CHAPTER 11
Trees

SECTION 11.1 Introduction to Trees

These exercises give the reader experience working with tree terminology, and in particular with the relationships between the height and the numbers of vertices, leaves, and internal vertices of a tree. Exercise 13 should be done to get a feeling for the structure of trees. One good way to organize your enumeration of trees (such as all nonisomorphic trees with five vertices) is to focus on a particular parameter, such as the length of a longest path in the tree. This makes it easier to include all the trees and not count any of them twice. Review the theorems in this section before working the exercises involving the relationships between the height and the numbers of vertices, leaves, and internal vertices of a tree. For a challenge that gives a good feeling for the flavor of arguments in graph theory, the reader should try Exercise 43. In many ways trees are recursive creatures, and Exercises 45 and 46 are worth looking at in this regard.

1. **a)** This graph is connected and has no simple circuits, so it is a tree.

b) This graph is not connected, so it is not a tree.

c) This graph is connected and has no simple circuits, so it is a tree.

d) This graph has a simple circuit, so it is not a tree.

e) This graph is connected and has no simple circuits, so it is a tree.

f) This graph has a simple circuit, so it is not a tree.

3. **a)** Vertex a is the root, since it is drawn at the top.

b) The internal vertices are the vertices with children, namely a, b, c, d, f, h, j, q, and t.

c) The leaves are the vertices without children, namely e, g, i, k, l, m, n, o, p, r, s, and u.

d) The children of j are the vertices adjacent to j and below j, namely q and r.

e) The parent of h is the vertex adjacent to h and above h, namely c.

f) Vertex o has only one sibling, namely p, which is the other child of o's parent, h.

g) The ancestors of m are all the vertices on the unique simple path from m back to the root, namely f, b, and a.

h) The descendants of b are all the vertices that have b as an ancestor, namely e, f, l, m, and n.

5. This is not a full m-ary tree for any m. It is an m-ary tree for all $m \geq 3$, since each vertex has at most 3 children, but since some vertices have 3 children, while others have 1 or 2, it is not full for any m.

7. We can easily determine the levels from the drawing. The root a is at level 0. The vertices in the row below a are at level 1, namely b, c, and d. The vertices below that, namely e through k (in alphabetical order), are at level 2. Similarly l through r are at level 3, s and t are at level 4, and u is at level 5.

9. We describe the answers, rather than actually drawing pictures.

a) The subtree rooted at a is the entire tree, since a is the root.

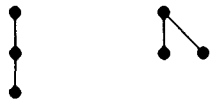
b) The subtree rooted at c consists of five vertices—the root c, children g and h of this root, and grandchildren o and p—and the four edges cg, ch, ho, and hp.

c) The subtree rooted at e is just the vertex e.

11. We find the answer by carefully enumerating these trees, i.e., drawing a full set of nonisomorphic trees. One way to organize this work so as to avoid leaving any trees out or counting the same tree (up to isomorphism) more than once is to list the trees by the length of their longest simple path (or longest simple path from the root in the case of rooted trees).

a) There is only one tree with three vertices, namely $K_{1,2}$ (which can also be thought of as the simple path of length 2).

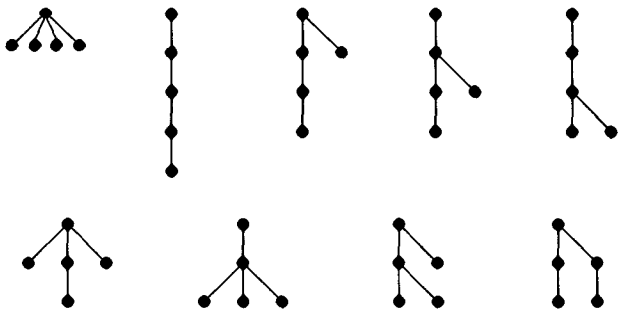
b) With three vertices, the longest path from the root can have length 1 or 2. There is only one tree of each type, so there are exactly two nonisomorphic rooted trees with 3 vertices, as shown below.



