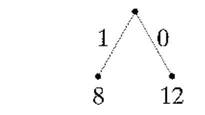


Figure 9.1.14 The tree that replaces the vertex labeled 20 in Figure 9.1.13.

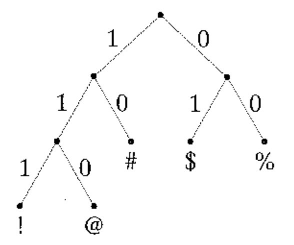


Figure 9.1.15 The final tree of Figure 9.1.13 with each frequency replaced by a character having that frequency.

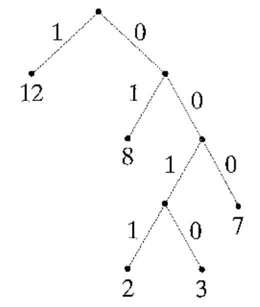


Figure 9.1.16 Another optimal Huffman tree for Example 9.1.10.

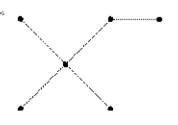
Notice that the Huffman tree for Table 9.1.2 is not unique. When 12 is replaced by 5, 7, because there are two vertices labeled 12, there is a choice. In Figure 9.1.13, we arbitrarily chose one of the vertices labeled 12. If we choose the other vertex labeled 12, we will obtain the tree of Figure 9.1.16. Either of the Huffman trees gives an optimal code; that is, either will encode text having the frequencies of Table 9.1.2 in exactly the same (optimal) space. ◀

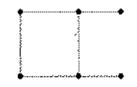
Section Review Exercises

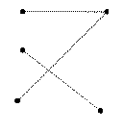
1. Define *free tree*.
2. Define *rooted tree*.
3. What is the level of a vertex in a rooted tree?
4. What is the height of a rooted tree?
5. Give an example of a hierarchical definition tree.
6. Explain how files and folders in a computer system are organized into a rooted tree structure.
7. What is a Huffman code?
8. Explain how to construct an optimal Huffman code.

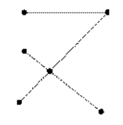
Exercises

Which of the graphs in Exercises 1–4 are trees? Explain.

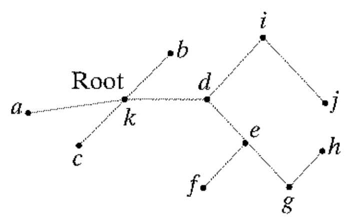
1.


2.


3.


4.


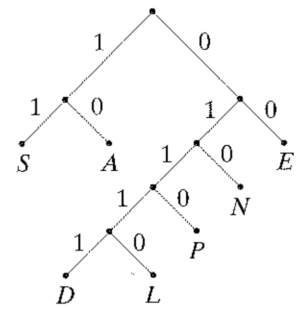
5. For which values of m and n is the complete bipartite graph on m and n vertices a tree?
6. For which values of n is the complete graph on n vertices a tree?
7. For which values of n is the n-cube a tree?
8. Find the level of each vertex in the tree shown.



9. Find the height of the tree of Exercise 8.
10. Draw the tree T of Figure 9.1.5 as a rooted tree with a as root. What is the height of the resulting tree?
11. Draw the tree T of Figure 9.1.5 as a rooted tree with b as root. What is the height of the resulting tree?

12. Give an example similar to Example 9.1.5 of a tree that is used to specify hierarchical relationships.
13. Give an example different from Example 9.1.7 of a hierarchical definition tree.

Decode each bit string using the Huffman code given.



14. 011000010
15. 01110100110
16. 01111001001110
17. 1110011101001111

Encode each word using the preceding Huffman code.

18. DEN
19. NEED
20. LEADEN
21. PENNED
22. What factors in addition to the amount of memory used should be considered when choosing a code, such as ASCII or a Huffman code, to represent characters in a computer?
23. What techniques in addition to the use of Huffman codes might be used to save memory when storing text?
24. Construct an optimal Huffman code for the set of letters in the table.

Letter	*Frequency*	*Letter*	*Frequency*
α	5	δ	11
β	6	ε	20
γ	6		

25. Construct an optimal Huffman code for the set of letters in the table.

Letter	*Frequency*	*Letter*	*Frequency*
I	7.5	C	5.0
U	20.0	H	10.0
B	2.5	M	2.5
S	27.5	P	25.0

26. Use the code developed in Exercise 25 to encode the following words (which have frequencies consistent with the table of Exercise 25):

BUS, CUPS, MUSH, PUSS, SIP, PUSH,
CUSS, HIP, PUP, PUPS, HIPS.

27. Construct two optimal Huffman coding trees for the table of Exercise 24 of different heights.
28. Construct an optimal Huffman code for the set of letters in the table.

Letter	*Frequency*	*Letter*	*Frequency*
a	2	*d*	8
b	3	*e*	13
c	5	*f*	21

29. Professor Ter A. Byte needs to store text made up of the characters A, B, C, D, E, which occur with the following frequencies:

Character	*Frequency*	*Character*	*Frequency*
A	6	D	2
B	2	E	8
C	3		

Professor Byte suggests using the variable-length codes

Character	*Code*
A	1
B	00
C	01
D	10
E	0

which, he argues, store the text in less space than that used by an optimal Huffman code. Is the professor correct? Explain.

30. Show that any tree with two or more vertices has a vertex of degree 1.
31. Show that a tree is a planar graph.
32. Show that a tree is a bipartite graph.
33. Show that the vertices of a tree can be colored with two colors so that each edge is incident on vertices of different colors.

The eccentricity *of a vertex v in a tree T is the maximum length of a simple path that begins at v.*

34. Find the eccentricity of each vertex in the tree of Figure 9.1.5.

A vertex v in a tree T is a center *for T if the eccentricity of v is minimal.*

35. Find the center(s) of the tree of Figure 9.1.5.
★36. Show that a tree has either one or two centers.
★37. Show that if a tree has two centers they are adjacent.
38. Define the radius r of a tree using the concepts of eccentricity and center. The diameter d of any graph was defined before Exercise 70, Section 8.2. Is it always true, according to your definition of radius, that $2r = d$? Explain.
39. Give an example of a tree T that does not satisfy the following property: If v and w are vertices in T, there is a unique path from v to w.

Section Review Exercises

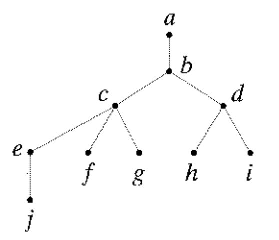
1. Define *parent* in a rooted tree.
2. Define *descendant* in a rooted tree.
3. Define *sibling* in a rooted tree.
4. Define *terminal vertex* in a rooted tree.
5. Define *internal vertex* in a rooted tree.
6. Define *acyclic graph*.
7. Give alternative characterizations of trees.

Exercises

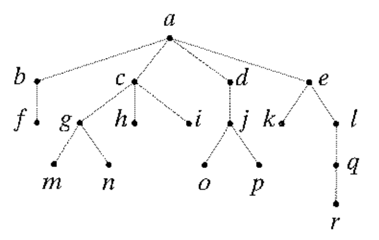
Answer the questions in Exercises 1–6 for the tree in Figure 9.2.1.

