.45.$$ ◀

Note that Huffman coding is a greedy algorithm. Replacing the two subtrees with the smallest weight at each step leads to an optimal code in the sense that no binary prefix code for these symbols can encode these symbols using fewer bits. We leave the proof that Huffman codes are optimal as Exercise 32.

There are many variations of Huffman coding. For example, instead of encoding single symbols, we can encode blocks of symbols of a specified length, such as blocks of two symbols. Doing so may reduce the number of bits required to encode the string (see Exercise 30). We can also use more than two symbols to encode the original symbols in the string (see the preamble to Exercise 28). Furthermore, a variation known as adaptive Huffman coding (see [Sa00]) can be used when the frequency of each symbol in a string is not known in advance, so that encoding is done at the same time the string is being read.

Huffman coding is used
in JPEG image coding

Game Trees

Links

Trees can be used to analyze certain types of games such as tic-tac-toe, nim, checkers, and chess. In each of these games, two players take turns making moves. Each player knows the moves made by the other player and no element of chance enters into the game. We model such games using **game trees**; the vertices of these trees represent the positions that a game can be in as it progresses; the edges represent legal moves between these positions. Because game trees are usually large, we simplify game trees by representing all symmetric positions of a game by the same vertex. However, the same position of a game may be represented by different vertices
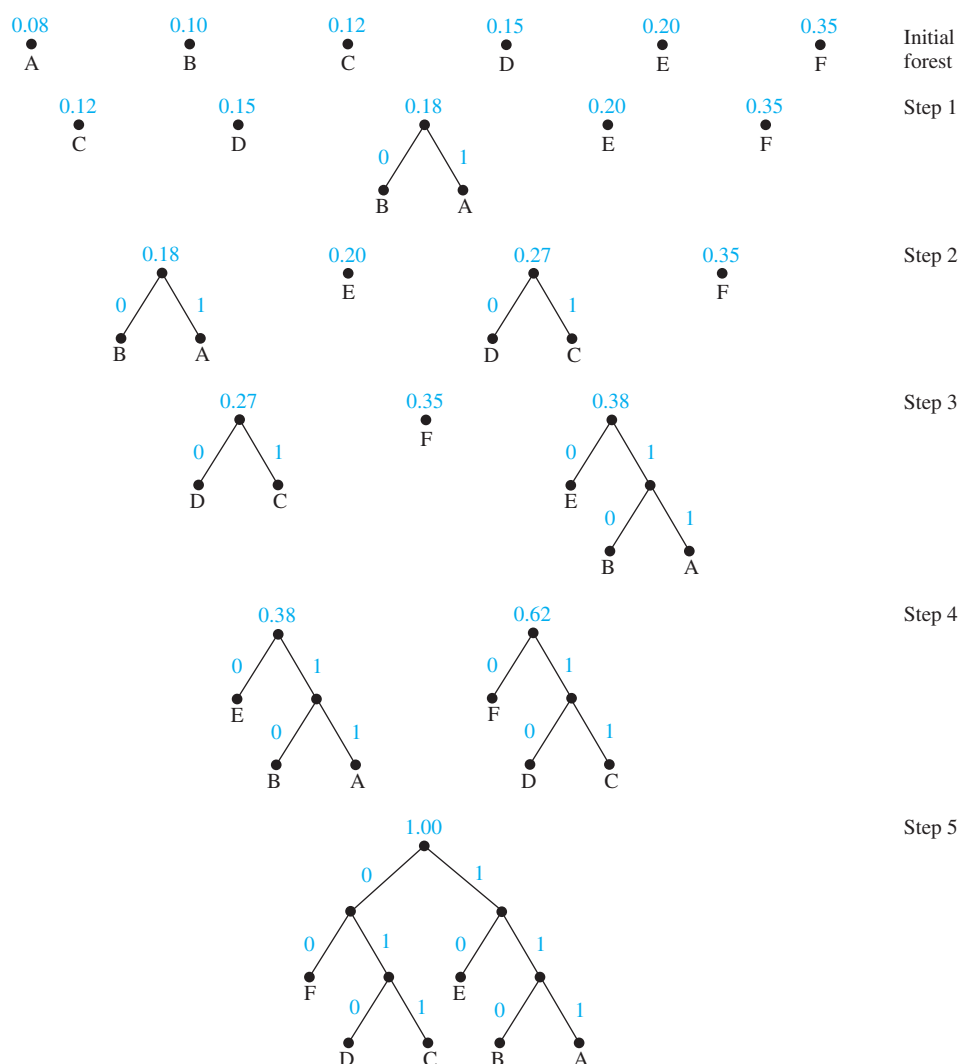
FIGURE 6   **Huffman Coding of Symbols in Example 4.**

if different sequences of moves lead to this position. The root represents the starting position. The usual convention is to represent vertices at even levels by boxes and vertices at odd levels by circles. When the game is in a position represented by a vertex at an even level, it is the first player's move; when the game is in a position represented by a vertex at an odd level, it is the second player's move. Game trees may be infinite when the games they represent never end, such as games that can enter infinite loops, but for most games there are rules that lead to finite game trees.

The leaves of a game tree represent the final positions of a game. We assign a value to each leaf indicating the payoff to the first player if the game terminates in the position represented by this leaf. For games that are win–lose, we label a terminal vertex represented by a circle with a 1 to indicate a win by the first player and we label a terminal vertex represented by a box with a −1 to indicate a win by the second player. For games where draws are allowed, we label a terminal vertex corresponding to a draw position with a 0. Note that for win–lose games, we have assigned values to terminal vertices so that the larger the value, the better the outcome for the first player.

In Example 6 we display a game tree for a well-known and well-studied game.
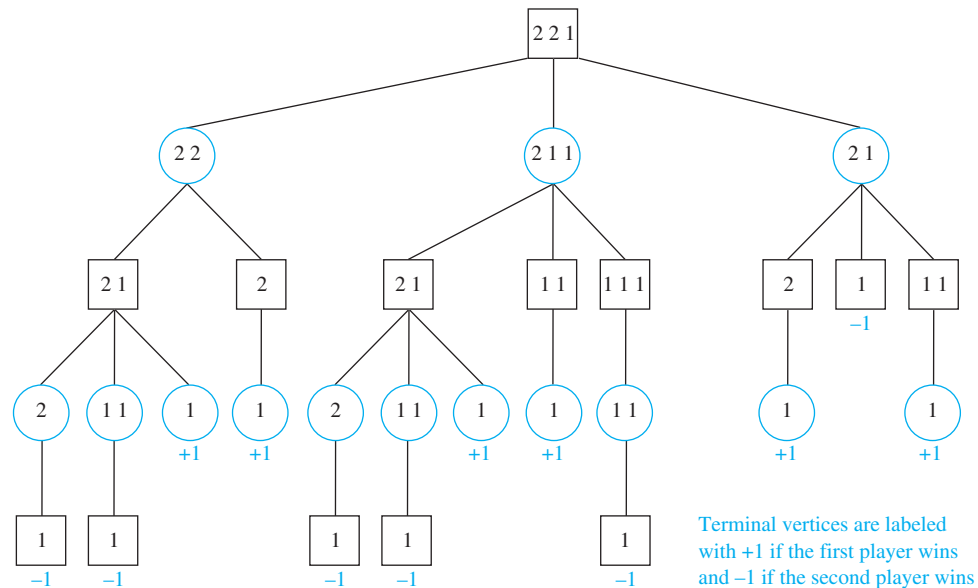
FIGURE 7 The Game Tree for a Game of Nim.

EXAMPLE 6

Although nim is an ancient game, Charles Bouton coined its modern name in 1901 after an archaic English word meaning "to steal."

**Nim** In a version of the game of **nim**, at the start of a game there are a number of piles of stones. Two players take turns making moves; a legal move consists of removing one or more stones from one of the piles, without removing all the stones left. A player without a legal move loses. (Another way to look at this is that the player removing the last stone loses because the position with no piles of stones is not allowed.) The game tree shown in Figure 7 represents this version of nim given the starting position where there are three piles of stones containing two, two, and one stone each, respectively. We represent each position with an unordered list of the number of stones in the different piles (the order of the piles does not matter). The initial move by the first player can lead to three possible positions because this player can remove one stone from a pile with two stones (leaving three piles containing one, one, and two stones); two stones from a pile containing two stones (leaving two piles containing two stones and one stone); or one stone from the pile containing one stone (leaving two piles of two stones). When only one pile with one stone is left, no legal moves are possible, so such positions are terminal positions. Because nim is a win–lose game, we label the terminal vertices with $+1$ when they represent wins for the first player and -1 when they represent wins for the second player. ◄

EXAMPLE 7

Tic-tac-toe The game tree for tic-tac-toe is extremely large and cannot be drawn here, although a computer could easily build such a tree. We show a portion of the game tic-tac-toe in Figure 8(a). Note that by considering symmetric positions equivalent, we need only consider three possible initial moves, as shown in Figure 8(a). We also show a subtree of this game tree leading to terminal positions in Figure 8(b), where a player who can win makes a winning move. ◄

We can recursively define the values of all vertices in a game tree in a way that enables us to determine the outcome of this game when both players follow optimal strategies. By a **strategy** we mean a set of rules that tells a player how to select moves to win the game. An optimal strategy for the first player is a strategy that maximizes the payoff to this player and for the second player is a strategy that minimizes this payoff. We now recursively define the value of a vertex.

(a)

(b)
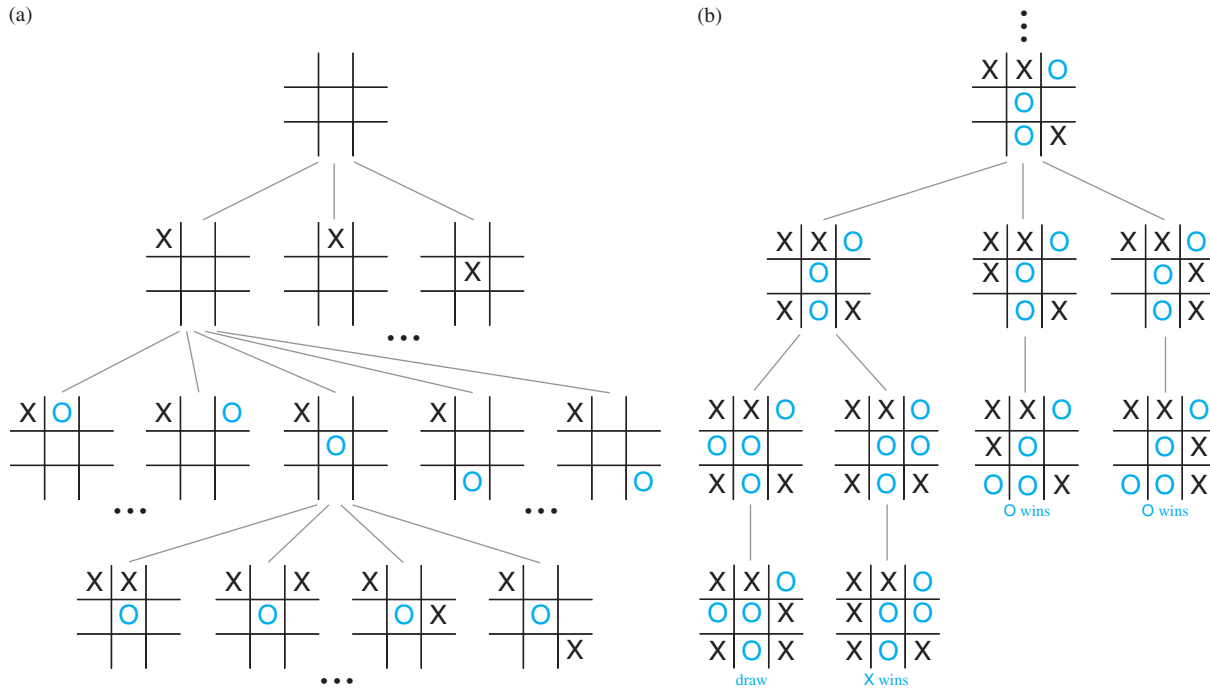
FIGURE 8   **Some of the Game Tree for Tic-Tac-Toe.**

DEFINITION 1   The *value of a vertex in a game tree* is defined recursively as:

   (*i*) the value of a leaf is the payoff to the first player when the game terminates in the position represented by this leaf.

   (*ii*) the value of an internal vertex at an even level is the maximum of the values of its children, and the value of an internal vertex at an odd level is the minimum of the values of its children.

The strategy where the first player moves to a position represented by a child with maximum value and the second player moves to a position of a child with minimum value is called the **minmax strategy**. We can determine who will win the game when both players follow the minmax strategy by calculating the value of the root of the tree; this value is called the **value** of the tree. This is a consequence of Theorem 3.

THEOREM 3   The value of a vertex of a game tree tells us the payoff to the first player if both players follow the minmax strategy and play starts from the position represented by this vertex.

Proof: We will use induction to prove this theorem.

BASIS STEP: If the vertex is a leaf, by definition the value assigned to this vertex is the payoff to the first player.
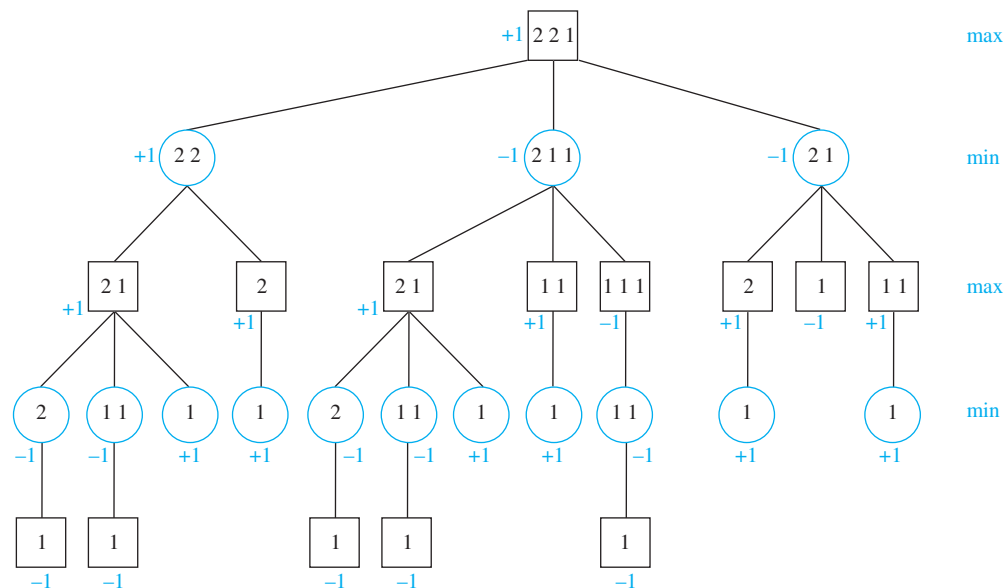
FIGURE 9   **Showing the Values of Vertices in the Game of Nim.**

INDUCTIVE STEP: The inductive hypothesis is the assumption that the values of the children of a vertex are the payoffs to the first player, assuming that play starts at each of the positions represented by these vertices. We need to consider two cases, when it is the first player's turn and when it is the second player's turn.

When it is the first player's turn, this player follows the minmax strategy and moves to the position represented by the child with the largest value. By the inductive hypothesis, this value is the payoff to the first player when play starts at the position represented by this child and follows the minmax strategy. By the recursive step in the definition of the value of an internal vertex at an even level (as the maximum value of its children), the value of this vertex is the payoff when play begins at the position represented by this vertex.

When it is the second player's turn, this player follows the minmax strategy and moves to the position represented by the child with the least value. By the inductive hypothesis, this value is the payoff to the first player when play starts at the position represented by this child and both players follow the minmax strategy. By the recursive definition of the value of an internal vertex at an odd level as the minimum value of its children, the value of this vertex is the payoff when play begins at the position represented by this vertex. ◁

Remark: By extending the proof of Theorem 3, it can be shown that the minmax strategy is the optimal strategy for both players.

Example 8 illustrates how the minmax procedure works. It displays the values assigned to the internal vertices in the game tree from Example 6. Note that we can shorten the computation required by noting that for win–lose games, once a child of a square vertex with value $+1$ is found, the value of the square vertex is also $+1$ because $+1$ is the largest possible payoff. Similarly, once a child of a circle vertex with value -1 is found, this is the value of the circle vertex also.

EXAMPLE 8   In Example 6 we constructed the game tree for nim with a starting position where there are three piles containing two, two, and one stones. In Figure 9 we show the values of the vertices of this game tree. The values of the vertices are computed using the values of the leaves and working one level up at a time. In the right margin of this figure we indicate whether we use the maximum or minimum of the values of the children to find the value of an internal vertex at

each level. For example, once we have found the values of the three children of the root, which are $1, -1$, and -1, we find the value of the root by computing $\max(1, -1, -1) = 1$. Because the value of the root is 1, it follows that the first player wins when both players follow a minmax strategy. ◄

Game trees for some well-known games can be extraordinarily large, because these games have many different possible moves. For example, the game tree for chess has been estimated to have as many as 10^{100} vertices! It may be impossible to use Theorem 3 directly to study a game because of the size of the game tree. Therefore, various approaches have been devised to help determine good strategies and to determine the outcome of such games. One useful technique, called *alpha-beta pruning*, eliminates much computation by pruning portions of the game tree that cannot affect the values of ancestor vertices. (For information about alpha-beta pruning, consult [Gr90].) Another useful approach is to use *evaluation functions*, which estimate the value of internal vertices in the game tree when it is not feasible to compute these values exactly. For example, in the game of tic-tac-toe, as an evaluation function for a position, we may use the number of files (rows, columns, and diagonals) containing no Os (used to indicate moves of the second player) minus the number of files containing no Xs (used to indicate moves of the first player). This evaluation function provides some indication of which player has the advantage in the game. Once the values of an evaluation function are inserted, the value of the game can be computed following the rules used for the minmax strategy. Computer programs created to play chess, such as the famous Deep Blue program, are based on sophisticated evaluation functions. For more information about how computers play chess see [Le91].

Chess programs on smartphones can now play at the grandmaster level.

Links

Exercises

1. Build a binary search tree for the words *banana, peach, apple, pear, coconut, mango*, and *papaya* using alphabetical order.

2. Build a binary search tree for the words *oenology, phrenology, campanology, ornithology, ichthyology, limnology, alchemy*, and *astrology* using alphabetical order.

3. How many comparisons are needed to locate or to add each of these words in the search tree for Exercise 1, starting fresh each time?

  a) *pear*   **b)** *banana*
  c) *kumquat*   **d)** *orange*

4. How many comparisons are needed to locate or to add each of the words in the search tree for Exercise 2, starting fresh each time?

  a) *palmistry*   **b)** *etymology*
  c) *paleontology*   **d)** *glaciology*

5. Using alphabetical order, construct a binary search tree for the words in the sentence "*The quick brown fox jumps over the lazy dog.*"

6. How many weighings of a balance scale are needed to find a lighter counterfeit coin among four coins? Describe an algorithm to find the lighter coin using this number of weighings.

7. How many weighings of a balance scale are needed to find a counterfeit coin among four coins if the counterfeit coin may be either heavier or lighter than the others? Describe an algorithm to find the counterfeit coin using this number of weighings.

*8. How many weighings of a balance scale are needed to find a counterfeit coin among eight coins if the counterfeit coin is either heavier or lighter than the others? Describe an algorithm to find the counterfeit coin using this number of weighings.

*9. How many weighings of a balance scale are needed to find a counterfeit coin among 12 coins if the counterfeit coin is lighter than the others? Describe an algorithm to find the lighter coin using this number of weighings.

*10. One of four coins may be counterfeit. If it is counterfeit, it may be lighter or heavier than the others. How many weighings are needed, using a balance scale, to determine whether there is a counterfeit coin, and if there is, whether it is lighter or heavier than the others? Describe an algorithm to find the counterfeit coin and determine whether it is lighter or heavier using this number of weighings.