13. We find the answer by carefully enumerating these trees, i.e., drawing a full set of nonisomorphic trees. One way to organize this work so as to avoid leaving any trees out or counting the same tree (up to isomorphism) more than once is to list the trees by the length of their longest simple path (or longest simple path from the root in the case of rooted trees).

a) If the longest simple path has length 4, then the entire tree is just this path. If the longest simple path has length 3, then the fifth vertex must be attached to one of the middle vertices of this path. If the longest simple path has length 2, then the tree is just $K_{1,4}$. Thus there are only three trees with five vertices. They can be pictured as the first, second, and fourth pictures in the top row below.

b) For rooted trees of length 5, the longest path from the root can have length 1, 2, 3 or 4. There is only one tree with longest path of length 1 (the other four vertices are at level 1), and only one with longest path of length 4. If the longest path has length 3, then the fifth vertex (after using four vertices to draw this path) can be "attached" to either the root or the vertex at level 1 or the vertex at level 2, giving us three nonisomorphic trees. If the longest path has length 2, then there are several possibilities for where the fourth and fifth vertices can be "attached." They can both be adjacent to the root; they can both be adjacent to the vertex at level 1; one can be adjacent to the root and the other to the vertex at level 1; or one can be adjacent to the root and the other to this vertex: in all there are four possibilities in this case. Thus there are a total of nine nonisomorphic rooted trees on 5 vertices, as shown below.



15. a) We will prove this statement using mathematical induction on n, the number of vertices of G. (This exercise can also be done by using Exercise 14 and Theorem 2. Such a proof is given in the answer section of the textbook.) If $n = 1$, then there is only one possibility for G, it *is* a tree, it is connected, and it has $1 - 1 = 0$ edges. Thus the statement is true. Now let us assume that the statement is true for simple graphs with n vertices, and let G be a simple graph with $n + 1$ vertices.

There are two things to prove here. First let us suppose that G is a tree; we must show that G is connected and has $(n + 1) - 1 = n$ edges. Of course G is connected by definition. In order to prove that G

has the required number of edges, we need the following fact: a tree with at least one edge must contain a vertex of degree 1. (To see that this is so, let P be a simple path of greatest possible length; since the tree has no simple circuits, such a maximum length simple path exists. The ends of this path must be vertices of degree 1, since otherwise the simple path could be extended.) Let v be a vertex of degree 1 in G, and let G' be G with v and its incident edge removed. Now G' is still a tree: it has no simple circuits (since G had none) and it is still connected (the removed edge is clearly not needed to form paths between vertices different from v). Therefore by the inductive hypothesis, G', which has n vertices, has $n-1$ edges; it follows that G, which has one more edge than G', has n edges.

Conversely, suppose that G is connected and has n edges. If G is not a tree, then it must contain a simple circuit. If we remove one edge from this simple circuit, then the resulting graph (call it G') is still connected. If G' is a tree then we stop; otherwise we repeat this process. Since G had only finitely many edges to begin with, this process must eventually terminate at some tree T with $n+1$ vertices (T has all the vertices that G had). By the paragraph above, T therefore has n edges. But this contradicts the fact that we removed at least one edge of G in order to construct T. Therefore our assumption that G was not a tree is wrong, and our proof is complete.

b) For the "only if" direction, suppose that G is a tree. By part **(a)**, G has $n-1$ edges, and by definition, G has no simple circuits. For the "if" direction, suppose that G has no simple circuits and has $n-1$ edges. The only thing left to prove is that G is connected. Let c equal the number of components of G, each of which is necessarily a tree, say with n_i vertices, where $\sum_{i=1}^{c} n_i = n$. By part **(a)**, the total number of edges in G is $\sum_{i=1}^{c}(n_i - 1) = n - c$. Since we are given that this equals $n-1$, it follows that $c = 1$, i.e., G is connected.