1. Find the parent of Poseidon.
2. Find the ancestors of Eros.
3. Find the children of Uranus.
4. Find the descendants of Zeus.
5. Find the siblings of Ares.
6. Draw the subtree rooted at Aphrodite.

Answer the questions in Exercises 7–15 for the following tree.



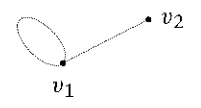
7. Find the parents of c and of h.
8. Find the ancestors of c and of j.
9. Find the children of d and of e.
10. Find the descendants of c and of e.
11. Find the siblings of f and of h.
12. Find the terminal vertices. 13. Find the internal vertices.
14. Draw the subtree rooted at j. 15. Draw the subtree rooted at e.
16. Answer the questions in Exercises 7–15 for the following tree.



17. What can you say about two vertices in a rooted tree that have the same parent?
18. What can you say about two vertices in a rooted tree that have the same ancestors?
19. What can you say about a vertex in a rooted tree that has no ancestors?
20. What can you say about two vertices in a rooted tree that have a descendant in common?
21. What can you say about a vertex in a rooted tree that has no descendants?

In Exercises 22–26, draw a graph having the given properties or explain why no such graph exists.

22. Six edges; eight vertices
23. Acyclic; four edges, six vertices
24. Tree; all vertices of degree 2
25. Tree; six vertices having degrees 1, 1, 1, 1, 3, 3
26. Tree; four internal vertices; six terminal vertices
27. Explain why if we allow cycles of length 0, a graph consisting of a single vertex and no edges is not acyclic.
28. Explain why if we allow cycles to repeat edges, a graph consisting of a single edge and two vertices is not acyclic.
29. The connected graph shown has a unique simple path from any vertex to any other vertex, but it is not a tree. Explain.



A forest is a simple graph with no cycles.

30. Explain why a forest is a union of trees.
31. If a forest F consists of m trees and has n vertices, how many edges does F have?
32. If $P_1 = (v_0, \ldots, v_n)$ and $P_2 = (w_0, \ldots, w_m)$ are distinct simple paths from a to b in a simple graph G, is

$$(v_0, \ldots, v_n = w_m, w_{m-1}, \ldots, w_1, w_0)$$

necessarily a cycle? Explain. (This exercise is relevant to the last paragraph of the proof of Theorem 9.2.3.)

33. Show that a graph G with n vertices and fewer than $n - 1$ edges is not connected.

★34. Prove that T is a tree if and only if T is connected and when an edge is added between any two vertices, exactly one cycle is created.

35. Show that if G is a tree, every vertex of degree 2 or more is an articulation point. ("Articulation point" is defined before Exercise 55, Section 8.2.)
36. Give an example to show that the converse of Exercise 35 is false, even if G is assumed to be connected.

Problem-Solving Corner: Trees

Problem

Let T be a simple graph. Prove that T is a tree if and only if T is connected but the removal of any edge (but no vertices) from T disconnects T.

Attacking the Problem

Let's be clear about what we have to prove. Since the statement is "if and only if," we must prove two statements:

If T is a tree, then T is connected but the removal of any edge (but no vertices) from T disconnects T. **(1)**

If T is connected but the removal of any edge (but no vertices) from T disconnects T, then T is a tree. **(2)**

In (1), from the assumption that T is a tree, we must deduce that T is connected but the removal of any edge (but no vertices) from T disconnects T. In (2), from the assumption that T is connected but the removal of any edge (but no vertices) from T disconnects T, we must deduce that T is a tree.

In developing a proof, it is usually helpful to review definitions and other results related to the statement to be proved. Here of direct relevance are the definition of a tree and Theorem 9.2.3, which gives equivalent conditions for a graph to be a tree.

Definition 9.1.1 states:

A tree T is a simple graph satisfying the following: If v and w are vertices in T, there is a unique simple path from v to w. **(3)**

Theorem 9.2.3 states that the following are equivalent for an n-vertex graph T:

T is a tree.	**(4)**
T is connected and acyclic.	**(5)**
T is connected and has $n - 1$ edges.	**(6)**
T is acyclic and has $n - 1$ edges.	**(7)**

Finding a Solution

Let's first try to prove (1). We assume that T is a tree. We must prove two things: T is connected, and the removal of any edge (but no vertices) from T disconnects T.

Statements (5) and (6) immediately tell us that T is connected. None of the statements (3) through (7) directly tells us anything about removal of edges or about a graph that's not connected. However, if we argue by contradiction and assume that the removal of some edge (but no vertices) from T does not disconnect T, then when we remove that edge from T, the graph T' that results is connected. In this case, for the graph T', (5) is true, but (6) and (7) are false, which is a contradiction since either (5), (6), and (7) are all true (and the graph is a tree), or (5) , (6) , and (7) are all false (and the graph is a not a tree).

Now consider proving (2). We assume that T is connected and that the removal of any edge (but no vertices) from T disconnects T. We must show that T is a tree. Let's try to show that T is connected and acyclic. We can then appeal to (5) to conclude that T is a tree.

Since T is connected, all we have to do is show that T is acyclic. Again, we'll approach this by contradiction. Suppose that T has a cycle. Keeping in mind what we're assuming (the removal of any edge from T disconnects T), try to figure out how to deduce a contradiction from the assumption that T contains a cycle before reading on.

If we delete an edge from T's cycle, T will remain connected. This contradiction shows that T is acyclic. By (5), T is a tree.

Formal Solution

Suppose that T has n vertices.

Suppose that T is a tree. Then by Theorem 9.2.3, T is connected and has $n - 1$ edges. Suppose that we can remove an edge from T to obtain T' so that T' is connected. Since T contains no cycles, T' also contains no cycles. By Theorem 9.2.3, T' is a tree. Again by Theorem 9.2.3, T' has $n - 1$ edges. This is a contradiction. Therefore, T is connected, but the removal of any edge (but no vertices) from T disconnects T.

If T is connected and the removal of any edge (but no vertices) from T disconnects T, then T contains no cycles. By Theorem 9.2.3, T is a tree.

Summary of Problem-Solving Techniques

- When trying to construct a proof, write out carefully what is assumed and what is to be proved.

```
            k = k + 1
            row(k) = 0
        }
        else // backtrack to previous column
            k = k − 1
    }
    return false // no solution
}
```

The tree that Algorithm 9.3.10 generates is shown in Figure 9.3.3. The numbering indicates the order in which the vertices are generated. The solution is found at vertex 8.