11. Find the least number of comparisons needed to sort four elements and devise an algorithm that sorts these elements using this number of comparisons.

*12. Find the least number of comparisons needed to sort five elements and devise an algorithm that sorts these elements using this number of comparisons.

The **tournament sort** is a sorting algorithm that works by building an ordered binary tree. We represent the elements to be sorted by vertices that will become the leaves. We build up the tree one level at a time as we would construct the tree representing the winners of matches in a tournament. Working left to right, we compare pairs of consecutive elements, adding a parent vertex labeled with the larger of the two elements under comparison. We make similar comparisons between labels of vertices at each level until we reach the root of the tree that is labeled with the largest element. The tree constructed by the tournament sort of 22, 8, 14, 17, 3, 9, 27, 11 is illustrated in part (a) of the figure. Once the largest element has been determined, the leaf with this label is relabeled by $-\infty$, which is defined to be less than every element. The labels of all vertices on the path from this vertex up to the root of the tree are recalculated, as shown in part (b) of the figure. This produces the second largest element. This process continues until the entire list has been sorted.
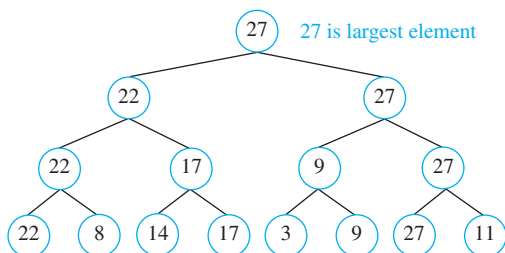

(a)
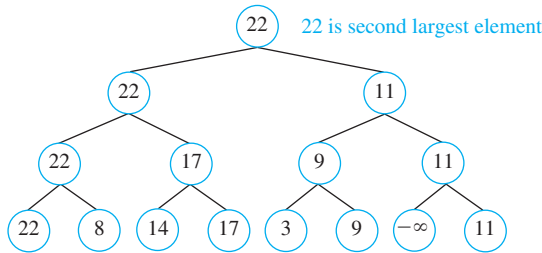
(b)

13. Complete the tournament sort of the list 22, 8, 14, 17, 3, 9, 27, 11. Show the labels of the vertices at each step.

14. Use the tournament sort to sort the list 17, 4, 1, 5, 13, 10, 14, 6.

15. Describe the tournament sort using pseudocode.

16. Assuming that n, the number of elements to be sorted, equals 2^k for some positive integer k, determine the number of comparisons used by the tournament sort to find the largest element of the list using the tournament sort.

17. How many comparisons does the tournament sort use to find the second largest, the third largest, and so on, up to the $(n-1)$st largest (or second smallest) element?

18. Show that the tournament sort requires $\Theta(n \log n)$ comparisons to sort a list of n elements. [*Hint:* By inserting the appropriate number of dummy elements defined to be smaller than all integers, such as $-\infty$, assume that $n = 2^k$ for some positive integer k.]

19. Which of these codes are prefix codes?
   a) a: 11, e: 00, t: 10, s: 01
   b) a: 0, e: 1, t: 01, s: 001
   c) a: 101, e: 11, t: 001, s: 011, n: 010
   d) a: 010, e: 11, t: 011, s: 1011, n: 1001, i: 10101

20. Construct the binary tree with prefix codes representing these coding schemes.
   a) a: 11, e: 0, t: 101, s: 100
   b) a: 1, e: 01, t: 001, s: 0001, n: 00001
   c) a: 1010, e: 0, t: 11, s: 1011, n: 1001, i: 10001

**21.** What are the codes for *a, e, i, k, o, p,* and *u* if the coding scheme is represented by this tree?
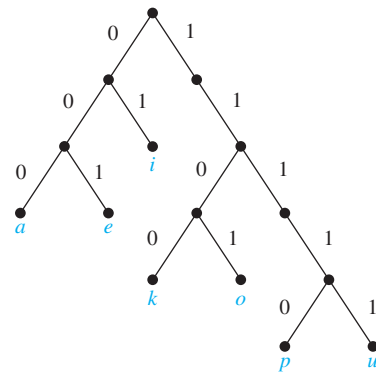
22. Given the coding scheme a: 001, b: 0001, e: 1, r: 0000, s: 0100, t: 011, x: 01010, find the word represented by
   a) 01110100011.   **b)** 0001110000.
   c) 0100101010.   **d)** 01100101010.

23. Use Huffman coding to encode these symbols with given frequencies: a: 0.20, b: 0.10, c: 0.15, d: 0.25, e: 0.30. What is the average number of bits required to encode a character?

24. Use Huffman coding to encode these symbols with given frequencies: A: 0.10, B: 0.25, C: 0.05, D: 0.15, E: 0.30, F: 0.07, G: 0.08. What is the average number of bits required to encode a symbol?

25. Construct two different Huffman codes for these symbols and frequencies: t: 0.2, u: 0.3, v: 0.2, w: 0.3.

26. **a)** Use Huffman coding to encode these symbols with frequencies a: 0.4, b: 0.2, c: 0.2, d: 0.1, e: 0.1 in two different ways by breaking ties in the algorithm differently. First, among the trees of minimum weight select two trees with the largest number of vertices to combine at each stage of the algorithm. Second, among the trees of minimum weight select two trees with the smallest number of vertices at each stage.

   b) Compute the average number of bits required to encode a symbol with each code and compute the variances of this number of bits for each code. Which tie-breaking procedure produced the smaller variance in the number of bits required to encode a symbol?

27. Construct a Huffman code for the letters of the English alphabet where the frequencies of letters in typical English text are as shown in this table.

| Letter	Frequency	Letter	Frequency
A	0.0817	N	0.0662
B	0.0145	O	0.0781
C	0.0248	P	0.0156
D	0.0431	Q	0.0009
E	0.1232	R	0.0572
F	0.0209	S	0.0628
G	0.0182	T	0.0905
H	0.0668	U	0.0304
I	0.0689	V	0.0102
J	0.0010	W	0.0264
K	0.0080	X	0.0015
L	0.0397	Y	0.0211
M	0.0277	Z	0.0005

Suppose that m is a positive integer with $m \geq 2$. An m-ary Huffman code for a set of N symbols can be constructed analogously to the construction of a binary Huffman code. At the initial step, $((N - 1) \bmod (m - 1)) + 1$ trees consisting of a single vertex with least weights are combined into a rooted tree with these vertices as leaves. At each subsequent step, the m trees of least weight are combined into an m-ary tree.

28. Describe the m-ary Huffman coding algorithm in pseudocode.

29. Using the symbols 0, 1, and 2 use ternary ($m = 3$) Huffman coding to encode these letters with the given frequencies: A: 0.25, E: 0.30, N: 0.10, R: 0.05, T: 0.12, Z: 0.18.

30. Consider the three symbols A, B, and C with frequencies A: 0.80, B: 0.19, C: 0.01.

   a) Construct a Huffman code for these three symbols.

   b) Form a new set of nine symbols by grouping together blocks of two symbols, AA, AB, AC, BA, BB, BC, CA, CB, and CC. Construct a Huffman code for these nine symbols, assuming that the occurrences of symbols in the original text are independent.

   c) Compare the average number of bits required to encode text using the Huffman code for the three symbols in part (a) and the Huffman code for the nine blocks of two symbols constructed in part (b). Which is more efficient?

31. Given $n + 1$ symbols $x_1, x_2, \ldots, x_n, x_{n+1}$ appearing 1, $f_1, f_2, \ldots, f_n$ times in a symbol string, respectively, where f_j is the jth Fibonacci number, what is the maximum number of bits used to encode a symbol when all possible tie-breaking selections are considered at each stage of the Huffman coding algorithm?

*32. Show that Huffman codes are optimal in the sense that they represent a string of symbols using the fewest bits among all binary prefix codes.

33. Draw a game tree for nim if the starting position consists of two piles with two and three stones, respectively. When drawing the tree represent by the same vertex symmetric positions that result from the same move. Find the value of each vertex of the game tree. Who wins the game if both players follow an optimal strategy?

34. Draw a game tree for nim if the starting position consists of three piles with one, two, and three stones, respectively. When drawing the tree represent by the same vertex symmetric positions that result from the same move. Find the value of each vertex of the game tree. Who wins the game if both players follow an optimal strategy?

35. Suppose that we vary the payoff to the winning player in the game of nim so that the payoff is n dollars when n is the number of legal moves made before a terminal position is reached. Find the payoff to the first player if the initial position consists of

   a) two piles with one and three stones, respectively.

   b) two piles with two and four stones, respectively.

   c) three piles with one, two, and three stones, respectively.

36. Suppose that in a variation of the game of nim we allow a player to either remove one or more stones from a pile or merge the stones from two piles into one pile as long as at least one stone remains. Draw the game tree for this variation of nim if the starting position consists of three piles containing two, two, and one stone, respectively. Find the values of each vertex in the game tree and determine the winner if both players follow an optimal strategy.

37. Draw the subtree of the game tree for tic-tac-toe beginning at each of these positions. Determine the value of each of these subtrees.
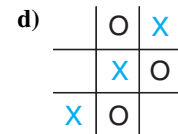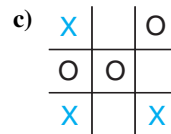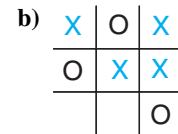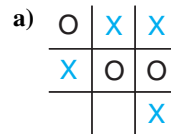
38. Suppose that the first four moves of a tic-tac-toe game are as shown. Does the first player (whose moves are marked by Xs) have a strategy that will always win?
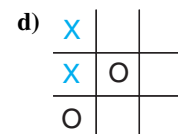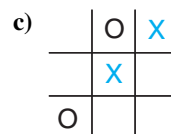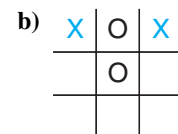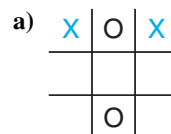
39. Show that if a game of nim begins with two piles containing the same number of stones, as long as this number is at least two, then the second player wins when both players follow optimal strategies.

40. Show that if a game of nim begins with two piles containing different numbers of stones, the first player wins when both players follow optimal strategies.

41. How many children does the root of the game tree for checkers have? How many grandchildren does it have?

42. How many children does the root of the game tree for nim have and how many grandchildren does it have if the starting position is

  a) piles with four and five stones, respectively.

  b) piles with two, three, and four stones, respectively.

  c) piles with one, two, three, and four stones, respectively.

  d) piles with two, two, three, three, and five stones, respectively.

43. Draw the game tree for the game of tic-tac-toe for the levels corresponding to the first two moves. Assign the value of the evaluation function mentioned in the text that assigns to a position the number of files containing no Os minus the number of files containing no Xs as the value of each vertex at this level and compute the value of the tree for vertices as if the evaluation function gave the correct values for these vertices.

44. Use pseudocode to describe an algorithm for determining the value of a game tree when both players follow a minmax strategy.

11.3   Tree Traversal

Introduction

Links

Ordered rooted trees are often used to store information. We need procedures for visiting each vertex of an ordered rooted tree to access data. We will describe several important algorithms for visiting all the vertices of an ordered rooted tree. Ordered rooted trees can also be used to represent various types of expressions, such as arithmetic expressions involving numbers, variables, and operations. The different listings of the vertices of ordered rooted trees used to represent expressions are useful in the evaluation of these expressions.

Universal Address Systems

Procedures for traversing all vertices of an ordered rooted tree rely on the orderings of children. In ordered rooted trees, the children of an internal vertex are shown from left to right in the drawings representing these directed graphs.

   We will describe one way we can totally order the vertices of an ordered rooted tree. To produce this ordering, we must first label all the vertices. We do this recursively:

1. Label the root with the integer 0. Then label its k children (at level 1) from left to right with $1, 2, 3, \ldots, k$.
2. For each vertex v at level n with label A, label its k_v children, as they are drawn from left to right, with $A.1, A.2, \ldots, A.k_v$.

Following this procedure, a vertex v at level n, for $n \geq 1$, is labeled $x_1.x_2.\ldots.x_n$, where the unique path from the root to v goes through the x_1st vertex at level 1, the x_2nd vertex at level 2, and so on. This labeling is called the **universal address system** of the ordered rooted tree.

   We can totally order the vertices using the lexicographic ordering of their labels in the universal address system. The vertex labeled $x_1.x_2.\ldots.x_n$ is less than the vertex labeled $y_1.y_2.\ldots.y_m$ if there is an i, $0 \leq i \leq n$, with $x_1 = y_1, x_2 = y_2, \ldots, x_{i-1} = y_{i-1}$, and $x_i < y_i$; or if $n < m$ and $x_i = y_i$ for $i = 1, 2, \ldots, n$.
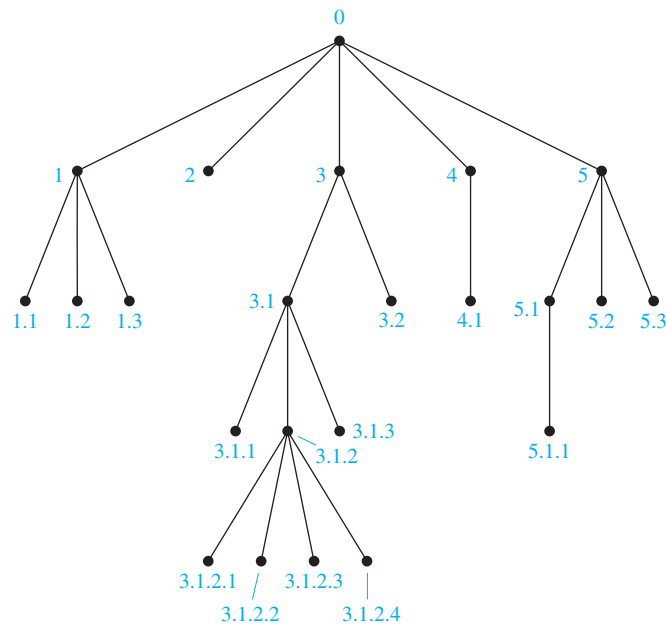
FIGURE 1   The Universal Address System of an Ordered Rooted Tree.

EXAMPLE 1   We display the labelings of the universal address system next to the vertices in the ordered rooted tree shown in Figure 1. The lexicographic ordering of the labelings is

$$0 < 1 < 1.1 < 1.2 < 1.3 < 2 < 3 < 3.1 < 3.1.1 < 3.1.2 < 3.1.2.1 < 3.1.2.2$$
$$< 3.1.2.3 < 3.1.2.4 < 3.1.3 < 3.2 < 4 < 4.1 < 5 < 5.1 < 5.1.1 < 5.2 < 5.3$$

◀

Traversal Algorithms

Procedures for systematically visiting every vertex of an ordered rooted tree are called **traversal algorithms**. We will describe three of the most commonly used such algorithms, **preorder traversal**, **inorder traversal**, and **postorder traversal**. Each of these algorithms can be defined recursively. We first define preorder traversal.

DEFINITION 1   Let T be an ordered rooted tree with root r. If T consists only of r, then r is the *preorder traversal* of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right in T. The *preorder traversal* begins by visiting r. It continues by traversing T_1 in preorder, then T_2 in preorder, and so on, until T_n is traversed in preorder.

The reader should verify that the preorder traversal of an ordered rooted tree gives the same ordering of the vertices as the ordering obtained using a universal address system. Figure 2 indicates how a preorder traversal is carried out.
    Example 2 illustrates preorder traversal.

EXAMPLE 2   In which order does a preorder traversal visit the vertices in the ordered rooted tree T shown in Figure 3?

Solution: The steps of the preorder traversal of T are shown in Figure 4. We traverse T in preorder by first listing the root a, followed by the preorder list of the subtree with root b, the preorder list of the subtree with root c (which is just c) and the preorder list of the subtree with root d.
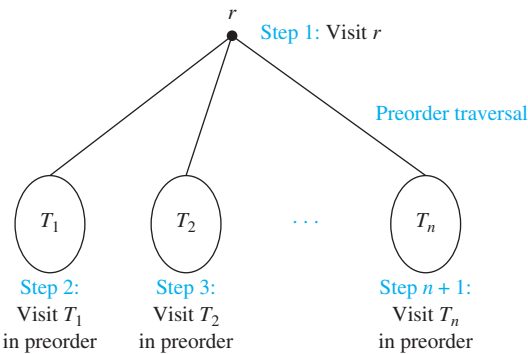
FIGURE 2 Preorder Traversal.
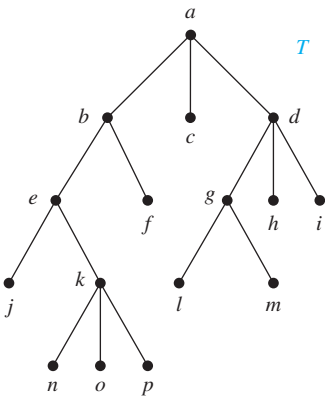
FIGURE 3 The Ordered Rooted Tree T.
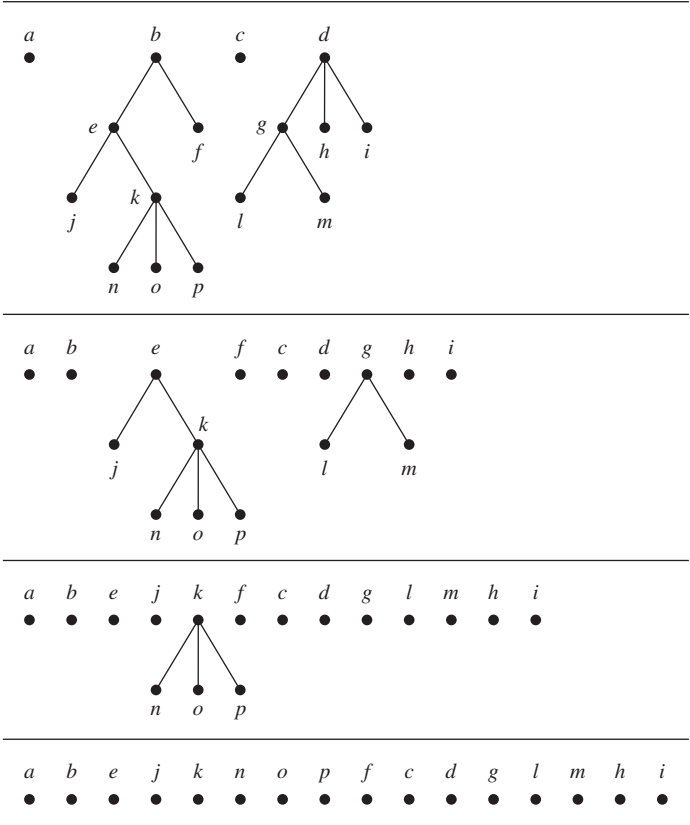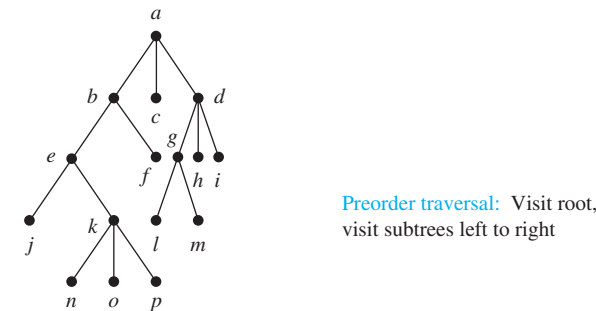
FIGURE 4 The Preorder Traversal of T.
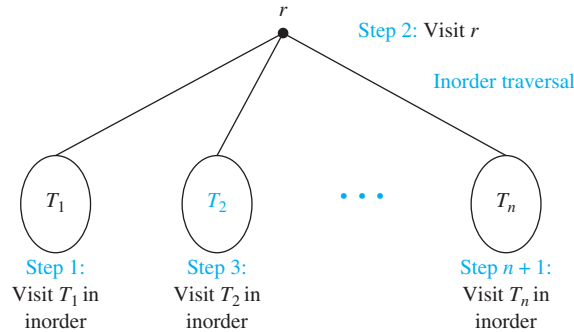
FIGURE 5   Inorder Traversal.