17. Since a tree with n vertices has $n-1$ edges, the answer is 9999.

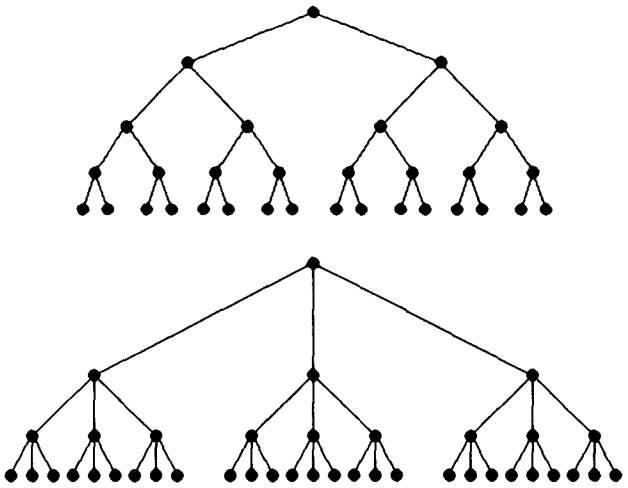
19. Each internal vertex has exactly 2 edges leading from it to its children. Therefore we can count the edges by multiplying the number of internal vertices by 2. Thus there are $2 \cdot 1000 = 2000$ edges.

21. We can model the tournament as a full binary tree. Each internal vertex represents the winner of the game played by its two children. There are 1000 leaves, one for each contestant. The root is the winner of the entire tournament. By Theorem 4(*iii*), with $m = 2$ and $l = 1000$, we see that $i = (l-1)/(m-1) = 999$. Thus exactly 999 games must be played to determine the champion.

23. Let P be a person sending out the letter. Then 10 people receive a letter with P's name at the bottom of the list (in the sixth position). Later 100 people receive a letter with P's name in the fifth position. Similarly, 1000 people receive a letter with P's name in the fourth position, and so on, until 1,000,000 people receive the letter with P's name in the first position. Therefore P should receive \$1,000,000. The model here is a full 10-ary tree.

25. No such tree exists. Suppose it did. By Theorem 4(*iii*), we know that a tree with these parameters must have $i = 83/(m-1)$ internal vertices. In order for this to be a whole number, $m-1$ must be a divisor of 83. Since 83 is prime, this means that $m = 2$ or $m = 84$. If $m = 2$, then we can have at most 15 vertices in all (the root, two at level 1, four at level 2, and eight at level 3). So m cannot be 2. If $m = 84$, then $i = 1$, which tells us that the root is the only internal vertex, and hence the height is only 1, rather than the desired 3. These contradictions tell us that no tree with 84 leaves and height 3 exists.

27. The complete binary tree of height 4 has 5 rows of vertices (levels 0 through 4), with each vertex not in the bottom row having two children. The complete 3-ary tree of height 3 has 4 rows of vertices (levels 0 through 3), with each vertex not in the bottom row having three children.



29. For both parts we use algebra on the equations $n = i + l$ (which is true by definition) and $n = mi + 1$ (which is proved in Theorem 3).

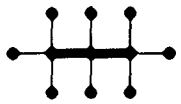
a) That $n = mi + 1$ is one of the given equations. For the second equality here, we have $l = n - i = (mi + 1) - i = (m - 1)i + 1$.

b) If we subtract the two given equations, then we obtain $0 = (1-m)i + (l-1)$, or $(m-1)i = l-1$. It follows that $i = (l-1)/(m-1)$. Then $n = i+l = [(l-1)/(m-1)]+l = (l-1+lm-l)/(m-1) = (lm-1)/(m-1)$.