The n-queens problem is to place n tokens on an $n \times n$ grid so that no two tokens are on the same row, column, or diagonal. Checking that there is no solution to the two- or three-queens problem (see Exercise 10) is straightforward. We have just seen that

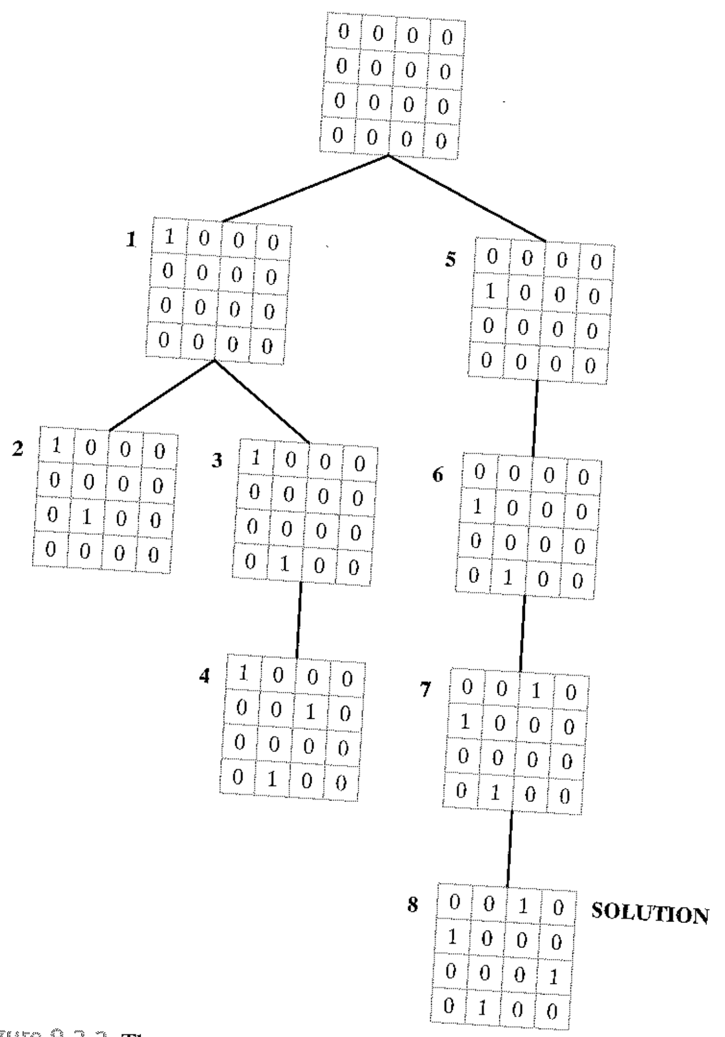


Figure 9.3.3 The tree generated by the backtracking algorithm (Algorithm 9.3.10) in the search for a solution to the four-queens problem.

Algorithm 9.3.10 generates a solution to the four-queens problem. Many constructions have been given to generate solutions to the n-queens problem for all $n \geq 4$ (see, e.g., [Johnsonbaugh]).

Backtracking or depth-first search is especially attractive in a problem such as that in Example 9.3.9, where all that is desired is one solution. Since a solution, if one exists, is found at a terminal vertex, by moving to the terminal vertices as rapidly as possible, in general we can avoid generating some unnecessary vertices.

Problem-Solving Tips

Depth-first search and breadth-first search are the basis of many graph algorithms. For example, either can be used to determine whether a graph is connected: If we can visit *every* vertex in a graph from an initial vertex, the graph is connected; otherwise, it is not connected (see Exercises 30 and 31). Depth-first search can be used as a searching algorithm, in which case it is called backtracking. In Algorithm 9.3.10, backtracking is used to search for solutions to the 4-queens problem. Backtracking can also be used to search for Hamiltonian cycles in a graph, to generate permutations, to solve Sudoku puzzles, and to determine whether two graphs are isomorphic.

Section Review Exercises

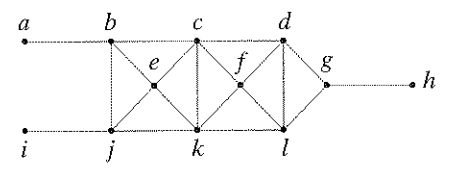
1. What is a spanning tree?
2. State a necessary and sufficient condition for a graph to have a spanning tree.
3. Explain how breadth-first search works.
4. Explain how depth-first search works.
5. What is backtracking?

Exercises

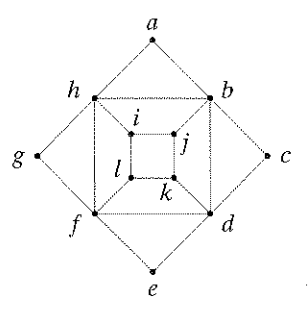
1. Use breadth-first search (Algorithm 9.3.6) with the vertex ordering *hgfedcba* to find a spanning tree for graph G of Figure 9.3.1.
2. Use breadth-first search (Algorithm 9.3.6) with the vertex ordering *hfdbgeca* to find a spanning tree for graph G of Figure 9.3.1.
3. Use breadth-first search (Algorithm 9.3.6) with the vertex ordering *chbgadfe* to find a spanning tree for graph G of Figure 9.3.1.
4. Use depth-first search (Algorithm 9.3.7) with the vertex ordering *hgfedcba* to find a spanning tree for graph G of Figure 9.3.1.
5. Use depth-first search (Algorithm 9.3.7) with the vertex ordering *hfdbgeca* to find a spanning tree for graph G of Figure 9.3.1.
6. Use depth-first search (Algorithm 9.3.7) with the vertex ordering *dhcbefag* to find a spanning tree for graph G of Figure 9.3.1.

In Exercises 7–9, find a spanning tree for each graph.

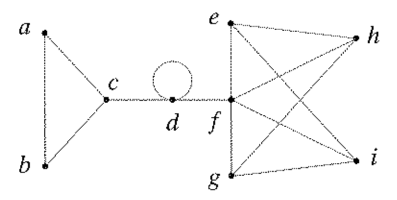
7.



8.



9.



10. Show that there is no solution to the two-queens or the three-queens problem.
11. Show all solutions to the four-queens problem.
12. Find a solution to the five-queens and six-queens problems.

13. Show all solutions to the five-queens problem in which one queen is in the first column, second row.

14. How many solutions are there to the five-queens problem?

15. Show all solutions to the six-queens problem in which one queen is in row 2, column 1, and a second queen is in row 4, column 2.

16. True or false? If G is a connected graph and T is a spanning tree for G, there is an ordering of the vertices of G such that Algorithm 9.3.6 produces T as a spanning tree. If true, prove it; otherwise, give a counterexample.

17. True or false? If G is a connected graph and T is a spanning tree for G, there is an ordering of the vertices of G such that Algorithm 9.3.7 produces T as a spanning tree. If true, prove it; otherwise, give a counterexample.

18. Show, by an example, that Algorithm 9.3.6 can produce identical spanning trees for a connected graph G from two distinct vertex orderings of G.

19. Show, by an example, that Algorithm 9.3.7 can produce identical spanning trees for a connected graph G from two distinct vertex orderings of G.

20. Prove that Algorithm 9.3.6 is correct.

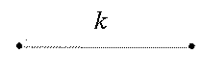
21. Prove that Algorithm 9.3.7 is correct.

22. Under what conditions is an edge in a connected graph G contained in every spanning tree of G?

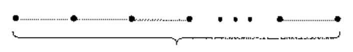
23. Let T and T' be two spanning trees of a connected graph G. Suppose that an edge x is in T but not in T'. Show that there is an edge y in T' but not in T such that $(T - \{x\}) \cup \{y\}$ and $(T' - \{y\}) \cup \{x\}$ are spanning trees of G.