The preorder list of the subtree with root b begins by listing b, then the vertices of the subtree with root e in preorder, and then the subtree with root f in preorder (which is just f). The preorder list of the subtree with root d begins by listing d, followed by the preorder list of the subtree with root g, followed by the subtree with root h (which is just h), followed by the subtree with root i (which is just i).

The preorder list of the subtree with root e begins by listing e, followed by the preorder listing of the subtree with root j (which is just j), followed by the preorder listing of the subtree with root k. The preorder listing of the subtree with root g is g followed by l, followed by m. The preorder listing of the subtree with root k is k, n, o, p. Consequently, the preorder traversal of T is $a, b, e, j, k, n, o, p, f, c, d, g, l, m, h, i$. ◀

We will now define inorder traversal.

DEFINITION 2   Let T be an ordered rooted tree with root r. If T consists only of r, then r is the *inorder traversal* of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right. The *inorder traversal* begins by traversing T_1 in inorder, then visiting r. It continues by traversing T_2 in inorder, then T_3 in inorder, ..., and finally T_n in inorder.

Figure 5 indicates how inorder traversal is carried out. Example 3 illustrates how inorder traversal is carried out for a particular tree.

**EXAMPLE 3**   In which order does an inorder traversal visit the vertices of the ordered rooted tree T in Figure 3?

Solution: The steps of the inorder traversal of the ordered rooted tree T are shown in Figure 6. The inorder traversal begins with an inorder traversal of the subtree with root b, the root a, the inorder listing of the subtree with root c, which is just c, and the inorder listing of the subtree with root d.

The inorder listing of the subtree with root b begins with the inorder listing of the subtree with root e, the root b, and f. The inorder listing of the subtree with root d begins with the inorder listing of the subtree with root g, followed by the root d, followed by h, followed by i.

The inorder listing of the subtree with root e is j, followed by the root e, followed by the inorder listing of the subtree with root k. The inorder listing of the subtree with root g is l, g, m. The inorder listing of the subtree with root k is n, k, o, p. Consequently, the inorder listing of the ordered rooted tree is $j, e, n, k, o, p, b, f, a, c, l, g, m, d, h, i$. ◀

Inorder traversal: Visit leftmost subtree, visit root, visit other subtrees left to right

FIGURE 6 The Inorder Traversal of T.

We now define postorder traversal.

DEFINITION 3

Let T be an ordered rooted tree with root r. If T consists only of r, then r is the *postorder traversal* of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right. The *postorder traversal* begins by traversing T_1 in postorder, then T_2 in postorder, ..., then T_n in postorder, and ends by visiting r.

Figure 7 illustrates how postorder traversal is done. Example 4 illustrates how postorder traversal works.
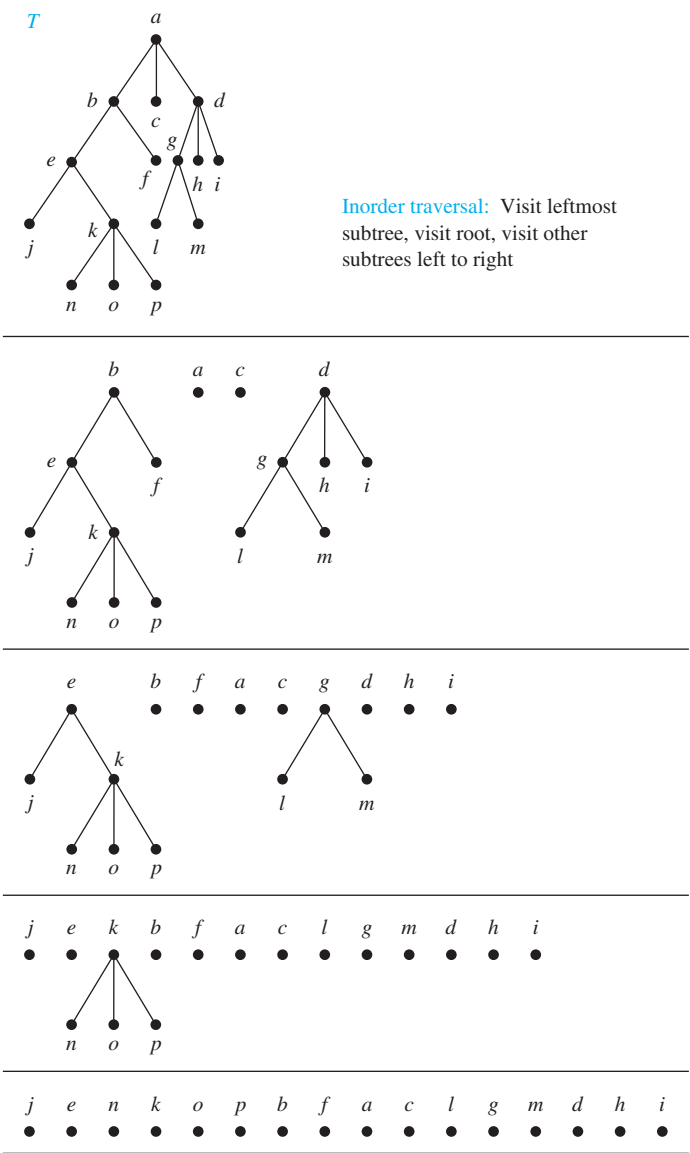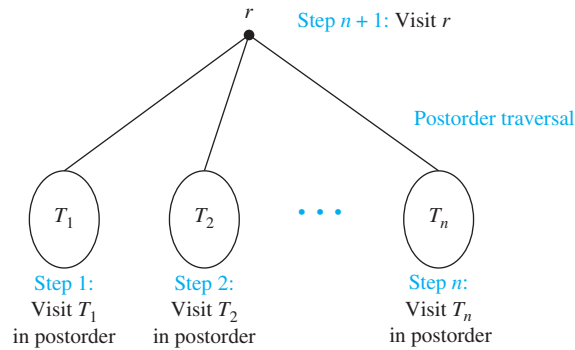
FIGURE 7   **Postorder Traversal.**

EXAMPLE 4   In which order does a postorder traversal visit the vertices of the ordered rooted tree T shown in Figure 3?

Solution: The steps of the postorder traversal of the ordered rooted tree T are shown in Figure 8. The postorder traversal begins with the postorder traversal of the subtree with root b, the postorder traversal of the subtree with root c, which is just c, the postorder traversal of the subtree with root d, followed by the root a.

The postorder traversal of the subtree with root b begins with the postorder traversal of the subtree with root e, followed by f, followed by the root b. The postorder traversal of the rooted tree with root d begins with the postorder traversal of the subtree with root g, followed by h, followed by i, followed by the root d.

The postorder traversal of the subtree with root e begins with j, followed by the postorder traversal of the subtree with root k, followed by the root e. The postorder traversal of the subtree with root g is l, m, g. The postorder traversal of the subtree with root k is n, o, p, k. Therefore, the postorder traversal of T is $j, n, o, p, k, e, f, b, c, l, m, g, h, i, d, a$.   ◄

There are easy ways to list the vertices of an ordered rooted tree in preorder, inorder, and postorder. To do this, first draw a curve around the ordered rooted tree starting at the root, moving along the edges, as shown in the example in Figure 9. We can list the vertices in preorder by listing each vertex the first time this curve passes it. We can list the vertices in inorder by listing a leaf the first time the curve passes it and listing each internal vertex the second time the curve passes it. We can list the vertices in postorder by listing a vertex the last time it is passed on the way back up to its parent. When this is done in the rooted tree in Figure 9, it follows that the preorder traversal gives $a, b, d, h, e, i, j, c, f, g, k$, the inorder traversal gives $h, d, b, i, e, j, a, f, c, k, g$; and the postorder traversal gives $h, d, i, j, e, b, f, k, g, c, a$.

Algorithms for traversing ordered rooted trees in preorder, inorder, or postorder are most easily expressed recursively.

ALGORITHM 1 **Preorder Traversal.**

procedure *preorder*(T: ordered rooted tree)
$r :=$ root of T
list r
for each child c of r from left to right
    $T(c) :=$ subtree with c as its root
    preorder($T(c)$)
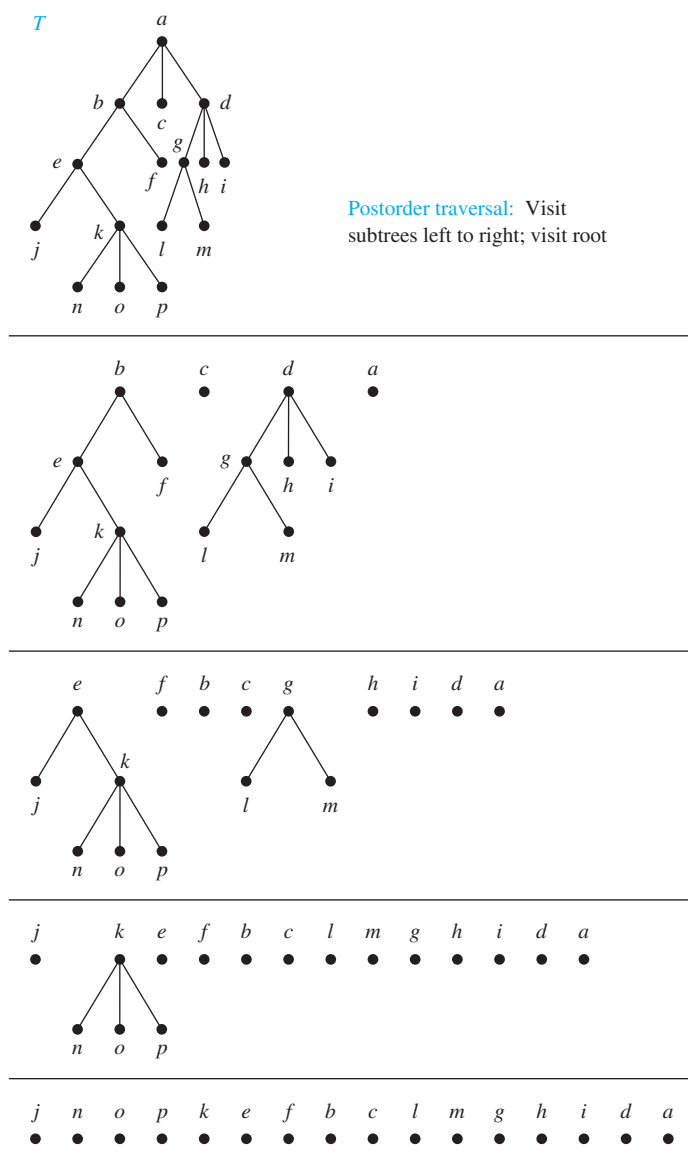
FIGURE 8   The Postorder Traversal of *T*.
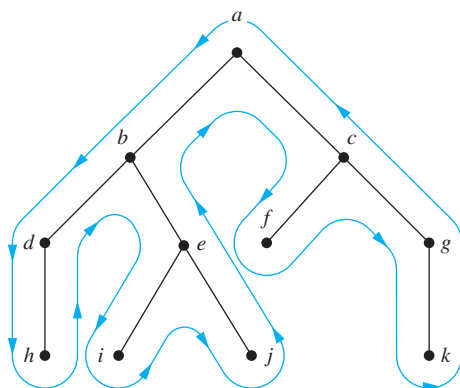
FIGURE 9   A Shortcut for Traversing an Ordered Rooted Tree in Preorder, Inorder, and Postorder.

ALGORITHM 2 Inorder Traversal.

procedure *inorder*(*T*: ordered rooted tree)
r := root of *T*
if *r* is a leaf **then** list *r*
else
   l := first child of *r* from left to right
   T(*l*) := subtree with *l* as its root
   inorder(*T*(*l*))
   list *r*
   for each child *c* of *r* except for *l* from left to right
     T(*c*) := subtree with *c* as its root
     inorder(*T*(*c*))

ALGORITHM 3 Postorder Traversal.

procedure *postorder*(*T*: ordered rooted tree)
r := root of *T*
for each child *c* of *r* from left to right
   T(*c*) := subtree with *c* as its root
   postorder(*T*(*c*))
list *r*

Note that both the preorder traversal and the postorder traversal encode the structure of an ordered rooted tree when the number of children of each vertex is specified. That is, an ordered rooted tree is uniquely determined when we specify a list of vertices generated by a preorder traversal or by a postorder traversal of the tree, together with the number of children of each vertex (see Exercises 26 and 27). In particular, both a preorder traversal and a postorder traversal encode the structure of a full ordered m-ary tree. However, when the number of children of vertices is not specified, neither a preorder traversal nor a postorder traversal encodes the structure of an ordered rooted tree (see Exercises 28 and 29).

Infix, Prefix, and Postfix Notation

We can represent complicated expressions, such as compound propositions, combinations of sets, and arithmetic expressions using ordered rooted trees. For instance, consider the representation of an arithmetic expression involving the operators + (addition), − (subtraction), ∗ (multiplication), / (division), and ↑ (exponentiation). We will use parentheses to indicate the order of the operations. An ordered rooted tree can be used to represent such expressions, where the internal vertices represent operations, and the leaves represent the variables or numbers. Each operation operates on its left and right subtrees (in that order).

EXAMPLE 5 What is the ordered rooted tree that represents the expression $((x + y) \uparrow 2) + ((x - 4)/3)$?

Solution: The binary tree for this expression can be built from the bottom up. First, a subtree for the expression $x + y$ is constructed. Then this is incorporated as part of the larger subtree representing $(x + y) \uparrow 2$. Also, a subtree for $x - 4$ is constructed, and then this is incorporated into a subtree representing $(x - 4)/3$. Finally the subtrees representing $(x + y) \uparrow 2$
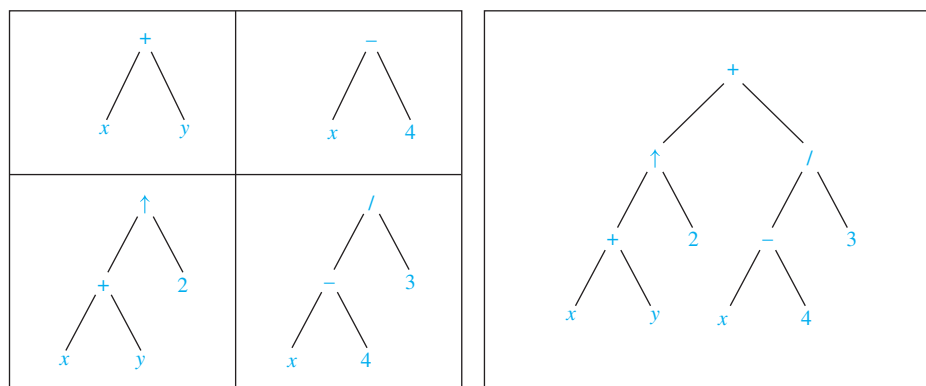
FIGURE 10 A Binary Tree Representing $((x + y) \uparrow 2) + ((x - 4)/3)$.

and $(x - 4)/3$ are combined to form the ordered rooted tree representing $((x + y) \uparrow 2) + ((x - 4)/3)$. These steps are shown in Figure 10. ◀

An inorder traversal of the binary tree representing an expression produces the original expression with the elements and operations in the same order as they originally occurred, except for unary operations, which instead immediately follow their operands. For instance, inorder traversals of the binary trees in Figure 11, which represent the expressions $(x + y)/(x + 3)$, $(x + (y/x)) + 3$, and $x + (y/(x + 3))$, all lead to the infix expression $x + y/x + 3$. To make such expressions unambiguous it is necessary to include parentheses in the inorder traversal whenever we encounter an operation. The fully parenthesized expression obtained in this way is said to be in **infix form**.

We obtain the **prefix form** of an expression when we traverse its rooted tree in preorder. Expressions written in prefix form are said to be in **Polish notation**, which is named after the Polish logician Jan Łukasiewicz. An expression in prefix notation (where each operation has a specified number of operands), is unambiguous, so no parentheses are needed in such an expression. The verification of this is left as an exercise for the reader.

EXAMPLE 6 What is the prefix form for $((x + y) \uparrow 2) + ((x - 4)/3)$?

Solution: We obtain the prefix form for this expression by traversing the binary tree that represents it in preorder, shown in Figure 10. This produces $+ \uparrow + x\ y\ 2 / - x\ 4\ 3$. ◀

In the prefix form of an expression, a binary operator, such as $+$, precedes its two operands. Hence, we can evaluate an expression in prefix form by working from right to left. When we encounter an operator, we perform the corresponding operation with the two operands
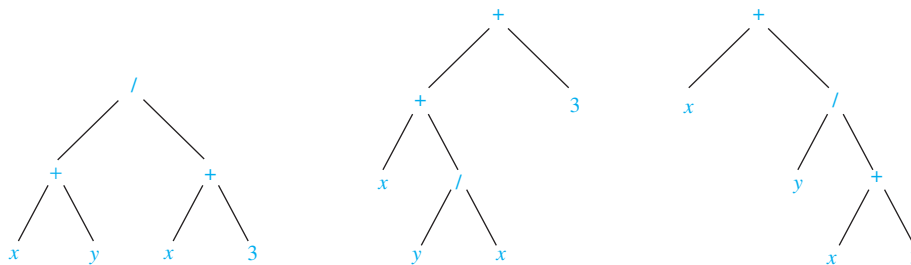
FIGURE 11 Rooted Trees Representing $(x + y)/(x + 3)$, $(x + (y/x)) + 3$, and $x + (y/(x + 3))$.

Left column (Figure 12):

```
+  −  *  2  3  5  /  ↑  2  3  4
                    └─────┘
                    2↑3 = 8

   +  −  *  2  3  5  /  8  4
                    └────┘
                    8 / 4 = 2

      +  −  *  2  3  5  2
            └────┘
            2 * 3 = 6

         +  −  6  5  2
            └────┘
            6 − 5 = 1

            +  1  2
            └────┘
            1 + 2 = 3
```
Value of expression: 3
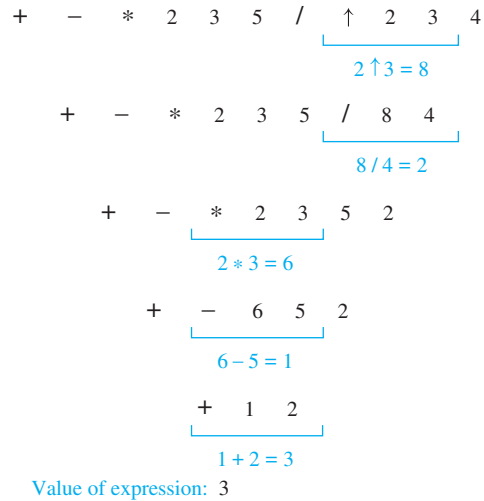
FIGURE 12   Evaluating a Prefix Expression.

Right column (Figure 13):

```
7  2  3  *  −  4  ↑  9  3  /  +
   └────┘
   2 * 3 = 6

7  6  −  4  ↑  9  3  /  +
└────┘
7 − 6 = 1

   1  4  ↑  9  3  /  +
   └────┘
   1⁴ = 1

      1  9  3  /  +
         └────┘
         9 / 3 = 3

         1  3  +
         └────┘
         1 + 3 = 4
```
Value of expression: 4

FIGURE 13   Evaluating a Postfix Expression.

immediately to the right of this operand. Also, whenever an operation is performed, we consider the result a new operand.

EXAMPLE 7   What is the value of the prefix expression $+ - * \, 2 \, 3 \, 5/ \uparrow 2 \, 3 \, 4$?