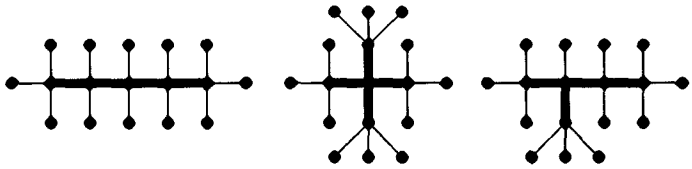
31. In each of the t trees, there is one fewer edge than there are vertices. Therefore altogether there are t fewer edges than vertices. Thus there are $n - t$ edges.

33. The number of isomers is the number of nonisomorphic trees with the given numbers of atoms. Since the hydrogen atoms play no role in determining the structure (they simply are attached to each carbon atom in sufficient number to make the degree of each carbon atom exactly 4), we need only look at the trees formed by the carbon atoms. In drawing our answers, we will show the tree of carbon atoms in heavy lines, with the hydrogen atom attachments in thinner lines.

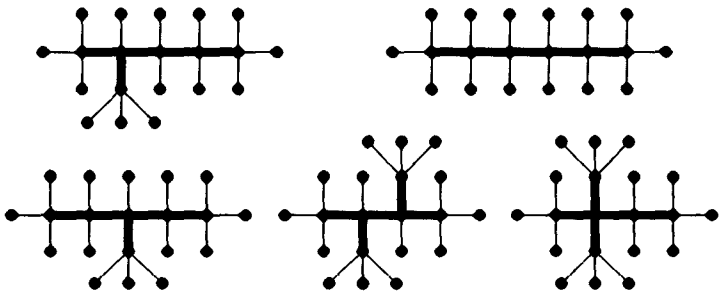
a) There is only one tree with three vertices (up to isomorphism), the path of length 2. Thus the answer is 1. The heavy lines in this diagram of the molecule form this tree.



b) There are 3 nonisomorphic trees with 5 vertices: the path of length 4, the "star" $K_{1,4}$, and the tree that consists of a path of length 3 together with one more vertex attached to one of the middle vertices in the path. Thus the answer is 3. Again the heavy lines in the diagrams of the molecules form these trees.



c) We need to find all the nonisomorphic trees with 6 vertices, except that we must not count the (one) tree with a vertex of degree 5 (since each carbon can only be attached to four other atoms). The complete set of trees is shown below (the heavy lines in these diagrams). Thus the answer is 5.



35. **a)** The parent of a vertex v is the directory in which the file or directory represented by v is contained.

b) The child of a vertex v (and v must represent a directory) is a file or directory contained in the directory that v represents.

c) If u and v are siblings, then the files or directories that u and v represent are in the same directory.

d) The ancestors of vertex v are all directories in the path from the root directory to the file or directory represented by v.

e) The descendants of a vertex v are all the files and directories either contained in v, or contained in directories contained in v, etc.

f) The level of a vertex v tells how far from the root directory is the file or directory represented by v.

g) The height of the tree is the greatest depth (i.e., level) at which a file or directory is buried in the system.

37. Suppose that $n = 2^k$, where k is a positive integer. We want to show how to add n numbers in $\log n$ steps using a tree-connected network of $n-1$ processors (recall that $\log n$ means $\log_2 n$). Let us prove this by mathematical induction on k. If $k = 1$ there is nothing to prove, since then $n = 2$ and $n - 1 = 1$, and certainly in $\log 2 = 1$ step we can add 2 numbers with 1 processor. Assume the inductive hypothesis, that we can add $n = 2^k$ numbers in $\log n$ steps using a tree-connected network of $n-1$ processors. Suppose now that we have $2n = 2^{k+1}$ numbers to add, $x_1, x_2, \ldots, x_{2n}$. The tree-connected network of $2n-1$ processors consists of the tree-connected network of $n-1$ processors together with two new processors as children of each leaf in the $(n-1)$-processor network. In one step we can use the leaves of the larger network to add $x_1 + x_2$, $x_3 + x_4$, $\ldots$, $x_{2n-1} + x_{2n}$. This gives us n numbers. By the inductive hypothesis we can now use the rest of the network to add these numbers using $\log n$ steps. In all, then, we used $1 + (\log n)$ steps, and, just as desired, $\log(2n) = \log 2 + \log n = 1 + \log n$. This completes the proof.