24. Write an algorithm based on breadth-first search that finds the minimum length of each path in an unweighted graph from a fixed vertex v to all other vertices.

25. Let G be a weighted graph in which the weight of each edge is a positive integer. Let G' be the graph obtained from G by replacing each edge



in G of weight k by k unweighted edges in series:



Show that Dijkstra's algorithm for finding the minimum length of each path in the weighted graph G from a fixed vertex v to all other vertices (Algorithm 8.4.1) and performing a breadth-first search in the unweighted graph G' starting with vertex v are, in effect, the same process.

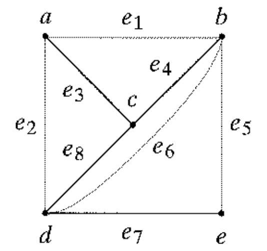
26. Let T be a spanning tree for a graph G. Show that if an edge in G, but not in T, is added to T, a unique cycle is produced.

A cycle as described in Exercise 26 is called a fundamental cycle. *The* fundamental cycle matrix *of a graph G has its rows indexed by the fundamental cycles of G relative to a spanning tree T for G and its columns indexed by the edges of G. The ijth entry is 1 if edge j is in the ith fundamental cycle and 0 otherwise. For example, the fundamental cycle matrix of the graph G of Figure 9.3.1 relative to the spanning tree shown in Figure 9.3.1 is*

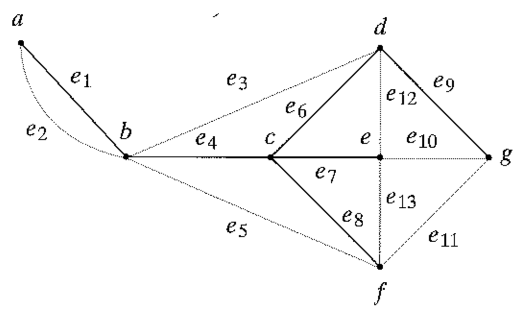
$$\begin{array}{c} \\ (abdca) \\ (efdbace) \\ (ageca) \\ (aga) \\ (abga) \end{array} \begin{array}{c} \begin{array}{cccccccccccc} e_7 & e_6 & e_{11} & e_{10} & e_2 & e_1 & e_3 & e_4 & e_5 & e_8 & e_9 & e_{12} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}.$$

Find the fundamental cycle matrix of each graph. The spanning tree to be used is drawn in black.

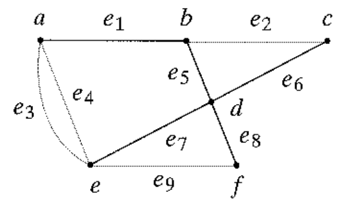
27.



28.



29.



30. Write a breadth-first search algorithm to test whether a graph is connected.

31. Write a depth-first search algorithm to test whether a graph is connected.

32. Write a depth-first search algorithm that finds all solutions to the four-queens problem.

33. Modify Algorithm 9.3.6 so that it tracks the parent p of a vertex c (p is the *parent* of c if c was visited from p).

34. Write an algorithm that uses the output of your algorithm in Exercise 33 to print each vertex and its parent.

35. Modify Algorithm 9.3.7 so that it tracks the parent p of a vertex c (p is the *parent* of c if c was visited from p).

36. Write an algorithm that uses the output of your algorithm in Exercise 35 to print each vertex and its parent.

37. Write a backtracking algorithm that outputs all permutations of $1, 2, \ldots, n$.

38. Write a backtracking algorithm that outputs all subsets of $\{1, 2, \ldots, n\}$.

39. Sudoku is a puzzle in which the goal is to fill in a 9×9 grid so that each of the numbers 1 through 9 appears in each column, each row, and each 3×3 box delineated by the heavy lines:

1				5		3		
6		2	8					
	5			4				2
			7					8
	3	8	2		4	7	6	
9					8			
8				2			3	
					9	5		4
		6		7				9

As shown, in each puzzle some numbers are given. Solve the preceding Sudoku puzzle.

40. Write a backtracking algorithm that solves an arbitrary Sudoku puzzle.

41. The *minimum-queens problem* asks for the minimum number of queens that can attack all of the squares of an $n \times n$ board (i.e., the minimum number of queens such that each row, column, and diagonal contains at least one queen). Write a backtracking algorithm that determines whether k queens can attack all squares of an $n \times n$ board.

42. The *subset-sum problem* is: Given a set $\{c_1, \ldots, c_n\}$ of positive integers and a positive integer M, find all subsets $\{c_{k_1}, \ldots, c_{k_j}\}$ of $\{c_1, \ldots, c_n\}$ satisfying

$$\sum_{i=1}^{j} c_{k_i} = M.$$

Write a backtracking algorithm to solve the subset-sum problem.

9.4 → Minimal Spanning Trees

The weighted graph G of Figure 9.4.1 shows six cities and the costs of building roads between certain pairs of cities. We want to build the lowest-cost road system that will connect the six cities. The solution can be represented by a subgraph. This subgraph must be a spanning tree since it must contain all the vertices (so that each city is in the road system), it must be connected (so that any city can be reached from any other), and it must have a unique simple path between each pair of vertices (since a graph containing multiple simple paths between a vertex pair could not represent a minimum-cost system). Thus what is needed is a spanning tree the sum of whose weights is a minimum. Such a tree is called a **minimal spanning tree.**

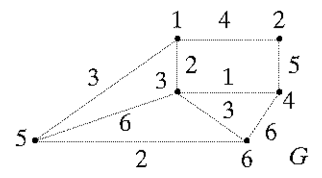


Figure 9.4.1 Six cities 1–6 and the costs of building roads between certain pairs of them.

Definition 9.4.1 ▸ Let G be a weighted graph. A *minimal spanning tree* of G is a spanning tree of G with minimum weight.

15 and 16, vertex 5 is added to the minimal spanning tree and edge (1, 5) is added to E.

The next time we execute the for loop in lines 7–14, the edges with one vertex in the tree and one vertex not in the tree are

Edge	*Weight*
(1, 2)	4
(2, 4)	5
(3, 6)	3
(4, 6)	6
(5, 6)	2

The edge (5, 6) with minimum weight is selected. At lines 15 and 16, vertex 6 is added to the minimal spanning tree and edge (5, 6) is added to E.

The last time we execute the for loop in lines 7–14, the edges with one vertex in the tree and one vertex not in the tree are

Edge	*Weight*
(1, 2)	4
(2, 4)	5

The edge (1, 2) with minimum weight is selected. At lines 15 and 16, vertex 2 is added to the minimal spanning tree and edge (1, 2) is added to E. The minimal spanning tree constructed is shown in Figure 9.4.3. ◀

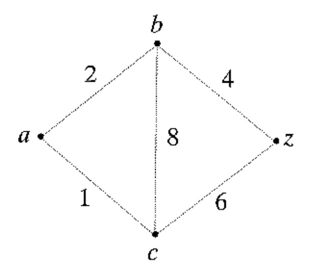


Figure 9.4.4 A graph that shows that selecting an edge having minimum weight incident on the most recently added vertex does *not* necessarily yield a shortest path. Starting at a, we obtain (a, c, z), but the shortest path from a to z is (a, b, z).