Solution: The steps used to evaluate this expression by working right to left, and performing operations using the operands on the right, are shown in Figure 12. The value of this expression is 3. ◀

We obtain the **postfix form** of an expression by traversing its binary tree in postorder. Expressions written in postfix form are said to be in **reverse Polish notation**. Expressions in reverse Polish notation are unambiguous, so parentheses are not needed. The verification of this is left to the reader. Reverse polish notation was extensively used in electronic calculators in the 1970s and 1980s.
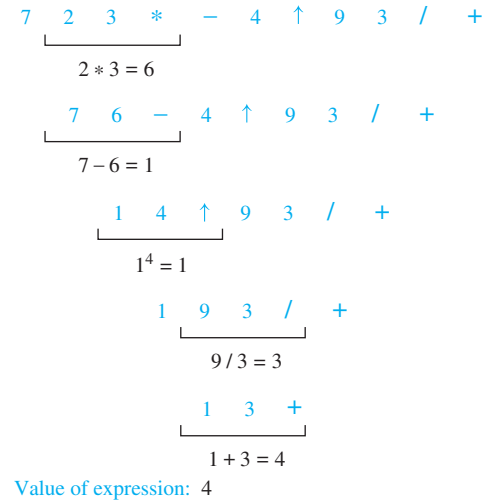
EXAMPLE 8   What is the postfix form of the expression $((x + y) \uparrow 2) + ((x - 4)/3)$?

Solution: The postfix form of the expression is obtained by carrying out a postorder traversal of the binary tree for this expression, shown in Figure 10. This produces the postfix expression: $x \, y + 2 \uparrow x \, 4 - 3 \,/\, +$. ◀

In the postfix form of an expression, a binary operator follows its two operands. So, to evaluate an expression from its postfix form, work from left to right, carrying out operations whenever an operator follows two operands. After an operation is carried out, the result of this operation becomes a new operand.

EXAMPLE 9   What is the value of the postfix expression $7 \, 2 \, 3 \, * - 4 \uparrow 9 \, 3/+$?

Solution: The steps used to evaluate this expression by starting at the left and carrying out operations when two operands are followed by an operator are shown in Figure 13. The value of this expression is 4. ◀
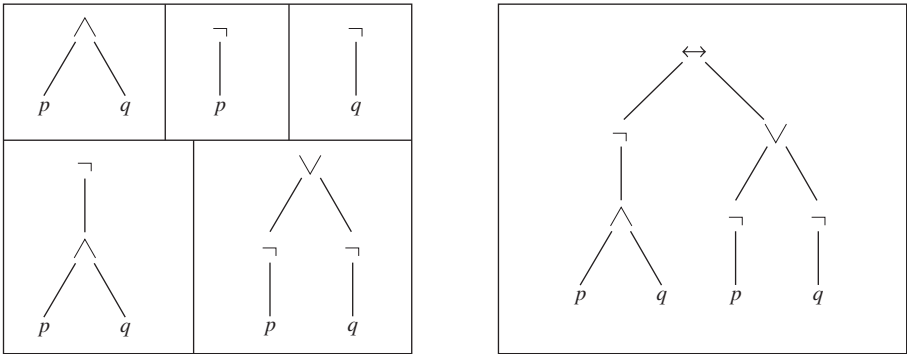
FIGURE 14   Constructing the Rooted Tree for a Compound Proposition.

Rooted trees can be used to represent other types of expressions, such as those representing compound propositions and combinations of sets. In these examples unary operators, such as the negation of a proposition, occur. To represent such operators and their operands, a vertex representing the operator and a child of this vertex representing the operand are used.

EXAMPLE 10   Find the ordered rooted tree representing the compound proposition $(\neg(p \wedge q)) \leftrightarrow (\neg p \vee \neg q)$. Then use this rooted tree to find the prefix, postfix, and infix forms of this expression.

Extra
Examples

Solution: The rooted tree for this compound proposition is constructed from the bottom up. First, subtrees for $\neg p$ and $\neg q$ are formed (where $\neg$ is considered a unary operator). Also, a subtree for $p \wedge q$ is formed. Then subtrees for $\neg(p \wedge q)$ and $(\neg p) \vee (\neg q)$ are constructed. Finally, these two subtrees are used to form the final rooted tree. The steps of this procedure are shown in Figure 14.

The prefix, postfix, and infix forms of this expression are found by traversing this rooted tree in preorder, postorder, and inorder (including parentheses), respectively. These traversals give $\leftrightarrow \neg \wedge pq \vee \neg p \neg q$, $pq \wedge \neg p \neg q \neg \vee \leftrightarrow$, and $(\neg(p \wedge q)) \leftrightarrow ((\neg p) \vee (\neg q))$, respectively. ◀

Because prefix and postfix expressions are unambiguous and because they can be evaluated easily without scanning back and forth, they are used extensively in computer science. Such expressions are especially useful in the construction of compilers.

Links

Exercises

In Exercises 1–3 construct the universal address system for the given ordered rooted tree. Then use this to order its vertices using the lexicographic order of their labels.

1.
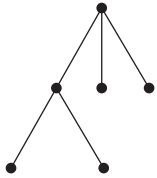
2.
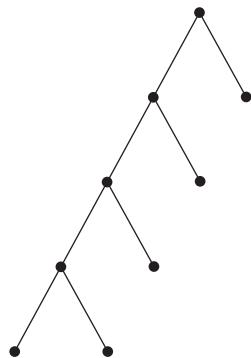
3.
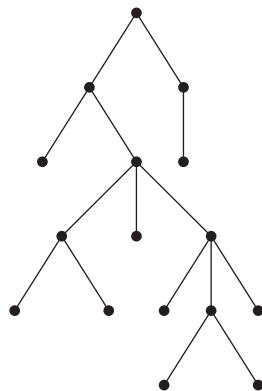
4. Suppose that the address of the vertex v in the ordered rooted tree T is 3.4.5.2.4.

   a) At what level is v?

   b) What is the address of the parent of v?

   c) What is the least number of siblings v can have?

   d) What is the smallest possible number of vertices in T if v has this address?

   e) Find the other addresses that must occur.

5. Suppose that the vertex with the largest address in an ordered rooted tree T has address 2.3.4.3.1. Is it possible to determine the number of vertices in T?

6. Can the leaves of an ordered rooted tree have the following list of universal addresses? If so, construct such an ordered rooted tree.

   a) 1.1.1, 1.1.2, 1.2, 2.1.1.1, 2.1.2, 2.1.3, 2.2, 3.1.1, 3.1.2.1, 3.1.2.2, 3.2

   b) 1.1, 1.2.1, 1.2.2, 1.2.3, 2.1, 2.2.1, 2.3.1, 2.3.2, 2.4.2.1, 2.4.2.2, 3.1, 3.2.1, 3.2.2

   c) 1.1, 1.2.1, 1.2.2, 1.2.2.1, 1.3, 1.4, 2, 3.1, 3.2, 4.1.1.1

In Exercises 7–9 determine the order in which a preorder traversal visits the vertices of the given ordered rooted tree.

7.

8.

9.
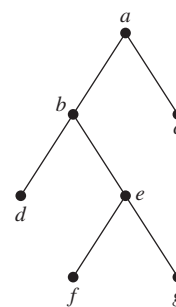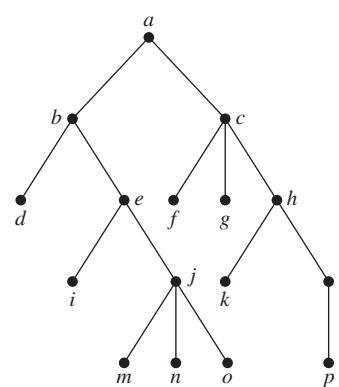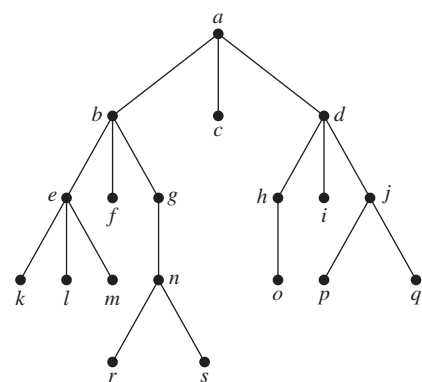
10. In which order are the vertices of the ordered rooted tree in Exercise 7 visited using an inorder traversal?

11. In which order are the vertices of the ordered rooted tree in Exercise 8 visited using an inorder traversal?

12. In which order are the vertices of the ordered rooted tree in Exercise 9 visited using an inorder traversal?

13. In which order are the vertices of the ordered rooted tree in Exercise 7 visited using a postorder traversal?

14. In which order are the vertices of the ordered rooted tree in Exercise 8 visited using a postorder traversal?

15. In which order are the vertices of the ordered rooted tree in Exercise 9 visited using a postorder traversal?

16. a) Represent the expression $((x + 2) \uparrow 3) * (y - (3 + x)) - 5$ using a binary tree.
Write this expression in

b) prefix notation.
c) postfix notation.
d) infix notation.

17. a) Represent the expressions $(x + xy) + (x/y)$ and $x + ((xy + x)/y)$ using binary trees.
Write these expressions in

b) prefix notation.
c) postfix notation.
d) infix notation.

18. a) Represent the compound propositions $\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q)$ and $(\neg p \wedge (q \leftrightarrow \neg p)) \vee \neg q$ using ordered rooted trees.
Write these expressions in

b) prefix notation.
c) postfix notation.
d) infix notation.

19. a) Represent $(A \cap B) - (A \cup (B - A))$ using an ordered rooted tree.
Write this expression in

b) prefix notation.
c) postfix notation.
d) infix notation.

*20. In how many ways can the string $\neg p \wedge q \leftrightarrow \neg p \vee \neg q$ be fully parenthesized to yield an infix expression?

*21. In how many ways can the string $A \cap B - A \cap B - A$ be fully parenthesized to yield an infix expression?

22. Draw the ordered rooted tree corresponding to each of these arithmetic expressions written in prefix notation. Then write each expression using infix notation.

a) $+ * + - 5\ 3\ 2\ 1\ 4$
b) $\uparrow + 2\ 3 - 5\ 1$
c) $* / 9\ 3 + * 2\ 4 - 7\ 6$

23. What is the value of each of these prefix expressions?

a) $- * 2 / 8\ 4\ 3$
b) $\uparrow - * 3\ 3 * 4\ 2\ 5$
c) $+ - \uparrow 3\ 2 \uparrow 2\ 3 / 6 - 4\ 2$
d) $* + 3 + 3 \uparrow 3 + 3\ 3\ 3$

24. What is the value of each of these postfix expressions?

a) $5\ 2\ 1 - - 3\ 1\ 4 + + *$
b) $9\ 3 / 5 + 7\ 2 - *$
c) $3\ 2 * 2 \uparrow 5\ 3 - 8\ 4 / * -$

25. Construct the ordered rooted tree whose preorder traversal is $a, b, f, c, g, h, i, d, e, j, k, l$, where a has four children, c has three children, j has two children, b and e have one child each, and all other vertices are leaves.

*26. Show that an ordered rooted tree is uniquely determined when a list of vertices generated by a preorder traversal of the tree and the number of children of each vertex are specified.

*27. Show that an ordered rooted tree is uniquely determined when a list of vertices generated by a postorder traversal of the tree and the number of children of each vertex are specified.

28. Show that preorder traversals of the two ordered rooted trees displayed below produce the same list of vertices. Note that this does not contradict the statement in Exercise 26, because the numbers of children of internal vertices in the two ordered rooted trees differ.
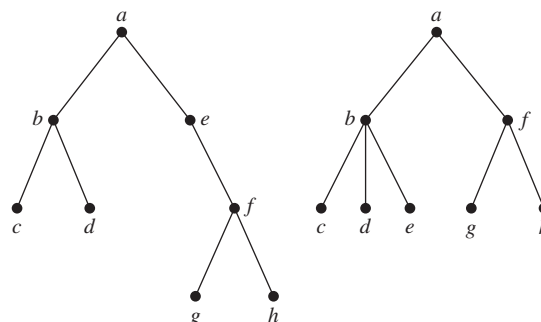
29. Show that postorder traversals of these two ordered rooted trees produce the same list of vertices. Note that this does not contradict the statement in Exercise 27, because the numbers of children of internal vertices in the two ordered rooted trees differ.

Well-formed formulae in prefix notation over a set of symbols and a set of binary operators are defined recursively by these rules:

(*i*) if x is a symbol, then x is a well-formed formula in prefix notation;
(*ii*) if X and Y are well-formed formulae and $*$ is an operator, then $* XY$ is a well-formed formula.

30. Which of these are well-formed formulae over the symbols $\{x, y, z\}$ and the set of binary operators $\{\times, +, \circ\}$?

a) $\times + + x\ y\ x$
b) $\circ x\ y \times x\ z$
c) $\times \circ x\ z \times \times x\ y$
d) $\times + \circ x\ x \circ x\ x\ x$
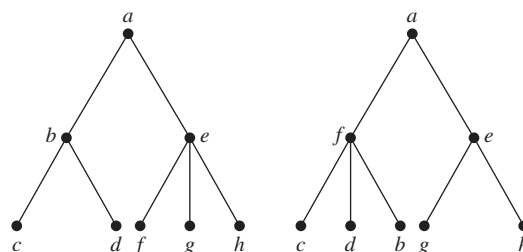
*31. Show that any well-formed formula in prefix notation over a set of symbols and a set of binary operators contains exactly one more symbol than the number of operators.

32. Give a definition of well-formed formulae in postfix notation over a set of symbols and a set of binary operators.

33. Give six examples of well-formed formulae with three or more operators in postfix notation over the set of symbols $\{x, y, z\}$ and the set of operators $\{+, \times, \circ\}$.

34. Extend the definition of well-formed formulae in prefix notation to sets of symbols and operators where the operators may not be binary.
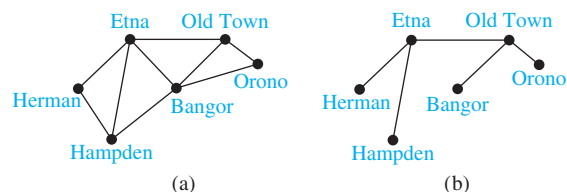
FIGURE 1 (a) A Road System and (b) a Set of Roads to Plow.
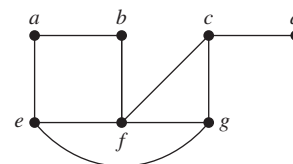
FIGURE 2 The Simple Graph *G*.

11.4 Spanning Trees

Introduction

Consider the system of roads in Maine represented by the simple graph shown in Figure 1(a). The only way the roads can be kept open in the winter is by frequently plowing them. The highway department wants to plow the fewest roads so that there will always be cleared roads connecting any two towns. How can this be done?

At least five roads must be plowed to ensure that there is a path between any two towns. Figure 1(b) shows one such set of roads. Note that the subgraph representing these roads is a tree, because it is connected and contains six vertices and five edges.

This problem was solved with a connected subgraph with the minimum number of edges containing all vertices of the original simple graph. Such a graph must be a tree.

DEFINITION 1 Let *G* be a simple graph. A *spanning tree* of *G* is a subgraph of *G* that is a tree containing every vertex of *G*.

A simple graph with a spanning tree must be connected, because there is a path in the spanning tree between any two vertices. The converse is also true; that is, every connected simple graph has a spanning tree. We will give an example before proving this result.

EXAMPLE 1 Find a spanning tree of the simple graph *G* shown in Figure 2.

Solution: The graph *G* is connected, but it is not a tree because it contains simple circuits. Remove the edge $\{a, e\}$. This eliminates one simple circuit, and the resulting subgraph is still connected and still contains every vertex of *G*. Next remove the edge $\{e, f\}$ to eliminate a second simple circuit. Finally, remove edge $\{c, g\}$ to produce a simple graph with no simple circuits. This subgraph is a spanning tree, because it is a tree that contains every vertex of *G*. The sequence of edge removals used to produce the spanning tree is illustrated in Figure 3.

Edge removed: $\{a, e\}$ $\{e, f\}$ $\{c, g\}$

(a)   (b)   (c)
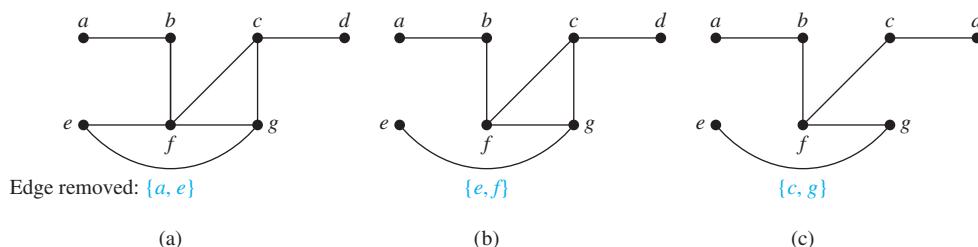
FIGURE 3 Producing a Spanning Tree for *G* by Removing Edges That Form Simple Circuits.
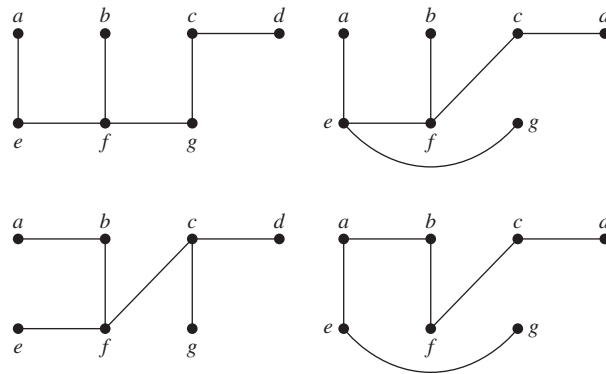
FIGURE 4 **Spanning Trees of G.**

The tree shown in Figure 3 is not the only spanning tree of G. For instance, each of the trees shown in Figure 4 is a spanning tree of G. ◄

THEOREM 1 A simple graph is connected if and only if it has a spanning tree.

Proof: First, suppose that a simple graph G has a spanning tree T. T contains every vertex of G. Furthermore, there is a path in T between any two of its vertices. Because T is a subgraph of G, there is a path in G between any two of its vertices. Hence, G is connected.

Now suppose that G is connected. If G is not a tree, it must contain a simple circuit. Remove an edge from one of these simple circuits. The resulting subgraph has one fewer edge but still contains all the vertices of G and is connected. This subgraph is still connected because when two vertices are connected by a path containing the removed edge, they are connected by a path not containing this edge. We can construct such a path by inserting into the original path, at the point where the removed edge once was, the simple circuit with this edge removed. If this subgraph is not a tree, it has a simple circuit; so as before, remove an edge that is in a simple circuit. Repeat this process until no simple circuits remain. This is possible because there are only a finite number of edges in the graph. The process terminates when no simple circuits remain. A tree is produced because the graph stays connected as edges are removed. This tree is a spanning tree because it contains every vertex of G. ◄

Spanning trees are important in data networking, as Example 2 shows.

EXAMPLE 2

IP Multicasting Spanning trees play an important role in multicasting over Internet Protocol (IP) networks. To send data from a source computer to multiple receiving computers, each of which is a subnetwork, data could be sent separately to each computer. This type of networking, called unicasting, is inefficient, because many copies of the same data are transmitted over the network. To make the transmission of data to multiple receiving computers more efficient, IP multicasting is used. With IP multicasting, a computer sends a single copy of data over the network, and as data reaches intermediate routers, the data are forwarded to one or more other routers so that ultimately all receiving computers in their various subnetworks receive these data. (Routers are computers that are dedicated to forwarding IP datagrams between subnetworks in a network. In multicasting, routers use Class D addresses, each representing a session that receiving computers may join; see Example 17 in Section 6.1.)