39. We need to compute the eccentricity of each vertex in order to find the center or centers. In practice, this does not involve much computation, since we can tell at a glance when the eccentricity is large. Intuitively, the center or centers are near the "middle" of the tree. The eccentricity of vertex c is 3, and it is the only vertex with eccentricity this small. Indeed, vertices a and b have eccentricities 4 and 5 (look at the paths to l); vertices d, f, g, j, and k all have eccentricities at least 4 (again look at the paths to l); and vertices e, h, i, and l also all have eccentricities at least 4 (look at the paths to k). Therefore c is the only center.

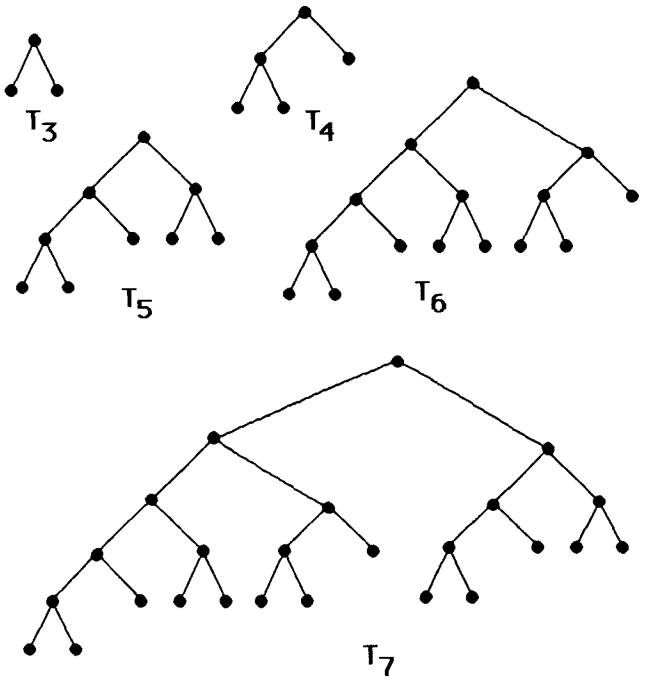
41. See the comments for the solution to Exercise 39. The eccentricity of vertices c and h are both 3. The eccentricities of the other vertices are all at least 4. Therefore c and h are the centers.

43. Certainly a tree has at least one center, since the set of eccentricities has a minimum value. First we prove that if u and v are any two distinct centers (say with minimum eccentricity e), then u and v are adjacent. Let P be the unique simple path from u to v. We will show that P is just u, v. If not, let c be any other vertex on P. Since the eccentricity of c is at least e, there is a vertex w such that the unique simple path Q from c to w has length at least e. This path Q may follow P for awhile, but once it diverges from P it cannot rejoin P without there being a simple circuit in the tree. In any case, Q cannot follow P towards

both u and v, so suppose without loss of generality that it does not follow P towards u. Then the path from u to c and then on to w is simple and of length greater than e, a contradiction. Thus no such c exists, and u and v are adjacent.

Finally, to see that there can be no more than two centers, note that we have just proved that every two centers are adjacent. If there were three (or more) centers, then we would have a K_3 contained in the tree, contradicting the definition that a tree has no simple circuits.

45. We follow the recursive definition and produce the following pictures for T_3 through T_7 (of course T_1 and T_2 are both the tree with just one vertex). For example, T_3 has T_2 (a single vertex) as its left subtree and T_1 (again a single vertex) as its right subtree.