Prim's Algorithm furnishes an example of a **greedy algorithm.** A greedy algorithm is an algorithm that optimizes the choice at each iteration. The principle can be summarized as "doing the best locally." In Prim's Algorithm, since we want a minimal spanning tree, at each iteration we simply add an available edge with minimum weight.

Optimizing at each iteration does not necessarily give an optimal solution to the original problem. We will show shortly (Theorem 9.4.5) that Prim's Algorithm is correct—we do obtain a minimal spanning tree. As an example of a greedy algorithm that does not lead to an optimal solution, consider a "shortest-path algorithm" in which at each step we select an available edge having minimum weight incident on the most recently added vertex. If we apply this algorithm to the weighted graph of Figure 9.4.4 to find a shortest path from a to z, we would select the edge (a, c) and then the edge (c, z). Unfortunately, this is not the shortest path from a to z.

We next show that Prim's Algorithm is correct.

Theorem 9.4.5 *Prim's Algorithm (Algorithm 9.4.3) is correct; that is, at the termination of Algorithm 9.4.3, T is a minimal spanning tree.*

Proof We let T_i denote the graph constructed by Algorithm 9.4.3 after the ith iteration of the for loop, lines 5–17. More precisely, the edge set of T_i is the set E constructed after the ith iteration of the for loop, lines 5–17, and the vertex set of T_i is the set of vertices on which the edges in E are incident. We let T_0 be the graph constructed by Algorithm 9.4.3 just before the for loop at line 5 is entered for the first time; T_0 consists of the single vertex s and no edges. Subsequently in this proof, we suppress the vertex set and refer to a graph by specifying its edge set.

By construction, at the termination of Algorithm 9.4.3, the resulting graph, T_{n-1}, is a connected, acyclic subgraph of the given graph G containing all the vertices of G; hence T_{n-1} is a spanning tree of G.

We use induction to show that for all $i = 0, \ldots, n-1$, T_i is contained in a minimal spanning tree. It then follows that at termination, T_{n-1} is a minimal spanning tree.

If $i = 0$, T_0 consists of a single vertex. In this case T_0 is contained in every minimal spanning tree. We have verified the Basis Step.

Next, assume that T_i is contained in a minimal spanning tree T'. Let V be the set of vertices in T_i. Algorithm 9.4.3 selects an edge (j, k) of minimum weight, where $j \in V$ and $k \notin V$, and adds it to T_i to produce T_{i+1}. If (j, k) is in T', then T_{i+1} is contained in the minimal spanning tree T'. If (j, k) is not in T', $T' \cup \{(j, k)\}$ contains a cycle C. Choose an edge (x, y) in C, different from (j, k), with $x \in V$ and $y \notin V$. Then

$$w(x, y) \geq w(j, k). \tag{9.4.1}$$

Because of (9.4.1), the graph $T'' = [T' \cup \{(j, k)\}] - \{(x, y)\}$ has weight less than or equal to the weight of T'. Since T'' is a spanning tree, T'' is a minimal spanning tree. Since T_{i+1} is contained in T'', the Inductive Step has been verified. The proof is complete.

Our version of Prim's Algorithm examines $\Theta(n^3)$ edges in the worst case (see Exercise 6) to find a minimal spanning tree for a graph having n vertices. It is possible (see Exercise 8) to implement Prim's Algorithm so that only $\Theta(n^2)$ edges are examined in the worst case. Since K_n has $\Theta(n^2)$ edges, the latter version is optimal.

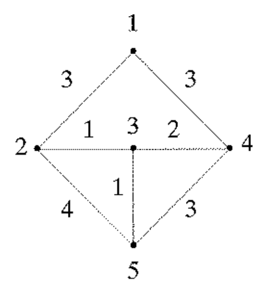
Section Review Exercises

1. What is a minimal spanning tree?
2. Explain how Prim's Algorithm finds a minimal spanning tree.
3. What is a greedy algorithm?

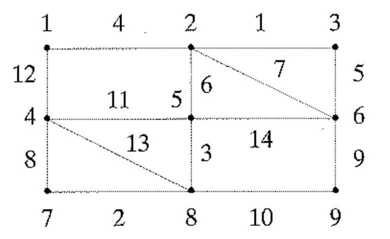
Exercises

In Exercises 1–5, find the minimal spanning tree given by Algorithm 9.4.3 for each graph.

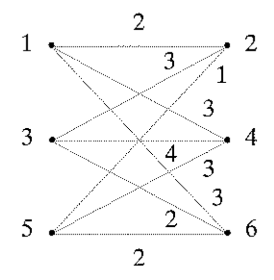
1.



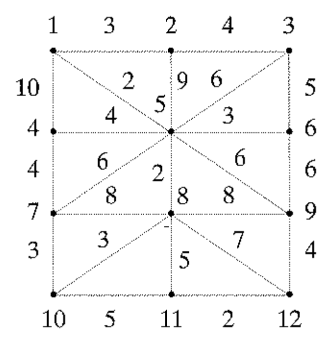
2.



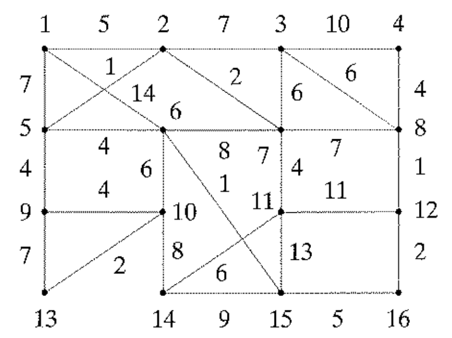
3.



4.



5.



6. Show that Algorithm 9.4.3 examines $\Theta(n^3)$ edges in the worst case.

Exercises 7–9 refer to an alternate version of Prim's Algorithm (Algorithm 9.4.6).

Algorithm 9.4.6
Alternate Version of Prim's Algorithm

This algorithm finds a minimal spanning tree in a connected, weighted graph G. At each step, some vertices have temporary labels and some have permanent labels. The label of vertex i is denoted L_i.

Input: A connected, weighted graph with vertices $1, \ldots, n$ and start vertex s. If (i, j) is an edge, $w(i, j)$ is equal to the weight of (i, j); if (i, j) is not an edge, $w(i, j)$ is equal to ∞ (a value greater than any actual weight).

Output: A minimal spanning tree T

```
prim_alternate(w, n, s) {
  let T be the graph with vertex s and no edges
  for j = 1 to n {
    $L_j = w(s, j)$ // these labels are temporary
    back(j) = s
  }
  $L_s = 0$
  make $L_s$ permanent
  while (temporary labels remain) {
    choose the smallest temporary label $L_i$
    make $L_i$ permanent
    add edge (i, back(i)) to T
    add vertex i to T
    for each temporary label $L_k$
      if ($w(i, k) < L_k$) {
        $L_k = w(i, k)$
        back(k) = i
      }
  }
  return T
}
```

7. Show how Algorithm 9.4.6 finds a minimal spanning tree for the graphs of Exercises 1–5.

8. Show that Algorithm 9.4.6 examines $\Theta(n^2)$ edges in the worst case.

9. Prove that Algorithm 9.4.6 is correct; that is, at the termination of Algorithm 9.4.6, T is a minimal spanning tree.

10. Let G be a connected, weighted graph, let v be a vertex in G, and let e be an edge of minimum weight incident on v. Show that e is contained in some minimal spanning tree.