For data to reach receiving computers as quickly as possible, there should be no loops (which in graph theory terminology are circuits or cycles) in the path that data take through the network. That is, once data have reached a particular router, data should never return to this

IP network

Multicast spanning tree

(a)

(b)

☐ Router
● Subnetwork
◉ Subnetwork with a receiving station
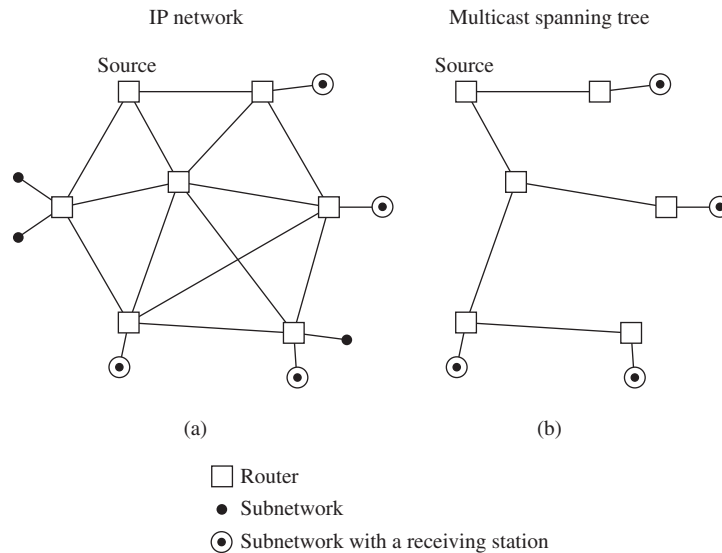
FIGURE 5   **A Multicast Spanning Tree.**

router. To avoid loops, the multicast routers use network algorithms to construct a spanning tree in the graph that has the multicast source, the routers, and the subnetworks containing receiving computers as vertices, with edges representing the links between computers and/or routers. The root of this spanning tree is the multicast source. The subnetworks containing receiving computers are leaves of the tree. (Note that subnetworks not containing receiving stations are not included in the graph.) This is illustrated in Figure 5. ◀

Depth-First Search

Links

Demo

The proof of Theorem 1 gives an algorithm for finding spanning trees by removing edges from simple circuits. This algorithm is inefficient, because it requires that simple circuits be identified. Instead of constructing spanning trees by removing edges, spanning trees can be built up by successively adding edges. Two algorithms based on this principle will be presented here.

We can build a spanning tree for a connected simple graph using **depth-first search**. We will form a rooted tree, and the spanning tree will be the underlying undirected graph of this rooted tree. Arbitrarily choose a vertex of the graph as the root. Form a path starting at this vertex by successively adding vertices and edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding vertices and edges to this path as long as possible. If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree. However, if the path does not go through all vertices, more vertices and edges must be added. Move back to the next to last vertex in the path, and, if possible, form a new path starting at this vertex passing through vertices that were not already visited. If this cannot be done, move back another vertex in the path, that is, two vertices back in the path, and try again.

Repeat this procedure, beginning at the last vertex visited, moving back up the path one vertex at a time, forming new paths that are as long as possible until no more edges can be added. Because the graph has a finite number of edges and is connected, this process ends with the production of a spanning tree. Each vertex that ends a path at a stage of the algorithm will be a leaf in the rooted tree, and each vertex where a path is constructed starting at this vertex will be an internal vertex.
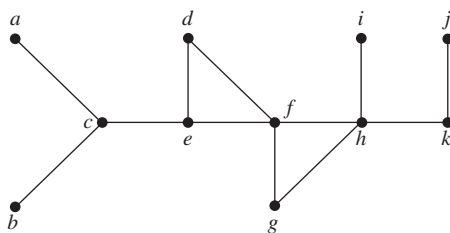
FIGURE 6   The Graph *G*.
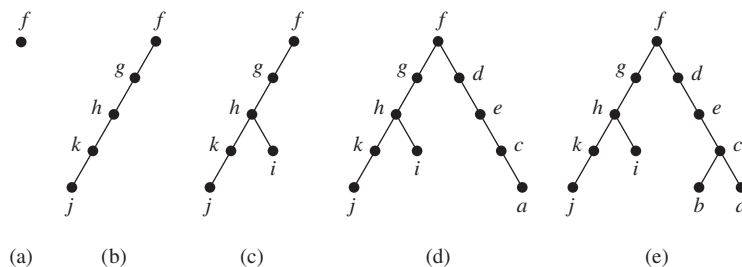
FIGURE 7   Depth-First Search of *G*.

The reader should note the recursive nature of this procedure. Also, note that if the vertices in the graph are ordered, the choices of edges at each stage of the procedure are all determined when we always choose the first vertex in the ordering that is available. However, we will not always explicitly order the vertices of a graph.

Depth-first search is also called **backtracking**, because the algorithm returns to vertices previously visited to add paths. Example 3 illustrates backtracking.

EXAMPLE 3   Use depth-first search to find a spanning tree for the graph *G* shown in Figure 6.

Solution: The steps used by depth-first search to produce a spanning tree of *G* are shown in Figure 7. We arbitrarily start with the vertex *f*. A path is built by successively adding edges incident with vertices not already in the path, as long as this is possible. This produces a path *f*, *g*, *h*, *k*, *j* (note that other paths could have been built). Next, backtrack to *k*. There is no path beginning at *k* containing vertices not already visited. So we backtrack to *h*. Form the path *h*, *i*. Then backtrack to *h*, and then to *f*. From *f* build the path *f*, *d*, *e*, *c*, *a*. Then backtrack to *c* and form the path *c*, *b*. This produces the spanning tree.   ◀

The edges selected by depth-first search of a graph are called **tree edges**. All other edges of the graph must connect a vertex to an ancestor or descendant of this vertex in the tree. These edges are called **back edges**. (Exercise 43 asks for a proof of this fact.)

EXAMPLE 4   In Figure 8 we highlight the tree edges found by depth-first search starting at vertex *f* by showing them with heavy colored lines. The back edges (*e*, *f*) and (*f*, *h*) are shown with thinner black lines.   ◀

We have explained how to find a spanning tree of a graph using depth-first search. However, our discussion so far has not brought out the recursive nature of depth-first search. To help make the recursive nature of the algorithm clear, we need a little terminology. We say that we
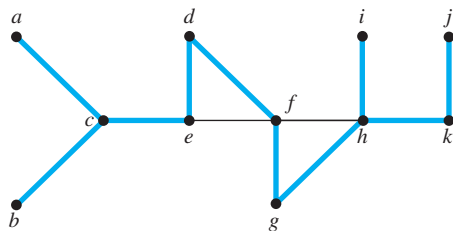
FIGURE 8   The Tree Edges and Back Edges of the Depth-First Search in Example 4.

explore from a vertex v when we carry out the steps of depth-first search beginning when v is added to the tree and ending when we have backtracked back to v for the last time. The key observation needed to understand the recursive nature of the algorithm is that when we add an edge connecting a vertex v to a vertex w, we finish exploring from w before we return to v to complete exploring from v.

In Algorithm 1 we construct the spanning tree of a graph G with vertices $v_1, \ldots, v_n$ by first selecting the vertex v_1 to be the root. We initially set T to be the tree with just this one vertex. At each step we add a new vertex to the tree T together with an edge from a vertex already in T to this new vertex and we explore from this new vertex. Note that at the completion of the algorithm, T contains no simple circuits because no edge is ever added that connects two vertices in the tree. Moreover, T remains connected as it is built. (These last two observations can be easily proved via mathematical induction.) Because G is connected, every vertex in G is visited by the algorithm and is added to the tree (as the reader should verify). It follows that T is a spanning tree of G.

ALGORITHM 1 Depth-First Search.

procedure *DFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of the vertex v_1
visit(v_1)

procedure *visit*(v: vertex of G)
for each vertex w adjacent to v and not yet in T
   add vertex w and edge $\{v, w\}$ to T
   visit(w)

We now analyze the computational complexity of the depth-first search algorithm. The key observation is that for each vertex v, the procedure *visit*(v) is called when the vertex v is first encountered in the search and it is not called again. Assuming that the adjacency lists for G are available (see Section 10.3), no computations are required to find the vertices adjacent to v. As we follow the steps of the algorithm, we examine each edge at most twice to determine whether to add this edge and one of its endpoints to the tree. Consequently, the procedure *DFS* constructs a spanning tree using $O(e)$, or $O(n^2)$, steps where e and n are the number of edges and vertices in G, respectively. [Note that a step involves examining a vertex to see whether it is already in the spanning tree as it is being built and adding this vertex and the corresponding edge if the vertex is not already in the tree. We have also made use of the inequality $e \leq n(n-1)/2$, which holds for any simple graph.]

Depth-first search can be used as the basis for algorithms that solve many different problems. For example, it can be used to find paths and circuits in a graph, it can be used to determine the connected components of a graph, and it can be used to find the cut vertices of a connected graph. As we will see, depth-first search is the basis of backtracking techniques used to search for solutions of computationally difficult problems. (See [GrYe05], [Ma89], and [CoLeRiSt09] for a discussion of algorithms based on depth-first search.)

Breadth-First Search

Demo

We can also produce a spanning tree of a simple graph by the use of **breadth-first search**. Again, a rooted tree will be constructed, and the underlying undirected graph of this rooted tree forms the spanning tree. Arbitrarily choose a root from the vertices of the graph. Then add all
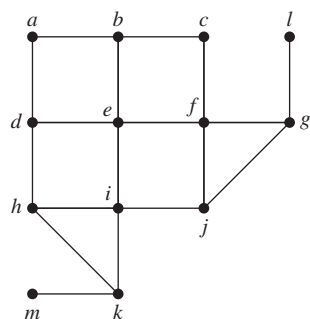
FIGURE 9   A Graph *G*.

edges incident to this vertex. The new vertices added at this stage become the vertices at level 1 in the spanning tree. Arbitrarily order them. Next, for each vertex at level 1, visited in order, add each edge incident to this vertex to the tree as long as it does not produce a simple circuit. Arbitrarily order the children of each vertex at level 1. This produces the vertices at level 2 in the tree. Follow the same procedure until all the vertices in the tree have been added. The procedure ends because there are only a finite number of edges in the graph. A spanning tree is produced because we have produced a tree containing every vertex of the graph. An example of breadth-first search is given in Example 5.

EXAMPLE 5   Use breadth-first search to find a spanning tree for the graph shown in Figure 9.

Solution: The steps of the breadth-first search procedure are shown in Figure 10. We choose the vertex *e* to be the root. Then we add edges incident with all vertices adjacent to *e*, so edges from *e* to *b*, *d*, *f*, and *i* are added. These four vertices are at level 1 in the tree. Next, add the edges from these vertices at level 1 to adjacent vertices not already in the tree. Hence, the edges from *b* to *a* and *c* are added, as are edges from *d* to *h*, from *f* to *j* and *g*, and from *i* to *k*. The new vertices *a*, *c*, *h*, *j*, *g*, and *k* are at level 2. Next, add edges from these vertices to adjacent vertices not already in the graph. This adds edges from *g* to *l* and from *k* to *m*.   ◀

We describe breadth-first search in pseudocode as Algorithm 2. In this algorithm, we assume the vertices of the connected graph *G* are ordered as $v_1, v_2, \ldots, v_n$. In the algorithm we use the term "process" to describe the procedure of adding new vertices, and corresponding edges, to the tree adjacent to the current vertex being processed as long as a simple circuit is not produced.
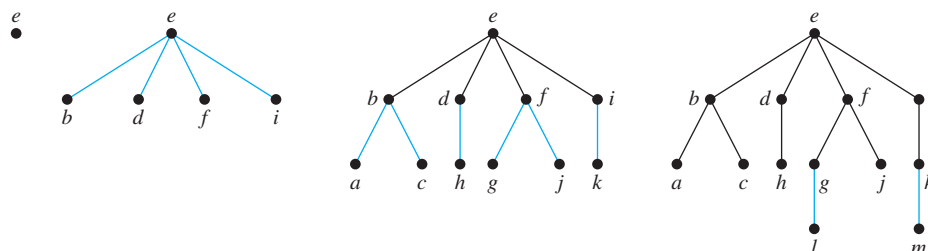
FIGURE 10   **Breadth-First Search of *G*.**

ALGORITHM 2 Breadth-First Search.

procedure *BFS* (*G*: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of vertex v_1
L := empty list
put v_1 in the list L of unprocessed vertices
while L is not empty
   remove the first vertex, v, from L
   for each neighbor w of v
     if w is not in L and not in T **then**
       add w to the end of the list L
       add w and edge $\{v, w\}$ to T

We now analyze the computational complexity of breadth-first search. For each vertex v in the graph we examine all vertices adjacent to v and we add each vertex not yet visited to the tree T. Assuming we have the adjacency lists for the graph available, no computation is required to determine which vertices are adjacent to a given vertex. As in the analysis of the depth-first search algorithm, we see that we examine each edge at most twice to determine whether we should add this edge and its endpoint not already in the tree. It follows that the breadth-first search algorithm uses $O(e)$ or $O(n^2)$ steps.

Breadth-first search is one of the most useful algorithms in graph theory. In particular, it can serve as the basis for algorithms that solve a wide variety of problems. For example, algorithms that find the connected components of a graph, that determine whether a graph is bipartite, and that find the path with the fewest edges between two vertices in a graph can all be built using breadth-first search.

Backtracking Applications

There are problems that can be solved only by performing an exhaustive search of all possible solutions. One way to search systematically for a solution is to use a decision tree, where each internal vertex represents a decision and each leaf a possible solution. To find a solution via backtracking, first make a sequence of decisions in an attempt to reach a solution as long as this is possible. The sequence of decisions can be represented by a path in the decision tree. Once it is known that no solution can result from any further sequence of decisions, backtrack to the parent of the current vertex and work toward a solution with another series of decisions, if this is possible. The procedure continues until a solution is found, or it is established that no solution exists. Examples 6 to 8 illustrate the usefulness of backtracking.

EXAMPLE 6 **Graph Colorings**   How can backtracking be used to decide whether a graph can be colored using n colors?

Solution: We can solve this problem using backtracking in the following way. First pick some vertex a and assign it color 1. Then pick a second vertex b, and if b is not adjacent to a, assign it color 1. Otherwise, assign color 2 to b. Then go on to a third vertex c. Use color 1, if possible, for c. Otherwise use color 2, if this is possible. Only if neither color 1 nor color 2 can be used should color 3 be used. Continue this process as long as it is possible to assign one of the n colors to each additional vertex, always using the first allowable color in the list. If a vertex is reached that cannot be colored by any of the n colors, backtrack to the last assignment made and change the coloring of the last vertex colored, if possible, using the next allowable color in the list. If it is not possible to change this coloring, backtrack farther to previous assignments, one step back at a time, until it is possible to change a coloring of a vertex. Then continue assigning
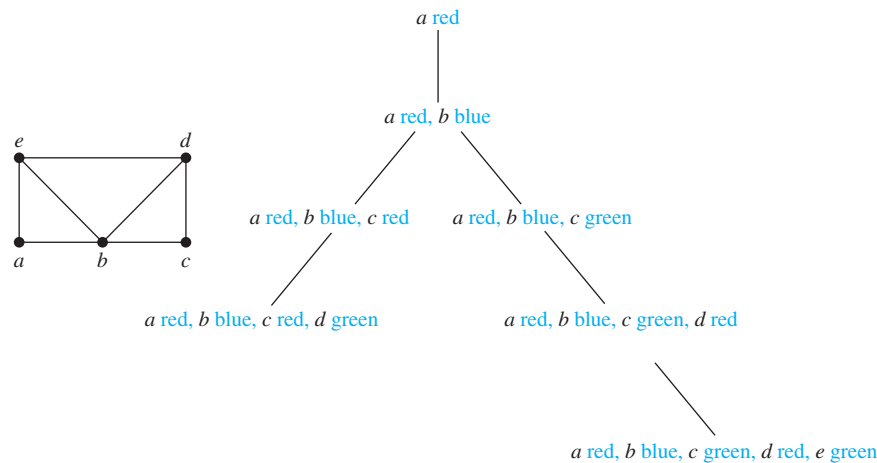
FIGURE 11 **Coloring a Graph Using Backtracking.**

colors of additional vertices as long as possible. If a coloring using n colors exists, backtracking will produce it. (Unfortunately this procedure can be extremely inefficient.)

In particular, consider the problem of coloring the graph shown in Figure 11 with three colors. The tree shown in Figure 11 illustrates how backtracking can be used to construct a 3-coloring. In this procedure, red is used first, then blue, and finally green. This simple example can obviously be done without backtracking, but it is a good illustration of the technique.

In this tree, the initial path from the root, which represents the assignment of red to a, leads to a coloring with a red, b blue, c red, and d green. It is impossible to color e using any of the three colors when a, b, c, and d are colored in this way. So, backtrack to the parent of the vertex representing this coloring. Because no other color can be used for d, backtrack one more level. Then change the color of c to green. We obtain a coloring of the graph by then assigning red to d and green to e. ◀

EXAMPLE 7

Links

The n-Queens Problem The n-queens problem asks how n queens can be placed on an $n \times n$ chessboard so that no two queens can attack one another. How can backtracking be used to solve the n-queens problem?

Solution: To solve this problem we must find n positions on an $n \times n$ chessboard so that no two of these positions are in the same row, same column, or in the same diagonal [a diagonal consists of all positions (i, j) with $i + j = m$ for some m, or $i - j = m$ for some m]. We will use backtracking to solve the n-queens problem. We start with an empty chessboard. At stage $k + 1$ we attempt putting an additional queen on the board in the $(k + 1)$st column, where there are already queens in the first k columns. We examine squares in the $(k + 1)$st column starting with the square in the first row, looking for a position to place this queen so that it is not in the same row or on the same diagonal as a queen already on the board. (We already know it is not in the same column.) If it is impossible to find a position to place the queen in the $(k + 1)$st column, backtrack to the placement of the queen in the kth column, and place this queen in the next allowable row in this column, if such a row exists. If no such row exists, backtrack further.

In particular, Figure 12 displays a backtracking solution to the four-queens problem. In this solution, we place a queen in the first row and column. Then we put a queen in the third row of the second column. However, this makes it impossible to place a queen in the third column. So we backtrack and put a queen in the fourth row of the second column. When we do this, we can place a queen in the second row of the third column. But there is no way to add a queen to the fourth column. This shows that no solution results when a queen is placed in the first row and column. We backtrack to the empty chessboard, and place a queen in the second row of the first column. This leads to a solution as shown in Figure 12. ◀
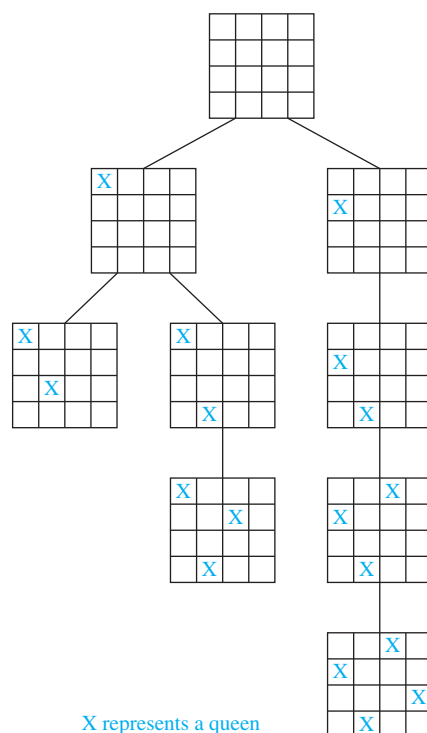
FIGURE 12   A Backtracking Solution of the Four-Queens Problem.

EXAMPLE 8   **Sums of Subsets**   Consider this problem. Given a set of positive integers $x_1, x_2, \ldots, x_n$, find a subset of this set of integers that has M as its sum. How can backtracking be used to solve this problem?

Solution: We start with a sum with no terms. We build up the sum by successively adding terms. An integer in the sequence is included if the sum remains less than M when this integer is added to the sum. If a sum is reached such that the addition of any term is greater than M, backtrack by dropping the last term of the sum.