11. Let G be a connected, weighted graph and let v be a vertex in G. Suppose that the weights of the edges incident on v are distinct. Let e be the edge of minimum weight incident on v. Must e be contained in every minimal spanning tree?

12. Show that any algorithm that finds a minimal spanning tree in K_n, when all the weights are the same, must examine every edge in K_n.

13. Show that if all weights in a connected graph G are distinct, G has a unique minimal spanning tree.

In Exercises 14–16, decide if the statement is true or false. If the statement is true, prove it; otherwise, give a counterexample. In each exercise, G is a connected, weighted graph.

14. If all the weights in G are distinct, distinct spanning trees of G have distinct weights.

15. If e is an edge in G whose weight is less than the weight of every other edge, e is in every minimal spanning tree of G.

16. If T is a minimal spanning tree of G, there is a labeling of the vertices of G so that Algorithm 9.4.3 produces T.

17. Let G be a connected, weighted graph. Show that if, as long as possible, we remove an edge from G having maximum weight whose removal does not disconnect G, the result is a minimal spanning tree for G.

★18. Write an algorithm that finds a maximal spanning tree in a connected, weighted graph.

19. Prove that your algorithm in Exercise 18 is correct.

Kruskal's Algorithm finds a minimal spanning tree in a connected, weighted graph G having n vertices as follows. The graph T initially consists of the vertices of G and no edges. At each iteration, we add an edge e to T having minimum weight that does not complete a cycle in T. When T has n − 1 edges, we stop.

20. Formally state Kruskal's Algorithm.

21. Show how Kruskal's Algorithm finds minimal spanning trees for the graphs of Exercises 1–5.

22. Show that Kruskal's Algorithm is correct; that is, at the termination of Kruskal's Algorithm, T is a minimal spanning tree.

23. Let V be a set of n vertices and let s be a "dissimilarity function" on $V \times V$ (see Example 8.1.7). Let G be the complete, weighted graph having vertices V and weights $w(v_i, v_j) = s(v_i, v_j)$. Modify Kruskal's Algorithm so that it groups data into classes. This modification is known as the **method of nearest neighbors** (see [Gose]).

Exercises 24–30 refer to the following situation. Suppose that we have stamps of various denominations and that we want to choose the minimum number of stamps to make a given amount of postage. Consider a greedy algorithm that selects stamps by choosing as many of the largest denomination as possible, then as many of the second-largest denomination as possible, and so on.

24. Show that if the available denominations are 1, 8, and 10 cents, the algorithm does not always produce the fewest number of stamps to make a given amount of postage.

★25. Show that if the available denominations are 1, 5, and 25 cents, the algorithm produces the fewest number of stamps to make any given amount of postage.

26. Find positive integers a_1 and a_2 such that $a_1 > 2a_2 > 1$, a_2 does not divide a_1, and the algorithm, with available denominations $1, a_1, a_2$, does not always produce the fewest number of stamps to make a given amount of postage.

★27. Find positive integers a_1 and a_2 such that $a_1 > 2a_2 > 1$, a_2 does not divide a_1, and the algorithm, with available denominations $1, a_1, a_2$, produces the fewest number of stamps to make any given amount of postage. Prove that your values do give an optimal solution.

★28. Suppose that the available denominations are

$$1 = a_1 < a_2 < \cdots < a_n.$$

Show, by giving counterexamples, that the condition

$$a_i \geq 2a_{i-1} - a_{i-2}, \qquad 3 \leq i \leq n,$$

is neither necessary nor sufficient for the greedy algorithm to be optimal for all amounts of postage.

★29. Suppose that the available denominations are

$$1 = a_1 < a_2 < \cdots < a_m.$$

Prove that if the greedy algorithm is optimal for all amounts of postage less than $a_{m-1} + a_m$, then it is optimal for all amounts of postage.

30. Show that the bound $a_{m-1} + a_m$ in Exercise 29 cannot be lowered.

31. What is wrong with the following "proof" that the greedy algorithm is optimal for all amounts of postage for the denominations 1, 5, and 6?

We will prove that for all $i \geq 1$, the greedy algorithm is optimal for all amounts of postage $n \leq 6i$. The Basis Step is $i = 1$, which is true by inspection.

For the Inductive Step, assume that the greedy algorithm is optimal for all amounts of postage $n \leq 6i$. We must show that the greedy algorithm is optimal for all amounts of postage $n \leq 6(i + 1)$. We may assume that $n > 6i$. Now $n - 6 \leq 6i$, so by the inductive assumption, the greedy algorithm is optimal for $n - 6$. Suppose that the greedy algorithm chooses k stamps for $n - 6$. In determining the postage for the amount n, the greedy algorithm will first choose a 6-cent stamp and then stamps for $n - 6$ for a total of $k + 1$ stamps These $k + 1$ stamps must be optimal or otherwise $n - 6$ would use less than k stamps. The Inductive Step is complete.

9.5 ➾ Binary Trees

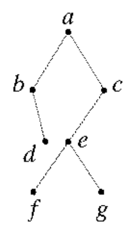


Figure 9.5.1 A binary tree.

Binary trees are among the most important special types of rooted trees. Every vertex in a binary tree has at most two children (see Figure 9.5.1). Moreover, each child is designated as either a **left child** or a **right child.** When a binary tree is drawn, a left child is drawn to the left and a right child is drawn to the right. The formal definition follows.

Definition 9.5.1 ▶ A *binary tree* is a rooted tree in which each vertex has either no children, one child, or two children. If a vertex has one child, that child is designated as either a left child or a right child (but not both). If a vertex has two children, one child is designated a left child and the other child is designated a right child. ◀

Example 9.5.2 ▶ In the binary tree of Figure 9.5.1, vertex b is the left child of vertex a and vertex c is the right child of vertex a. Vertex d is the right child of vertex b; vertex b has no left child. Vertex e is the left child of vertex c; vertex c has no right child. ◀

Example 9.5.3 ▶ A tree that defines a Huffman code is a binary tree. For example, in the Huffman coding tree of Figure 9.1.10, moving from a vertex to a left child corresponds to using the bit 1, and moving from a vertex to a right child corresponds to using the bit 0. ◀

```
            if ($v$ has no right child) {
                add a right child $r$ to $v$
                store $w_i$ in $r$
                search = false // end search
            }
            else
                $v$ = right child of $v$
        } // end while
    } // end for
    return $T$
}
```

Binary search trees are useful for locating data. That is, given a data item D, we can easily determine if D is in a binary search tree and, if it is present, where it is located. To determine if a data item D is in a binary search tree, we would begin at the root. We would then repeatedly compare D with the data item at the current vertex. If D is equal to the data item at the current vertex, we have found D, so we stop. If D is less than the data item at the current vertex v, we move to v's left child and repeat this process. If D is greater than the data item at the current vertex v, we move to v's right child and repeat this process. If at any point the child to move to is missing, we conclude that D is not in the tree. (Exercise 6 asks for a formal statement of this process.)