Figure 13 displays a backtracking solution to the problem of finding a subset of $\{31, 27, 15, 11, 7, 5\}$ with the sum equal to 39.   ◄

FIGURE 13   Find a Sum Equal to 39 Using Backtracking.

FIGURE 14   **Depth-First Search of a Directed Graph.**

Depth-First Search in Directed Graphs

We can easily modify both depth-first search and breadth-first search so that they can run given a directed graph as input. However, the output will not necessarily be a spanning tree, but rather a spanning forest. In both algorithms we can add an edge only when it is directed away from the vertex that is being visited and to a vertex not yet added. If at a stage of either algorithm we find that no edge exists starting at a vertex already added to one not yet added, the next vertex added by the algorithm becomes the root of a new tree in the spanning forest. This is illustrated in Example 9.

EXAMPLE 9   What is the output of depth-first search given the graph G shown in Figure 14(a) as input?

Solution: We begin the depth-first search at vertex a and add vertices b, c, and g and the corresponding edges where we are blocked. We backtrack to c but we are still blocked, and then backtrack to b, where we add vertices f and e and the corresponding edges. Backtracking takes us all the way back to a. We then start a new tree at d and add vertices h, l, k, and j and the corresponding edges. We backtrack to k, then l, then h, and back to d. Finally, we start a new tree at i, completing the depth-first search. The output is shown in Figure 14(b).   ◀

Depth-first search in directed graphs is the basis of many algorithms (see [GrYe05], [Ma89], and [CoLeRiSt09]). It can be used to determine whether a directed graph has a circuit, it can be used to carry out a topological sort of a graph, and it can also be used to find the strongly connected components of a directed graph.

We conclude this section with an application of depth-first search and breadth-first search to search engines on the Web.

EXAMPLE 10   **Web Spiders**   To index websites, search engines such as Google and Yahoo systematically explore the Web starting at known sites. These search engines use programs called Web spiders (or crawlers or bots) to visit websites and analyze their contents. Web spiders use both depth-first searching and breadth-first searching to create indices. As described in Example 5 in Section 10.1, Web pages and links between them can be modeled by a directed graph called the Web graph. Web pages are represented by vertices and links are represented by directed edges. Using depth-first search, an initial Web page is selected, a link is followed to a second Web page (if there is such a link), a link on the second Web page is followed to a third Web page, if there is such a link, and so on, until a page with no new links is found. Backtracking is then used to examine

links at the previous level to look for new links, and so on. (Because of practical limitations, Web spiders have limits to the depth they search in depth-first search.) Using breadth-first search, an initial Web page is selected and a link on this page is followed to a second Web page, then a second link on the initial page is followed (if it exists), and so on, until all links of the initial page have been followed. Then links on the pages one level down are followed, page by page, and so on.   ◀

Exercises

1. How many edges must be removed from a connected graph with n vertices and m edges to produce a spanning tree?

In Exercises 2–6 find a spanning tree for the graph shown by removing edges in simple circuits.

2.

3.

4.

5.

6.
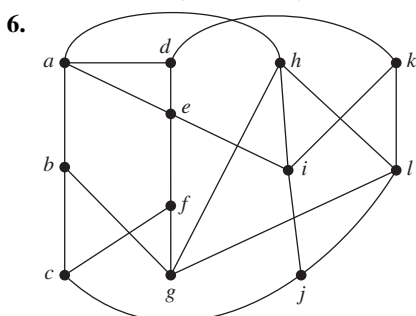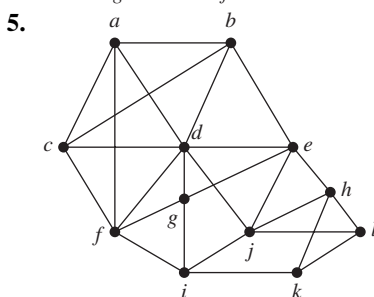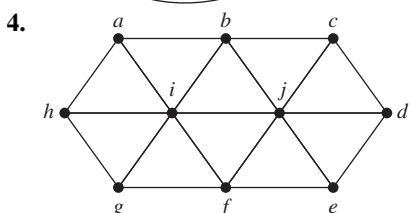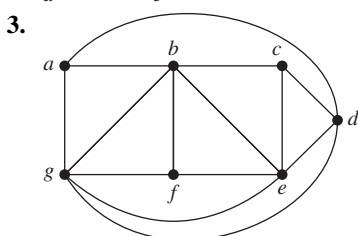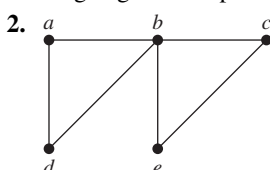
7. Find a spanning tree for each of these graphs.

   a) K_5 **b)** $K_{4,4}$ **c)** $K_{1,6}$

   d) Q_3 **e)** C_5 **f)** W_5

In Exercises 8–10 draw all the spanning trees of the given simple graphs.

8.

9.

10.
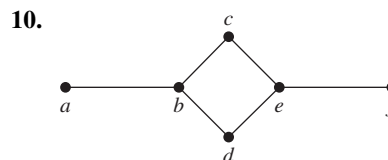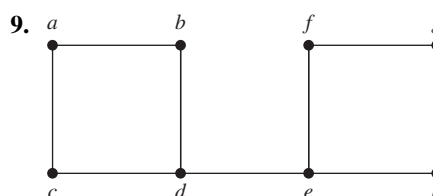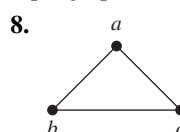
*11. How many different spanning trees does each of these simple graphs have?

   a) K_3 **b)** K_4 **c)** $K_{2,2}$ **d)** C_5

*12. How many nonisomorphic spanning trees does each of these simple graphs have?

   a) K_3 **b)** K_4 **c)** K_5

In Exercises 13–15 use depth-first search to produce a spanning tree for the given simple graph. Choose a as the root of this spanning tree and assume that the vertices are ordered alphabetically.

13.

14.

15.
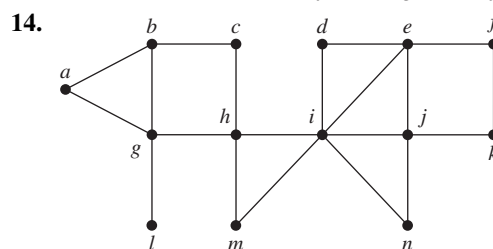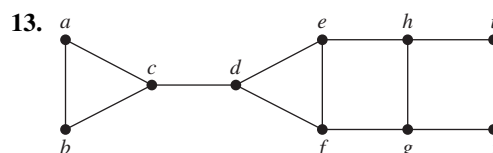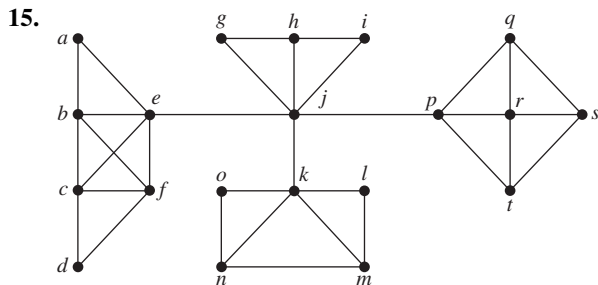
16. Use breadth-first search to produce a spanning tree for each of the simple graphs in Exercises 13–15. Choose a as the root of each spanning tree.

17. Use depth-first search to find a spanning tree of each of these graphs.
  a) W_6 (see Example 7 of Section 10.2), starting at the vertex of degree 6
  b) K_5
  c) $K_{3,4}$, starting at a vertex of degree 3
  d) Q_3

18. Use breadth-first search to find a spanning tree of each of the graphs in Exercise 17.

19. Describe the trees produced by breadth-first search and depth-first search of the wheel graph W_n, starting at the vertex of degree n, where n is an integer with $n \geq 3$. (See Example 7 of Section 10.2.) Justify your answers.

20. Describe the trees produced by breadth-first search and depth-first search of the complete graph K_n, where n is a positive integer. Justify your answers.

21. Describe the trees produced by breadth-first search and depth-first search of the complete bipartite graph $K_{m,n}$, starting at a vertex of degree m, where m and n are positive integers. Justify your answers.

22. Describe the tree produced by breadth-first search and depth-first search for the n-cube graph Q_n, where n is a positive integer.

23. Suppose that an airline must reduce its flight schedule to save money. If its original routes are as illustrated here, which flights can be discontinued to retain service between all pairs of cities (where it may be necessary to combine flights to fly from one city to another)?

24. Explain how breadth-first search or depth-first search can be used to order the vertices of a connected graph.

∗25. Show that the length of the shortest path between vertices v and u in a connected simple graph equals the level number of u in the breadth-first spanning tree of G with root v.

26. Use backtracking to try to find a coloring of each of the graphs in Exercises 7–9 of Section 10.8 using three colors.

27. Use backtracking to solve the n-queens problem for these values of n.
  a) $n = 3$ **b)** $n = 5$ **c)** $n = 6$

28. Use backtracking to find a subset, if it exists, of the set $\{27, 24, 19, 14, 11, 8\}$ with sum
  a) 20.   **b)** 41.   **c)** 60.

29. Explain how backtracking can be used to find a Hamilton path or circuit in a graph.

30. a) Explain how backtracking can be used to find the way out of a maze, given a starting position and the exit position. Consider the maze divided into positions, where at each position the set of available moves includes one to four possibilities (up, down, right, left).
  b) Find a path from the starting position marked by X to the exit in this maze.

A **spanning forest** of a graph G is a forest that contains every vertex of G such that two vertices are in the same tree of the forest when there is a path in G between these two vertices.

31. Show that every finite simple graph has a spanning forest.

32. How many trees are in the spanning forest of a graph?

33. How many edges must be removed to produce the spanning forest of a graph with n vertices, m edges, and c connected components?

34. Let G be a connected graph. Show that if T is a spanning tree of G constructed using breadth-first search, then an edge of G not in T must connect vertices at the same level or at levels that differ by 1 in this spanning tree.

35. Explain how to use breadth-first search to find the length of a shortest path between two vertices in an undirected graph.

36. Devise an algorithm based on breadth-first search that determines whether a graph has a simple circuit, and if so, finds one.

37. Devise an algorithm based on breadth-first search for finding the connected components of a graph.

38. Explain how breadth-first search and how depth-first search can be used to determine whether a graph is bipartite.

39. Which connected simple graphs have exactly one spanning tree?

40. Devise an algorithm for constructing the spanning forest of a graph based on deleting edges that form simple circuits.

41. Devise an algorithm for constructing the spanning forest of a graph based on depth-first searching.

42. Devise an algorithm for constructing the spanning forest of a graph based on breadth-first searching.

43. Let G be a connected graph. Show that if T is a spanning tree of G constructed using depth-first search, then an edge of G not in T must be a back edge, that is, it must connect a vertex to one of its ancestors or one of its descendants in T.

44. When must an edge of a connected simple graph be in every spanning tree for this graph?

45. For which graphs do depth-first search and breadth-first search produce identical spanning trees no matter which vertex is selected as the root of the tree? Justify your answer.

46. Use Exercise 43 to prove that if G is a connected, simple graph with n vertices and G does not contain a simple path of length k then it contains at most $(k-1)n$ edges.

47. Use mathematical induction to prove that breadth-first search visits vertices in order of their level in the resulting spanning tree.

48. Use pseudocode to describe a variation of depth-first search that assigns the integer n to the nth vertex visited in the search. Show that this numbering corresponds to the numbering of the vertices created by a preorder traversal of the spanning tree.

49. Use pseudocode to describe a variation of breadth-first search that assigns the integer m to the mth vertex visited in the search.

∗50. Suppose that G is a directed graph and T is a spanning tree constructed using breadth-first search. Show that every edge of G has endpoints that are at the same level or one level higher or lower.

51. Show that if G is a directed graph and T is a spanning tree constructed using depth-first search, then every edge not in the spanning tree is a **forward edge** connecting an ancestor to a descendant, a **back edge** connecting a descendant to an ancestor, or a **cross edge** connecting a vertex to a vertex in a previously visited subtree.

∗52. Describe a variation of depth-first search that assigns the smallest available positive integer to a vertex when the algorithm is totally finished with this vertex. Show that in this numbering, each vertex has a larger number than its children and that the children have increasing numbers from left to right.

Let T_1 and T_2 be spanning trees of a graph. The **distance** between T_1 and T_2 is the number of edges in T_1 and T_2 that are not common to T_1 and T_2.

53. Find the distance between each pair of spanning trees shown in Figures 3(c) and 4 of the graph G shown in Figure 2.

∗54. Suppose that T_1, T_2, and T_3 are spanning trees of the simple graph G. Show that the distance between T_1 and T_3 does not exceed the sum of the distance between T_1 and T_2 and the distance between T_2 and T_3.

∗∗55. Suppose that T_1 and T_2 are spanning trees of a simple graph G. Moreover, suppose that e_1 is an edge in T_1 that is not in T_2. Show that there is an edge e_2 in T_2 that is not in T_1 such that T_1 remains a spanning tree if e_1 is removed from it and e_2 is added to it, and T_2 remains a spanning tree if e_2 is removed from it and e_1 is added to it.

∗56. Show that it is possible to find a sequence of spanning trees leading from any spanning tree to any other by successively removing one edge and adding another.

A **rooted spanning tree** of a directed graph is a rooted tree containing edges of the graph such that every vertex of the graph is an endpoint of one of the edges in the tree.

57. For each of the directed graphs in Exercises 18–23 of Section 10.5 either find a rooted spanning tree of the graph or determine that no such tree exists.

∗58. Show that a connected directed graph in which each vertex has the same in-degree and out-degree has a rooted spanning tree. [*Hint:* Use an Euler circuit.]

∗59. Give an algorithm to build a rooted spanning tree for connected directed graphs in which each vertex has the same in-degree and out-degree.

∗60. Show that if G is a directed graph and T is a spanning tree constructed using depth-first search, then G contains a circuit if and only if G contains a back edge (see Exercise 51) relative to the spanning tree T.

∗61. Use Exercise 60 to construct an algorithm for determining whether a directed graph contains a circuit.

11.5 Minimum Spanning Trees

Introduction

Links

A company plans to build a communications network connecting its five computer centers. Any pair of these centers can be linked with a leased telephone line. Which links should be made to ensure that there is a path between any two computer centers so that the total cost of the network is minimized? We can model this problem using the weighted graph shown in Figure 1, where vertices represent computer centers, edges represent possible leased lines, and the weights on edges are the monthly lease rates of the lines represented by the edges. We can solve this problem
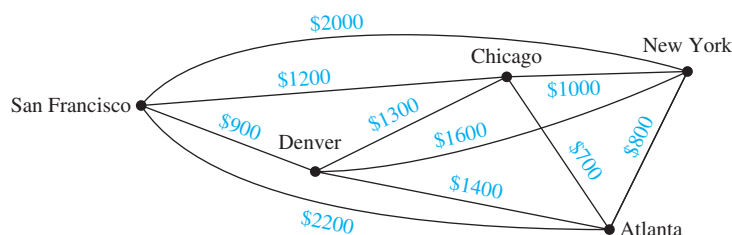
FIGURE 1   A Weighted Graph Showing Monthly Lease Costs for Lines in a Computer Network.

by finding a spanning tree so that the sum of the weights of the edges of the tree is minimized. Such a spanning tree is called a **minimum spanning tree**.

Algorithms for Minimum Spanning Trees

A wide variety of problems are solved by finding a spanning tree in a weighted graph such that the sum of the weights of the edges in the tree is a minimum.

DEFINITION 1

A *minimum spanning tree* in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

We will present two algorithms for constructing minimum spanning trees. Both proceed by successively adding edges of smallest weight from those edges with a specified property that have not already been used. Both are greedy algorithms. Recall from Section 3.1 that a greedy algorithm is a procedure that makes an optimal choice at each of its steps. Optimizing at each step does not guarantee that the optimal overall solution is produced. However, the two algorithms presented in this section for constructing minimum spanning trees are greedy algorithms that do produce optimal solutions.

The first algorithm that we will discuss was originally discovered by the Czech mathematician Vojtěch Jarník in 1930, who described it in a paper in an obscure Czech journal. The algorithm became well known when it was rediscovered in 1957 by Robert Prim. Because of this, it is known as **Prim's algorithm** (and sometimes as the **Prim-Jarník algorithm**). Begin by choosing any edge with smallest weight, putting it into the spanning tree. Successively add to the tree edges of minimum weight that are incident to a vertex already in the tree, never forming a simple circuit with those edges already in the tree. Stop when $n - 1$ edges have been added.

Later in this section, we will prove that this algorithm produces a minimum spanning tree for any connected weighted graph. Algorithm 1 gives a pseudocode description of Prim's algorithm.

ROBERT CLAY PRIM (BORN 1921)   Robert Prim, born in Sweetwater, Texas, received his B.S. in electrical engineering in 1941 and his Ph.D. in mathematics from Princeton University in 1949. He was an engineer at the General Electric Company from 1941 until 1944, an engineer and mathematician at the United States Naval Ordnance Lab from 1944 until 1949, and a research associate at Princeton University from 1948 until 1949. Among the other positions he has held are director of mathematics and mechanics research at Bell Telephone Laboratories from 1958 until 1961 and vice president of research at Sandia Corporation. He is currently retired.

Choice	Edge	Cost
1	{Chicago, Atlanta}	$ 700
2	{Atlanta, New York}	$ 800
3	{Chicago, San Francisco}	$ 1200
4	{San Francisco, Denver}	$ 900
	Total:	$ 3600
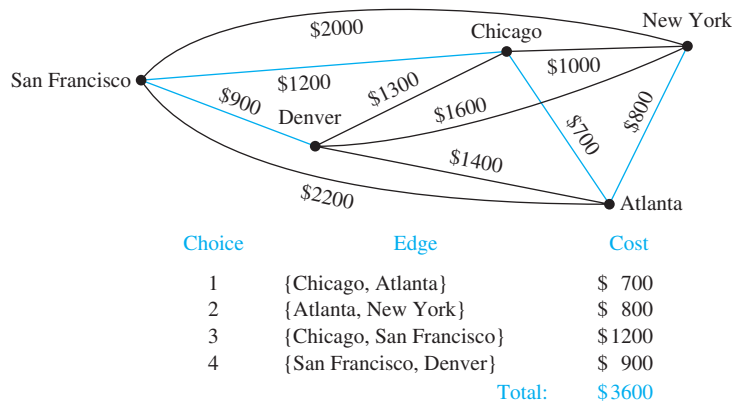
FIGURE 2   A Minimum Spanning Tree for the Weighted Graph in Figure 1.
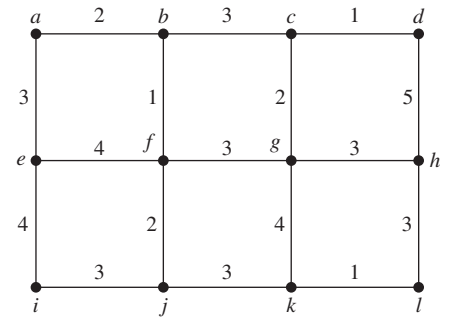
FIGURE 3   A Weighted Graph.

ALGORITHM 1 Prim's Algorithm.

procedure *Prim*(*G*: weighted connected undirected graph with *n* vertices)
T := a minimum-weight edge
for *i* := 1 **to** *n* − 2
   e := an edge of minimum weight incident to a vertex in *T* and not forming a
     simple circuit in *T* if added to *T*
   T := *T* with *e* added
return *T* {*T* is a minimum spanning tree of *G*}

Note that the choice of an edge to add at a stage of the algorithm is not determined when there is more than one edge with the same weight that satisfies the appropriate criteria. We need to order the edges to make the choices deterministic. We will not worry about this in the remainder of the section. Also note that there may be more than one minimum spanning tree for a given connected weighted simple graph. (See Exercise 9.) Examples 1 and 2 illustrate how Prim's algorithm is used.