The time spent searching for an item in a binary search tree is longest when the item is not present and we follow a longest path from the root. Thus the maximum time to search for an item in a binary search tree is approximately proportional to the height of the tree. Therefore, if the height of a binary search tree is small, searching the tree will always be very fast (see Exercise 25). Many ways are known to minimize the height of a binary search tree (see, e.g., [Cormen]).

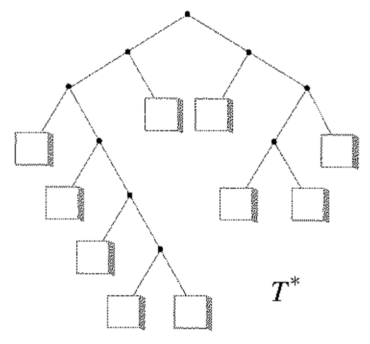


Figure 9.5.6 Expanding a binary search tree to a full binary tree.

We make more precise statements about worst-case searching in a binary search tree. Let T be a binary search tree with n vertices and let T^* be the full binary tree obtained from T by adding left and right children to existing vertices in T wherever possible. In Figure 9.5.6, we show the full binary tree that results from modifying the binary search tree of Figure 9.5.4. The added vertices are drawn as boxes. An unsuccessful search in T corresponds to arriving at an added (box) vertex in T^*. Let us define the worst-case time needed to execute the search procedure as the height h of the tree T^*. By Theorem 9.5.6, $\lg t \le h$, where t is the number of terminal vertices in T^*. The full binary tree T^* has n internal vertices, so by Theorem 9.5.4, $t = n + 1$. Thus in the worst case, the time will be equal to at least $\lg t = \lg(n + 1)$. Exercise 7 shows that if the height of T is minimized, the worst case requires time equal to $\lceil \lg(n + 1) \rceil$. For example, since

$$\lceil \lg(2{,}000{,}000 + 1) \rceil = 21,$$

it is possible to store 2 million items in a binary search tree and find an item, or determine that it is not present, in at most 21 steps.

Section Review Exercises

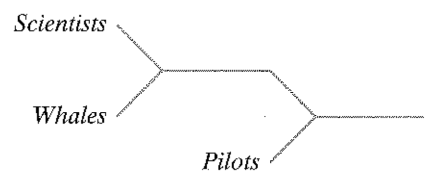
1. Define *binary tree.*
2. What is a left child in a binary tree?
3. What is a right child in a binary tree?
4. What is a full binary tree?
5. If T is a full binary tree with i internal vertices, how many terminal vertices does T have?
6. If T is a full binary tree with i internal vertices, how many total vertices does T have?

7. How is the height of a binary tree related to the number of its terminal vertices?
8. What is a binary search tree?
9. Give an example of a binary search tree.
10. Give an algorithm to construct a binary search tree.

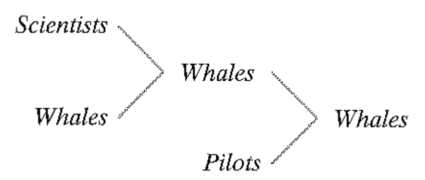
Exercises

Exercises 1–4 concern n teams that play a single-elimination tournament.

1. After the teams are assigned, in how many ways can the tournament unfold? For example, if there are three teams, Scientists, Whales, Pilots, assigned as



one way the tournament can unfold is



There are three other ways that the tournament can unfold:

(a) Whales defeat Scientists; Pilots defeat Whales.

(b) Scientist defeat Whales; Scientists defeat Pilots.

(c) Scientist defeat Whales; Pilots defeat Scientists.

Thus, if three teams play a single-elimination tournament, after the teams are assigned, the tournament can unfold in four ways.

2. As of 2007, the NCAA men's basketball tournament was a 65-team single-elimination tournament. After the teams are assigned, in how many ways can the tournament unfold? How many (base-10) digits does this number have?

3. Suppose that after the teams are assigned in the NCAA men's basketball tournament, someone randomly guesses how the tournament will unfold. What is the probability that the guess is correct?

4. Is the value in Exercise 3 a good estimate of the chance that someone knowledgeable about basketball will successfully predict how the tournament will unfold?

5. Place the words FOUR SCORE AND SEVEN YEARS AGO OUR FOREFATHERS BROUGHT FORTH, in the order in which they appear, in a binary search tree.

6. Write a formal algorithm for searching in a binary search tree.

7. Write an algorithm that stores n distinct words in a binary search tree T of minimal height. Show that the derived tree T^*, as described in the text, has height $\lceil \lg(n + 1) \rceil$.

8. True or false? Let T be a binary tree. If for every vertex v in T the data item in v is greater than the data item in the left child of v and the data item in v is less than the data item in the right child of v, then T is a binary search tree. Explain.

In Exercises 9–11, draw a graph having the given properties or explain why no such graph exists.

9. Full binary tree; four internal vertices; five terminal vertices

10. Full binary tree; height = 3; nine terminal vertices

11. Full binary tree; height = 4; nine terminal vertices

12. A **full m-ary tree** is a rooted tree such that every parent has m ordered children. If T is a full m-ary tree with i internal vertices, how many vertices does T have? How many terminal vertices does T have? Prove your results.

13. Give an algorithm for constructing a full binary tree with $n > 1$ terminal vertices.

14. Give a recursive algorithm to insert a word in a binary search tree.

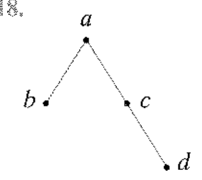
15. Find the maximum height of a full binary tree having t terminal vertices.

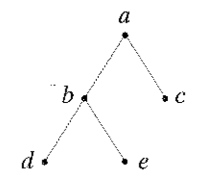
16. Write an algorithm that tests whether a binary tree in which data are stored in the vertices is a binary search tree.

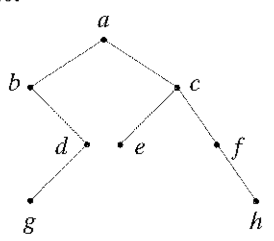
17. Let T be a full binary tree. Let I be the sum of the lengths of the simple paths from the root to the internal vertices. We call I the *internal path length*. Let E be the sum of the lengths of the simple paths from the root to the terminal vertices. We call E the *external path length*. Prove that if T has n internal vertices, then $E = I + 2n$.

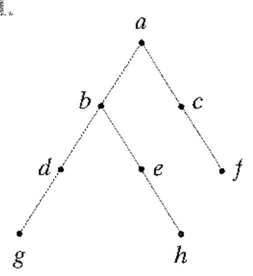
A binary tree T is balanced if for every vertex v in T, the heights of the left and right subtrees of v differ by at most 1. (Here the height of a "missing subtree" is defined to be −1.)

State whether each tree in Exercises 18–21 is balanced or not.

18.


19.


20.


21.


In Exercises 22–24, N_h is defined as the minimum number of vertices in a balanced binary tree of height h and $f_1, f_2, \ldots$ denotes the Fibonacci sequence.

22. Show that $N_0 = 1$, $N_1 = 2$, and $N_2 = 4$.
23. Show that $N_h = 1 + N_{h-1} + N_{h-2}$, for $h \geq 0$.
24. Show that $N_h = f_{h+3} - 1$, for $h \geq 0$.

★25. Show that the height h of an n-vertex balanced binary tree satisfies $h = O(\lg n)$. This result shows that the worst-case time to search in an n-vertex balanced binary search tree is $O(\lg n)$.

★26. Prove that if a binary tree of height h has $n \geq 1$ vertices, then $\lg n < h+1$. This result, together with Exercise 25, shows that the worst-case time to search in an n-vertex balanced binary search tree is $\Theta(\lg n)$.

9.6 ◈ Tree Traversals

Breadth-first search and depth-first search provide ways to "walk" a tree, that is, to traverse a tree in a systematic way so that each vertex is visited exactly once. In this section we consider three additional tree-traversal methods. We define these traversals recursively.

Algorithm 9.6.1 Preorder Traversal

This recursive algorithm processes the vertices of a binary tree using preorder traversal.

Input: *PT*, the root of a binary tree, or the special value *null* to indicate that no tree is input

Output: Dependent on how "process" is interpreted in line 3

```
   preorder(PT) {
1.   if (PT == null)
2.     return
3.   process PT
4.   l = left child of PT
5.   preorder(l)
6.   r = right child of PT
7.   preorder(r)
   }
```

Let us examine Algorithm 9.6.1 for some simple cases. If no tree is input (i.e., PT equals *null*), nothing is processed since, in this case, the algorithm simply returns at line 2.

Suppose that the input consists of a tree with a single vertex. We set PT to the root and call *preorder*(PT). Since PT is not equal to *null*, we proceed to line 3, where we process the root. At line 5, we call *preorder* with PT equal to *null* since there is no left child. However, we just saw that when no tree is input to *preorder*, nothing is processed. Similarly at line 7, when no tree is input to *preorder*, again nothing is processed. Thus when the input consists of a tree with a single vertex, we process the root and return.

Now suppose that the input is the tree of Figure 9.6.1. We set PT to the root and call *preorder*(PT). Since PT is not equal to *null*, we proceed to line 3, where we process the root. At line 5 we call *preorder* with PT equal to the left child of the root (see Figure 9.6.2). We just saw that if the tree input to *preorder* consists of a single vertex, *preorder* processes that vertex. Thus we next process vertex B. Similarly, at line 7, we process vertex C. Thus the vertices are processed in the order ABC.

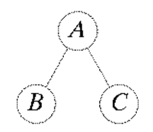


Figure 9.6.1 Input for Algorithm 9.6.1.

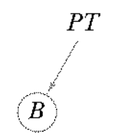


Figure 9.6.2 At line 5 of Algorithm 9.6.1, where the input is the tree of Figure 9.6.1.

Example 9.6.2 In what order are the vertices of the tree of Figure 9.6.3 processed if preorder traversal is used?

Following lines 3–7 (root/left/right) of Algorithm 9.6.1, the traversal proceeds as shown in Figure 9.6.4. Thus the order of processing is $ABCDEFGHIJ$.

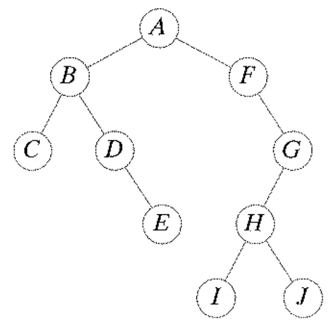


Figure 9.6.3 A binary tree. Preorder is $ABCDEFGHIJ$. Inorder is $CBDEAFIHJG$. Postorder is $CEDBIJHGFA$.

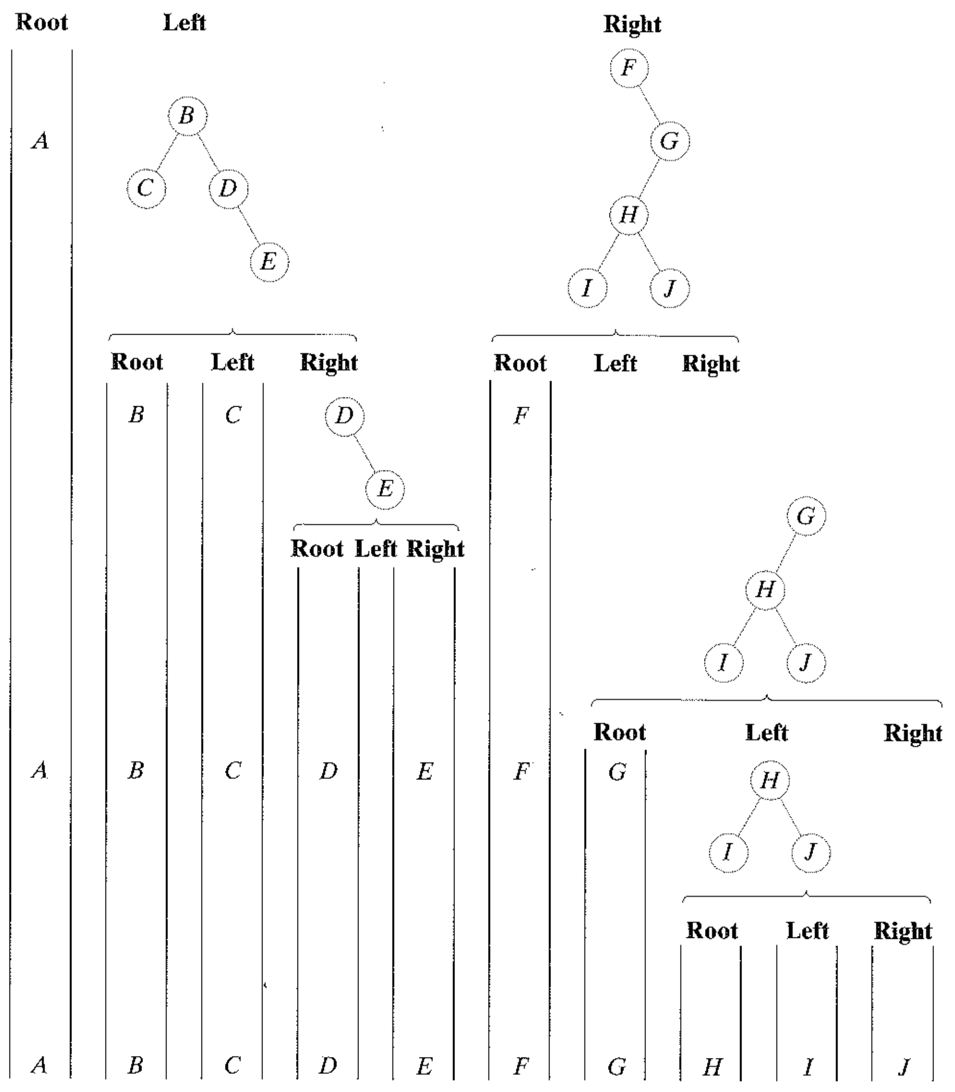


Figure 9.6.4 Preorder traversal of the tree in Figure 9.6.3. ◀