EXAMPLE 1   Use Prim's algorithm to design a minimum-cost communications network connecting all the computers represented by the graph in Figure 1.

Solution: We solve this problem by finding a minimum spanning tree in the graph in Figure 1. Prim's algorithm is carried out by choosing an initial edge of minimum weight and successively adding edges of minimum weight that are incident to a vertex in the tree and that do not form simple circuits. The edges in color in Figure 2 show a minimum spanning tree produced by Prim's algorithm, with the choice made at each step displayed. ◄

EXAMPLE 2   Use Prim's algorithm to find a minimum spanning tree in the graph shown in Figure 3.

Solution: A minimum spanning tree constructed using Prim's algorithm is shown in Figure 4. The successive edges chosen are displayed. ◄

Links   The second algorithm we will discuss was discovered by Joseph Kruskal in 1956, although the basic ideas it uses were described much earlier. To carry out **Kruskal's algorithm**, choose an edge in the graph with minimum weight.

Choice	Edge	Weight
1	$\{b, f\}$	1
2	$\{a, b\}$	2
3	$\{f, j\}$	2
4	$\{a, e\}$	3
5	$\{i, j\}$	3
6	$\{f, g\}$	3
7	$\{c, g\}$	2
8	$\{c, d\}$	1
9	$\{g, h\}$	3
10	$\{h, l\}$	3
11	$\{k, l\}$	1
	Total:	24

(a)   (b)
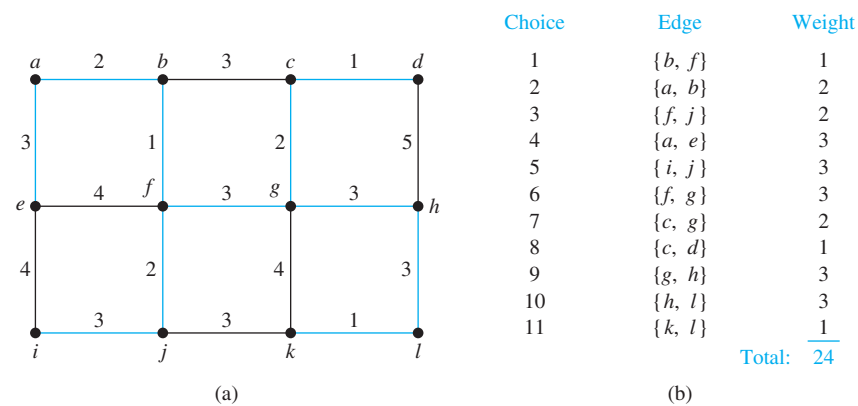
FIGURE 4 **A Minimum Spanning Tree Produced Using Prim's Algorithm.**

Successively add edges with minimum weight that do not form a simple circuit with those edges already chosen. Stop after $n - 1$ edges have been selected.

The proof that Kruskal's algorithm produces a minimum spanning tree for every connected weighted graph is left as an exercise. Pseudocode for Kruskal's algorithm is given in Algorithm 2.

ALGORITHM 2 Kruskal's Algorithm.

procedure *Kruskal*(*G*: weighted connected undirected graph with n vertices)
$T :=$ empty graph
for $i := 1$ **to** $n - 1$
    $e :=$ any edge in G with smallest weight that does not form a simple circuit
        when added to T
    $T := T$ with e added
return T {T is a minimum spanning tree of G}

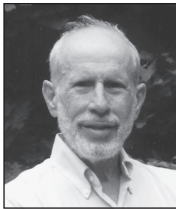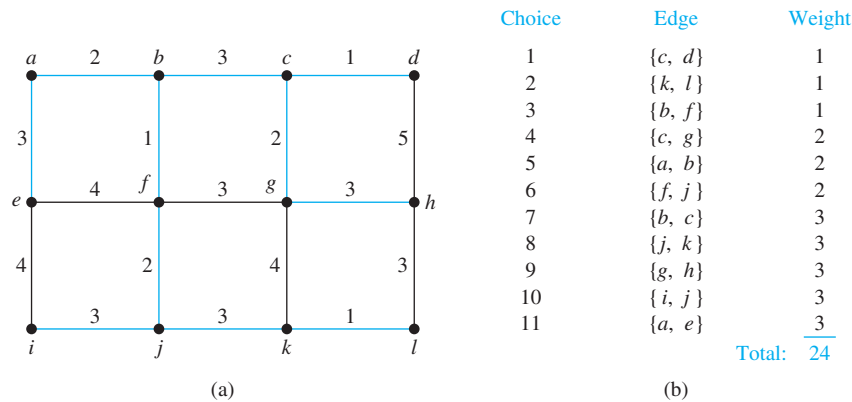**JOSEPH BERNARD KRUSKAL (1928–2010)** Joseph Kruskal was born in New York City, where his father was a fur dealer and his mother promoted the art of origami on early television. Kruskal attended the University of Chicago and received his Ph.D. from Princeton University in 1954. He was an instructor in mathematics at Princeton and at the University of Wisconsin, and later he was an assistant professor at the University of Michigan. In 1959 he became a member of the technical staff at Bell Laboratories, where he worked until his retirement in the late 1990s. Kruskal discovered his algorithm for producing minimum spanning trees when he was a second-year graduate student. He was not sure his $2\frac{1}{2}$-page paper on this subject was worthy of publication, but was convinced by others to submit it. His research interests included statistical linguistics and psychometrics. Besides his work on minimum spanning trees, Kruskal is also known for contributions to multidimensional scaling. It is noteworthy that Joseph Kruskal's two brothers, Martin and William, also were well known mathematicians.

HISTORICAL NOTE Joseph Kruskal and Robert Prim developed their algorithms for constructing minimum spanning trees in the mid-1950s. However, they were not the first people to discover such algorithms. For example, the work of the anthropologist Jan Czekanowski, in 1909, contains many of the ideas required to find minimum spanning trees. In 1926, Otakar Boruvka described methods for constructing minimum spanning trees in work relating to the construction of electric power networks, and as mentioned in the text what is now called Prim's algorithm was discovered by Vojtěch Jarník in 1930.

Choice	Edge	Weight
1	$\{c, d\}$	1
2	$\{k, l\}$	1
3	$\{b, f\}$	1
4	$\{c, g\}$	2
5	$\{a, b\}$	2
6	$\{f, j\}$	2
7	$\{b, c\}$	3
8	$\{j, k\}$	3
9	$\{g, h\}$	3
10	$\{i, j\}$	3
11	$\{a, e\}$	3
	Total:	24

(a)   (b)

FIGURE 5   **A Minimum Spanning Tree Produced by Kruskal's Algorithm.**

The reader should note the difference between Prim's and Kruskal's algorithms. In Prim's algorithm edges of minimum weight that are incident to a vertex already in the tree, and not forming a circuit, are chosen; whereas in Kruskal's algorithm edges of minimum weight that are not necessarily incident to a vertex already in the tree, and that do not form a circuit, are chosen. Note that as in Prim's algorithm, if the edges are not ordered, there may be more than one choice for the edge to add at a stage of this procedure. Consequently, the edges need to be ordered for the procedure to be deterministic. Example 3 illustrates how Kruskal's algorithm is used.

EXAMPLE 3   Use Kruskal's algorithm to find a minimum spanning tree in the weighted graph shown in Figure 3.

Solution: A minimum spanning tree and the choices of edges at each stage of Kruskal's algorithm are shown in Figure 5. ◄

We will now prove that Prim's algorithm produces a minimum spanning tree of a connected weighted graph.

Proof: Let G be a connected weighted graph. Suppose that the successive edges chosen by Prim's algorithm are $e_1, e_2, \ldots, e_{n-1}$. Let S be the tree with $e_1, e_2, \ldots, e_{n-1}$ as its edges, and let S_k be the tree with $e_1, e_2, \ldots, e_k$ as its edges. Let T be a minimum spanning tree of G containing the edges $e_1, e_2, \ldots, e_k$, where k is the maximum integer with the property that a minimum spanning tree exists containing the first k edges chosen by Prim's algorithm. The theorem follows if we can show that $S = T$.

Suppose that $S \neq T$, so that $k < n - 1$. Consequently, T contains $e_1, e_2, \ldots, e_k$, but not e_{k+1}. Consider the graph made up of T together with e_{k+1}. Because this graph is connected and has n edges, too many edges to be a tree, it must contain a simple circuit. This simple circuit must contain e_{k+1} because there was no simple circuit in T. Furthermore, there must be an edge in the simple circuit that does not belong to S_{k+1} because S_{k+1} is a tree. By starting at an endpoint of e_{k+1} that is also an endpoint of one of the edges $e_1, \ldots, e_k$, and following the circuit until it reaches an edge not in S_{k+1}, we can find an edge e not in S_{k+1} that has an endpoint that is also an endpoint of one of the edges $e_1, e_2, \ldots, e_k$.

By deleting e from T and adding e_{k+1}, we obtain a tree T' with $n - 1$ edges (it is a tree because it has no simple circuits). Note that the tree T' contains $e_1, e_2, \ldots, e_k, e_{k+1}$. Furthermore, because e_{k+1} was chosen by Prim's algorithm at the $(k + 1)$st step, and e was also available at that step, the weight of e_{k+1} is less than or equal to the weight of e. From this observation, it follows that T' is also a minimum spanning tree, because the sum of the weights of its edges

does not exceed the sum of the weights of the edges of T. This contradicts the choice of k as the maximum integer such that a minimum spanning tree exists containing $e_1, \ldots, e_k$. Hence, $k = n - 1$, and $S = T$. It follows that Prim's algorithm produces a minimum spanning tree. ◁

It can be shown (see [CoLeRiSt09]) that to find a minimum spanning tree of a graph with m edges and n vertices, Kruskal's algorithm can be carried out using $O(m \log m)$ operations and Prim's algorithm can be carried out using $O(m \log n)$ operations. Consequently, it is preferable to use Kruskal's algorithm for graphs that are **sparse**, that is, where m is very small compared to $C(n, 2) = n(n - 1)/2$, the total number of possible edges in an undirected graph with n vertices. Otherwise, there is little difference in the complexity of these two algorithms.
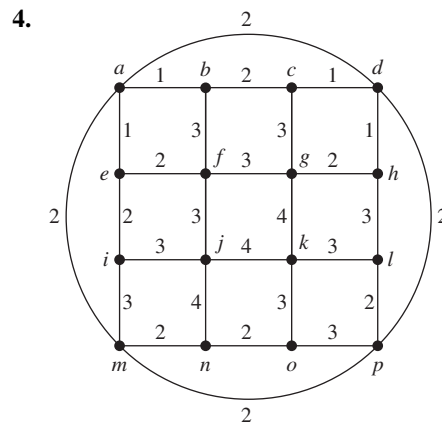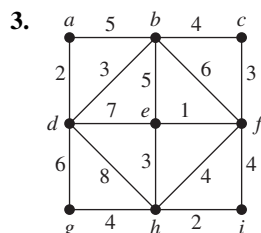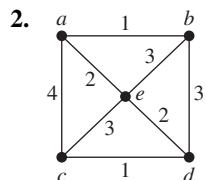
Exercises

1. The roads represented by this graph are all unpaved. The lengths of the roads between pairs of towns are represented by edge weights. Which roads should be paved so that there is a path of paved roads between each pair of towns so that a minimum road length is paved? (*Note:* These towns are in Nevada.)
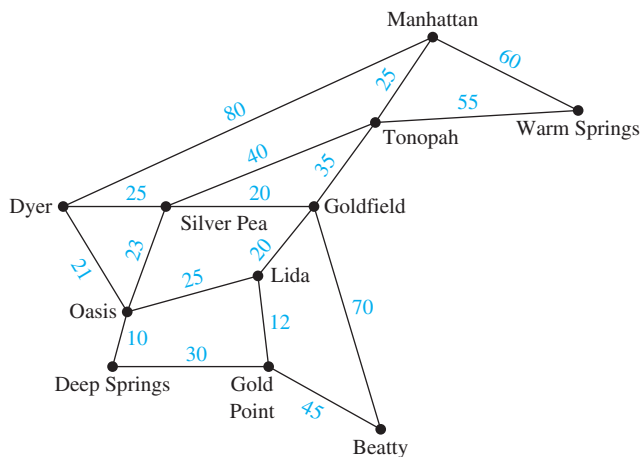
In Exercises 2–4 use Prim's algorithm to find a minimum spanning tree for the given weighted graph.

2.

3.

4.

5. Use Kruskal's algorithm to design the communications network described at the beginning of the section.

6. Use Kruskal's algorithm to find a minimum spanning tree for the weighted graph in Exercise 2.

7. Use Kruskal's algorithm to find a minimum spanning tree for the weighted graph in Exercise 3.

8. Use Kruskal's algorithm to find a minimum spanning tree for the weighted graph in Exercise 4.

9. Find a connected weighted simple graph with the fewest edges possible that has more than one minimum spanning tree.

10. A **minimum spanning forest** in a weighted graph is a spanning forest with minimal weight. Explain how Prim's and Kruskal's algorithms can be adapted to construct minimum spanning forests.

A **maximum spanning tree** of a connected weighted undirected graph is a spanning tree with the largest possible weight.

11. Devise an algorithm similar to Prim's algorithm for constructing a maximum spanning tree of a connected weighted graph.

12. Devise an algorithm similar to Kruskal's algorithm for constructing a maximum spanning tree of a connected weighted graph.

13. Find a maximum spanning tree for the weighted graph in Exercise 2.

14. Find a maximum spanning tree for the weighted graph in Exercise 3.

15. Find a maximum spanning tree for the weighted graph in Exercise 4.

16. Find the second least expensive communications network connecting the five computer centers in the problem posed at the beginning of the section.

*17. Devise an algorithm for finding the second shortest spanning tree in a connected weighted graph.

*18. Show that an edge with smallest weight in a connected weighted graph must be part of any minimum spanning tree.

19. Show that there is a unique minimum spanning tree in a connected weighted graph if the weights of the edges are all different.

20. Suppose that the computer network connecting the cities in Figure 1 must contain a direct link between New York and Denver. What other links should be included so that there is a link between every two computer centers and the cost is minimized?

21. Find a spanning tree with minimal total weight containing the edges $\{e, i\}$ and $\{g, k\}$ in the weighted graph in Figure 3.

22. Describe an algorithm for finding a spanning tree with minimal weight containing a specified set of edges in a connected weighted undirected simple graph.

23. Express the algorithm devised in Exercise 22 in pseudocode.

Sollin's algorithm produces a minimum spanning tree from a connected weighted simple graph $G = (V, E)$ by successively adding groups of edges. Suppose that the vertices in V are ordered. This produces an ordering of the edges where $\{u_0, v_0\}$ precedes $\{u_1, v_1\}$ if u_0 precedes u_1 or if $u_0 = u_1$ and v_0 precedes v_1. The algorithm begins by simultaneously choosing the edge of least weight incident to each vertex. The first edge in the ordering is taken in the case of ties. This produces a graph with no simple circuits, that is, a forest of trees (Exercise 24 asks for a proof of this fact). Next, simultaneously choose for each tree in the forest the shortest edge between a vertex in this tree and a vertex in a different tree. Again the first edge in the ordering is chosen in the case of ties. (This produces a graph with no simple circuits containing fewer trees than were present before this step; see Exercise 24.) Continue the process of simultaneously adding edges connecting trees until $n - 1$ edges have been chosen. At this stage a minimum spanning tree has been constructed.

*24. Show that the addition of edges at each stage of Sollin's algorithm produces a forest.

25. Use Sollin's algorithm to produce a minimum spanning tree for the weighted graph shown in
a) Figure 1.
b) Figure 3.

*26. Express Sollin's algorithm in pseudocode.

**27. Prove that Sollin's algorithm produces a minimum spanning tree in a connected undirected weighted graph.

*28. Show that the first step of Sollin's algorithm produces a forest containing at least $\lceil n/2 \rceil$ edges when the input is an undirected graph with n vertices.

*29. Show that if there are r trees in the forest at some intermediate step of Sollin's algorithm, then at least $\lceil r/2 \rceil$ edges are added by the next iteration of the algorithm.

*30. Show that when given as input an undirected graph with n vertices, no more than $\lfloor n/2^k \rfloor$ trees remain after the first step of Sollin's algorithm has been carried out and the second step of the algorithm has been carried out $k - 1$ times.

*31. Show that Sollin's algorithm requires at most $\log n$ iterations to produce a minimum spanning tree from a connected undirected weighted graph with n vertices.

32. Prove that Kruskal's algorithm produces minimum spanning trees.

33. Show that if G is a weighted graph with distinct edge weights, then for every simple circuit of G, the edge of maximum weight in this circuit does not belong to any minimum spanning tree of G.

When Kruskal invented the algorithm that finds minimum spanning trees by adding edges in order of increasing weight as long as they do not form a simple circuit, he also invented another algorithm sometimes called the **reverse-delete algorithm**. This algorithm proceeds by successively deleting edges of maximum weight from a connected graph as long as doing so does not disconnect the graph.

34. Express the reverse-delete algorithm in pseudocode.

35. Prove that the reverse-delete algorithm always produces a minimum spanning tree when given as input a weighted graph with distinct edge weights. [*Hint:* Use Exercise 33.]

Key Terms and Results

TERMS

tree: a connected undirected graph with no simple circuits

forest: an undirected graph with no simple circuits

rooted tree: a directed graph with a specified vertex, called the root, such that there is a unique path to every other vertex from this root

subtree: a subgraph of a tree that is also a tree

parent of v in a rooted tree: the vertex u such that (u, v) is an edge of the rooted tree

child of a vertex v in a rooted tree: any vertex with v as its parent

sibling of a vertex v in a rooted tree: a vertex with the same parent as v

ancestor of a vertex v in a rooted tree: any vertex on the path from the root to v

descendant of a vertex *v* in a rooted tree: any vertex that has *v* as an ancestor

internal vertex: a vertex that has children

leaf: a vertex with no children

level of a vertex: the length of the path from the root to this vertex

height of a tree: the largest level of the vertices of a tree

m-ary tree: a tree with the property that every internal vertex has no more than *m* children

full *m*-ary tree: a tree with the property that every internal vertex has exactly *m* children

binary tree: an *m*-ary tree with $m = 2$ (each child may be designated as a left or a right child of its parent)

ordered tree: a tree in which the children of each internal vertex are linearly ordered

balanced tree: a tree in which every leaf is at level *h* or $h - 1$, where *h* is the height of the tree

binary search tree: a binary tree in which the vertices are labeled with items so that a label of a vertex is greater than the labels of all vertices in the left subtree of this vertex and is less than the labels of all vertices in the right subtree of this vertex

decision tree: a rooted tree where each vertex represents a possible outcome of a decision and the leaves represent the possible solutions of a problem

game tree: a rooted tree where vertices represents the possible positions of a game as it progresses and edges represent legal moves between these positions

prefix code: a code that has the property that the code of a character is never a prefix of the code of another character

minmax strategy: the strategy where the first player and second player move to positions represented by a child with maximum and minimum value, respectively

value of a vertex in a game tree: for a leaf, the payoff to the first player when the game terminates in the position represented by this leaf; for an internal vertex, the maximum or minimum of the values of its children, for an internal vertex at an even or odd level, respectively

tree traversal: a listing of the vertices of a tree

preorder traversal: a listing of the vertices of an ordered rooted tree defined recursively—the root is listed, followed by the first subtree, followed by the other subtrees in the order they occur from left to right

inorder traversal: a listing of the vertices of an ordered rooted tree defined recursively—the first subtree is listed, followed by the root, followed by the other subtrees in the order they occur from left to right

postorder traversal: a listing of the vertices of an ordered rooted tree defined recursively—the subtrees are listed in the order they occur from left to right, followed by the root

infix notation: the form of an expression (including a full set of parentheses) obtained from an inorder traversal of the binary tree representing this expression

prefix (or Polish) notation: the form of an expression obtained from a preorder traversal of the tree representing this expression

postfix (or reverse Polish) notation: the form of an expression obtained from a postorder traversal of the tree representing this expression

spanning tree: a tree containing all vertices of a graph

minimum spanning tree: a spanning tree with smallest possible sum of weights of its edges

RESULTS

A graph is a tree if and only if there is a unique simple path between every pair of its vertices.

A tree with *n* vertices has $n - 1$ edges.

A full *m*-ary tree with *i* internal vertices has $mi + 1$ vertices.

The relationships among the numbers of vertices, leaves, and internal vertices in a full *m*-ary tree (see Theorem 4 in Section 11.1)

There are at most m^h leaves in an *m*-ary tree of height *h*.

If an *m*-ary tree has *l* leaves, its height *h* is at least $\lceil \log_m l \rceil$. If the tree is also full and balanced, then its height is $\lceil \log_m l \rceil$.

Huffman coding: a procedure for constructing an optimal binary code for a set of symbols, given the frequencies of these symbols

depth-first search, or backtracking: a procedure for constructing a spanning tree by adding edges that form a path until this is not possible, and then moving back up the path until a vertex is found where a new path can be formed

breadth-first search: a procedure for constructing a spanning tree that successively adds all edges incident to the last set of edges added, unless a simple circuit is formed

Prim's algorithm: a procedure for producing a minimum spanning tree in a weighted graph that successively adds edges with minimal weight among all edges incident to a vertex already in the tree so that no edge produces a simple circuit when it is added

Kruskal's algorithm: a procedure for producing a minimum spanning tree in a weighted graph that successively adds edges of least weight that are not already in the tree such that no edge produces a simple circuit when it is added

Review Questions

1. a) Define a tree.   **b)** Define a forest.

2. Can there be two different simple paths between the vertices of a tree?

3. Give at least three examples of how trees are used in modeling.

4. a) Define a rooted tree and the root of such a tree.

b) Define the parent of a vertex and a child of a vertex in a rooted tree.

c) What are an internal vertex, a leaf, and a subtree in a rooted tree?

d) Draw a rooted tree with at least 10 vertices, where the degree of each vertex does not exceed 3. Identify the

root, the parent of each vertex, the children of each vertex, the internal vertices, and the leaves.

5. a) How many edges does a tree with n vertices have?
  b) What do you need to know to determine the number of edges in a forest with n vertices?

6. a) Define a full m-ary tree.
  b) How many vertices does a full m-ary tree have if it has i internal vertices? How many leaves does the tree have?

7. a) What is the height of a rooted tree?
  b) What is a balanced tree?
  c) How many leaves can an m-ary tree of height h have?

8. a) What is a binary search tree?
  b) Describe an algorithm for constructing a binary search tree.
  c) Form a binary search tree for the words *vireo, warbler, egret, grosbeak, nuthatch*, and *kingfisher*.

9. a) What is a prefix code?
  b) How can a prefix code be represented by a binary tree?

10. a) Define preorder, inorder, and postorder tree traversal.
  b) Give an example of preorder, postorder, and inorder traversal of a binary tree of your choice with at least 12 vertices.

11. a) Explain how to use preorder, inorder, and postorder traversals to find the prefix, infix, and postfix forms of an arithmetic expression.
  b) Draw the ordered rooted tree that represents $((x - 3) + ((x/4) + (x - y) \uparrow 3))$.
  c) Find the prefix and postfix forms of the expression in part (b).

12. Show that the number of comparisons used by a sorting algorithm to sort a list of n elements is at least $\lceil \log n! \rceil$.

13. a) Describe the Huffman coding algorithm for constructing an optimal code for a set of symbols, given the frequency of these symbols.

b) Use Huffman coding to find an optimal code for these symbols and frequencies: A: 0.2, B: 0.1, C: 0.3, D: 0.4.

14. Draw the game tree for nim if the starting position consists of two piles with one and four stones, respectively. Who wins the game if both players follow an optimal strategy?

15. a) What is a spanning tree of a simple graph?
  b) Which simple graphs have spanning trees?
  c) Describe at least two different applications that require that a spanning tree of a simple graph be found.

16. a) Describe two different algorithms for finding a spanning tree in a simple graph.
  b) Illustrate how the two algorithms you described in part (a) can be used to find the spanning tree of a simple graph, using a graph of your choice with at least eight vertices and 15 edges.

17. a) Explain how backtracking can be used to determine whether a simple graph can be colored using n colors.
  b) Show, with an example, how backtracking can be used to show that a graph with a chromatic number equal to 4 cannot be colored with three colors, but can be colored with four colors.

18. a) What is a minimum spanning tree of a connected weighted graph?
  b) Describe at least two different applications that require that a minimum spanning tree of a connected weighted graph be found.

19. a) Describe Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees.
  b) Illustrate how Kruskal's algorithm and Prim's algorithm are used to find a minimum spanning tree, using a weighted graph with at least eight vertices and 15 edges.

Supplementary Exercises

*1. Show that a simple graph is a tree if and only if it contains no simple circuits and the addition of an edge connecting two nonadjacent vertices produces a new graph that has exactly one simple circuit (where circuits that contain the same edges are not considered different).

*2. How many nonisomorphic rooted trees are there with six vertices?

3. Show that every tree with at least one edge must have at least two pendant vertices.

4. Show that a tree with n vertices that has $n - 1$ pendant vertices must be isomorphic to $K_{1,n-1}$.

5. What is the sum of the degrees of the vertices of a tree with n vertices?

*6. Suppose that $d_1, d_2, \ldots, d_n$ are n positive integers with sum $2n - 2$. Show that there is a tree that has n vertices such that the degrees of these vertices are $d_1, d_2, \ldots, d_n$.

7. Show that every tree is a planar graph.

8. Show that every tree is bipartite.

9. Show that every forest can be colored using two colors.

A **B-tree of degree k** is a rooted tree such that all its leaves are at the same level, its root has at least two and at most k children unless it is a leaf, and every internal vertex other than the root has at least $\lceil k/2 \rceil$, but no more than k, children. Computer files can be accessed efficiently when B-trees are used to represent them.

10. Draw three different B-trees of degree 3 with height 4.

*11. Give an upper bound and a lower bound for the number of leaves in a B-tree of degree k with height h.

*12. Give an upper bound and a lower bound for the height of a B-tree of degree k with n leaves.

The **binomial trees** B_i, $i = 0, 1, 2, \ldots$, are ordered rooted trees defined recursively:

Basis step: The binomial tree B_0 is the tree with a single vertex.

Recursive step: Let k be a nonnegative integer. To construct the binomial tree B_{k+1}, add a copy of B_k to a second copy of B_k by adding an edge that makes the root of the first copy of B_k the leftmost child of the root of the second copy of B_k.

13. Draw B_k for $k = 0, 1, 2, 3, 4$.

14. How many vertices does B_k have? Prove that your answer is correct.

15. Find the height of B_k. Prove that your answer is correct.

16. How many vertices are there in B_k at depth j, where $0 \le j \le k$? Justify your answer.

17. What is the degree of the root of B_k? Prove that your answer is correct.

18. Show that the vertex of largest degree in B_k is the root.

A rooted tree T is called an S_k**-tree** if it satisfies this recursive definition. It is an S_0-tree if it has one vertex. For $k > 0$, T is an S_k-tree if it can be built from two S_{k-1}-trees by making the root of one the root of the S_k-tree and making the root of the other the child of the root of the first S_{k-1}-tree.

19. Draw an S_k-tree for $k = 0, 1, 2, 3, 4$.

20. Show that an S_k-tree has 2^k vertices and a unique vertex at level k. This vertex at level k is called the **handle**.

∗21. Suppose that T is an S_k-tree with handle v. Show that T can be obtained from disjoint trees $T_0, T_1, \ldots, T_{k-1}$, with roots $r_0, r_1, \ldots, r_{k-1}$, respectively, where v is not in any of these trees, where T_i is an S_i-tree for $i = 0, 1, \ldots, k - 1$, by connecting v to r_0 and r_i to r_{i+1} for $i = 0, 1, \ldots, k - 2$.

The listing of the vertices of an ordered rooted tree in **level order** begins with the root, followed by the vertices at level 1 from left to right, followed by the vertices at level 2 from left to right, and so on.

22. List the vertices of the ordered rooted trees in Figures 3 and 9 of Section 11.3 in level order.

23. Devise an algorithm for listing the vertices of an ordered rooted tree in level order.

∗24. Devise an algorithm for determining if a set of universal addresses can be the addresses of the leaves of a rooted tree.

25. Devise an algorithm for constructing a rooted tree from the universal addresses of its leaves.

A **cut set** of a graph is a set of edges such that the removal of these edges produces a subgraph with more connected components than in the original graph, but no proper subset of this set of edges has this property.

26. Show that a cut set of a graph must have at least one edge in common with any spanning tree of this graph.

A **cactus** is a connected graph in which no edge is in more than one simple circuit not passing through any vertex other than its initial vertex more than once or its initial vertex other than at its terminal vertex (where two circuits that contain the same edges are not considered different).

27. Which of these graphs are cacti?

a)

b)

c)
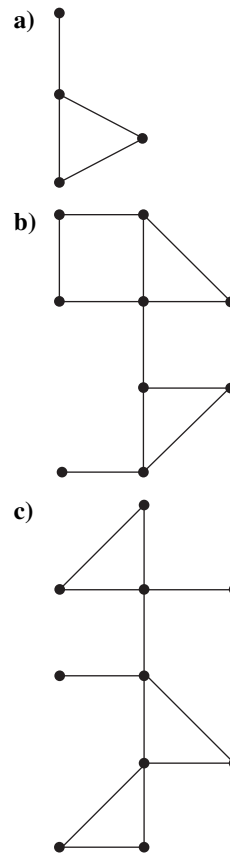
28. Is a tree necessarily a cactus?

29. Show that a cactus is formed if we add a circuit containing new edges beginning and ending at a vertex of a tree.

∗30. Show that if every circuit not passing through any vertex other than its initial vertex more than once in a connected graph contains an odd number of edges, then this graph must be a cactus.
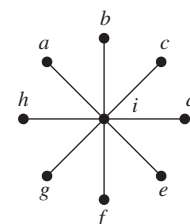
A **degree-constrained spanning tree** of a simple graph G is a spanning tree with the property that the degree of a vertex in this tree cannot exceed some specified bound. Degree-constrained spanning trees are useful in models of transportation systems where the number of roads at an intersection is limited, models of communications networks where the number of links entering a node is limited, and so on.

In Exercises 31–33 find a degree-constrained spanning tree of the given graph where each vertex has degree less than or equal to 3, or show that such a spanning tree does not exist.
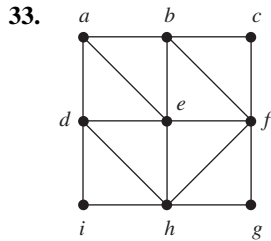
31.

32.

33.
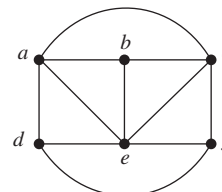
34. Show that a degree-constrained spanning tree of a simple graph in which each vertex has degree not exceeding 2 consists of a single Hamilton path in the graph.

35. A tree with n vertices is called **graceful** if its vertices can be labeled with the integers $1, 2, \ldots, n$ such that the absolute values of the difference of the labels of adjacent vertices are all different. Show that these trees are graceful.

a)

b)

c)

d)
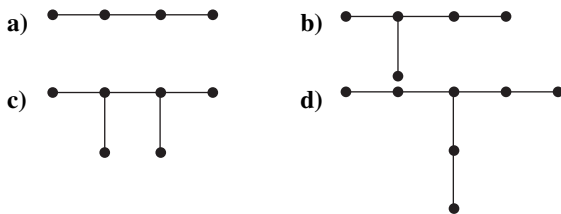
A **caterpillar** is a tree that contains a simple path such that every vertex not contained in this path is adjacent to a vertex in the path.

36. Which of the graphs in Exercise 35 are caterpillars?

37. How many nonisomorphic caterpillars are there with six vertices?

38. **a) Prove or disprove that all trees whose edges form a single path are graceful.
  b) Prove or disprove that all caterpillars are graceful.

39. Suppose that in a long bit string the frequency of occurrence of a 0 bit is 0.9 and the frequency of a 1 bit is 0.1 and bits occur independently.

  a) Construct a Huffman code for the four blocks of two bits, 00, 01, 10, and 11. What is the average number of bits required to encode a bit string using this code?
  b) Construct a Huffman code for the eight blocks of three bits. What is the average number of bits required to encode a bit string using this code?

40. Suppose that G is a directed graph with no circuits. Describe how depth-first search can be used to carry out a topological sort of the vertices of G.

*41. Suppose that e is an edge in a weighted graph that is incident to a vertex v such that the weight of e does not exceed the weight of any other edge incident to v. Show that there exists a minimum spanning tree containing this edge.

42. Three couples arrive at the bank of a river. Each of the wives is jealous and does not trust her husband when he is with one of the other wives (and perhaps with other people), but not with her. How can six people cross to the other side of the river using a boat that can hold no more than two people so that no husband is alone with a woman other than his wife? Use a graph theory model.

*43. Show that if no two edges in a weighted graph have the same weight, then the edge with least weight incident to a vertex v is included in every minimum spanning tree.

44. Find a minimum spanning tree of each of these graphs where the degree of each vertex in the spanning tree does not exceed 2.

a)

b)
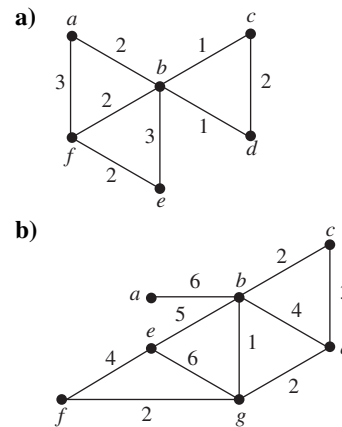
Let $G = (V, E)$ be a directed graph and let r be a vertex in G. An **arborescence** of G rooted at r is a subgraph $T = (V, F)$ of G such that the underlying undirected graph of T is a spanning tree of the underlying undirected graph of G and for every vertex $v \in V$ there is a path from r to v in T (with directions taken into account).

45. Show that a subgraph $T = (V, F)$ of the graph $G = (V, E)$ is an arborescence of G rooted at r if and only if T contains r, T has no simple circuits, and for every vertex $v \in V$ other than r, $\deg^-(v) = 1$ in T.

46. Show that a directed graph $G = (V, E)$ has an arborescence rooted at the vertex r if and only if for every vertex $v \in V$, there is a directed path from r to v.

47. In this exercise we will develop an algorithm to find the strong components of a directed graph $G = (V, E)$. Recall that a vertex $w \in V$ is **reachable** from a vertex $v \in V$ if there is a directed path from v to w.

  a) Explain how to use breadth-first search in the directed graph G to find all the vertices reachable from a vertex $v \in G$.
  b) Explain how to use breadth-first search in G^{conv} to find all the vertices from which a vertex $v \in G$ is reachable. (Recall that G^{conv} is the directed graph obtained from G by reversing the direction of all its edges.)
  c) Explain how to use parts (a) and (b) to construct an algorithm that finds the strong components of a directed graph G, and explain why your algorithm is correct.

Computer Projects

Write programs with these input and output.

1. Given the adjacency matrix of an undirected simple graph, determine whether the graph is a tree.

2. Given the adjacency matrix of a rooted tree and a vertex in the tree, find the parent, children, ancestors, descendants, and level of this vertex.

3. Given the list of edges of a rooted tree and a vertex in the tree, find the parent, children, ancestors, descendants, and level of this vertex.

4. Given a list of items, construct a binary search tree containing these items.

5. Given a binary search tree and an item, locate or add this item to the binary search tree.

6. Given the ordered list of edges of an ordered rooted tree, find the universal addresses of its vertices.

7. Given the ordered list of edges of an ordered rooted tree, list its vertices in preorder, inorder, and postorder.

8. Given an arithmetic expression in prefix form, find its value.

9. Given an arithmetic expression in postfix form, find its value.

10. Given the frequency of symbols, use Huffman coding to find an optimal code for these symbols.

11. Given an initial position in the game of nim, determine an optimal strategy for the first player.

12. Given the adjacency matrix of a connected undirected simple graph, find a spanning tree for this graph using depth-first search.

13. Given the adjacency matrix of a connected undirected simple graph, find a spanning tree for this graph using breadth-first search.

14. Given a set of positive integers and a positive integer N, use backtracking to find a subset of these integers that have N as their sum.

*15. Given the adjacency matrix of an undirected simple graph, use backtracking to color the graph with three colors, if this is possible.

*16. Given a positive integer n, solve the n-queens problem using backtracking.

17. Given the list of edges and their weights of a weighted undirected connected graph, use Prim's algorithm to find a minimum spanning tree of this graph.

18. Given the list of edges and their weights of a weighted undirected connected graph, use Kruskal's algorithm to find a minimum spanning tree of this graph.

Computations and Explorations

Use a computational program or programs you have written to do these exercises.

1. Display all trees with six vertices.

2. Display a full set of nonisomorphic trees with seven vertices.

*3. Construct a Huffman code for the symbols with ASCII codes given the frequency of their occurrence in representative input.

4. Compute the number of different spanning trees of K_n for $n = 1, 2, 3, 4, 5, 6$. Conjecture a formula for the number of such spanning trees whenever n is a positive integer.

5. Compare the number of comparisons needed to sort lists of n elements for $n = 100, 1000$, and $10,000$ from the set of positive integers less than 1,000,000, where the elements

are randomly selected positive integers, using the selection sort, the insertion sort, the merge sort, and the quick sort.

6. Compute the number of different ways n queens can be arranged on an $n \times n$ chessboard so that no two queens can attack each other for all positive integers n not exceeding 10.

*7. Find a minimum spanning tree of the graph that connects the capital cities of the 50 states in the United States to each other where the weight of each edge is the distance between the cities.

8. Draw the complete game tree for a game of checkers on a 4×4 board.

Writing Projects

Respond to these with essays using outside sources.

1. Explain how Cayley used trees to enumerate the number of certain types of hydrocarbons.

2. Explain how trees are used to represent ancestral relations in the study of evolution.

3. Discuss hierarchical cluster trees and how they are used.

4. Define *AVL-trees* (sometimes also known as *height-balanced trees*). Describe how and why AVL-trees are used in a variety of different algorithms.

5. Define *quad trees* and explain how images can be represented using them. Describe how images can be rotated, scaled, and translated by manipulating the corresponding quad tree.

6. Define a *heap* and explain how trees can be turned into heaps. Why are heaps useful in sorting?

7. Describe dynamic algorithms for data compression based on letter frequencies as they change as characters are successively read, such as adaptive Huffman coding.

8. Explain how *alpha-beta pruning* can be used to simplify the computation of the value of a game tree.

9. Describe the techniques used by chess-playing programs such as Deep Blue.

10. Define the type of graph known as a *mesh of trees*. Explain how this graph is used in applications to very large system integration and parallel computing.

11. Discuss the algorithms used in IP multicasting to avoid loops between routers.

12. Describe an algorithm based on depth-first search for finding the articulation points of a graph.

13. Describe an algorithm based on depth-first search to find the strongly connected components of a directed graph.

14. Describe the search techniques used by the crawlers and spiders in different search engines on the Web.

15. Describe an algorithm for finding the minimum spanning tree of a graph such that the maximum degree of any vertex in the spanning tree does not exceed a fixed constant k.

16. Compare and contrast some of the most important sorting algorithms in terms of their complexity and when they are used.

17. Discuss the history and origins of algorithms for constructing minimum spanning trees.

18. Describe algorithms for producing random trees.