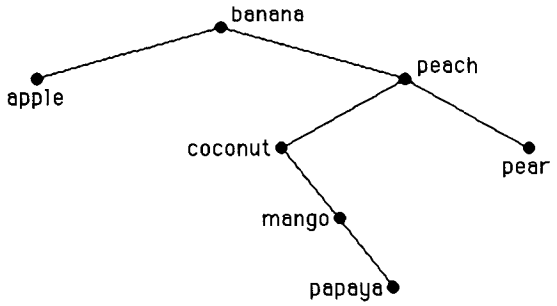


47. This "proof" shows that *there exists* a tree with n vertices having a path of length $n-1$. Note that the inductive step correctly takes the tree whose existence is guaranteed by the inductive hypothesis and correctly constructs a tree of the desired type. However, the statement was that *every* tree with n vertices has a path of length $n-1$, and this was not shown. A proof of the inductive step would need to start with an arbitrary tree with $n+1$ vertices and show that it had the required path. Of course no such proof is possible, since the statement is not true. Douglas West, whose *Introduction to Graph Theory* is an excellent book on that subject, calls this mistake the induction trap.

SECTION 11.2 Applications of Trees

Trees find many applications, especially in computer science. This section and subsequent ones deal with some of these applications. Binary search trees can be built up by adding new vertices one by one; searches in binary search trees are accomplished by moving down the tree until the desired vertex is found, branching either right or left as necessary. Huffman codes provide efficient means of encoding text in which some symbols occur more frequently than others; decoding is accomplished by moving down a binary tree. The coin-weighing problems presented here are but a few of the questions that can be asked. Try making up some of your own and answering them; it is easy to ask quite difficult questions of this type.

1. We first insert *banana* into the empty tree, giving us the tree with just a root, labeled *banana*. Next we insert *peach*, which, being greater than *banana* in alphabetical order, becomes the right child of the root. We continue in this manner, letting each new word find its place by coming down the tree, branching either right or left until it encounters a free position. The final tree is as shown.



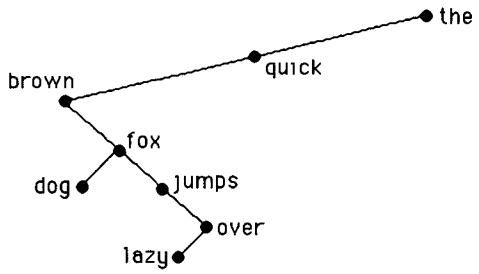
3. **a)** To find *pear*, we compare it with the root (*banana*), then with the right child of the root (*peach*), and finally with the right child of that vertex (*pear*). Thus 3 comparisons are needed.

b) Only 1 comparison is needed, since the item being searched for is the root.

c) We fail to locate *kumquat* by comparing it successively to *banana*, *peach*, *coconut*, and *mango*. Once we determine that *kumquat* should be in the left subtree of *mango*, and find no vertices there, we know that *kumquat* is not in the tree. Thus 4 comparisons were used.

d) This one is similar to part **(c)**, except that 5 comparisons are used. We compare *orange* successively to *banana*, *peach*, *coconut*, *mango*, and *papaya*.

5. We follow exactly the same procedure as in Exercise 1. The only unusual point is that the word "the" appears later in the sentence, after it is already in the tree. The algorithm finds that it is already in the tree, so it is not inserted again. The tree is shown below.



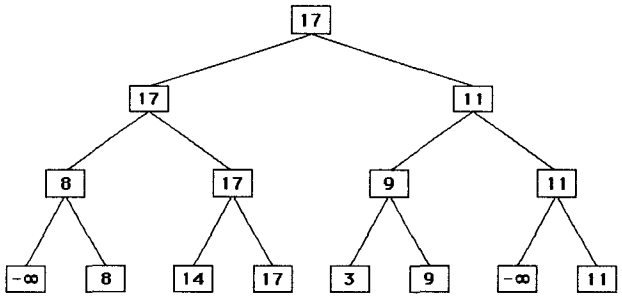
7. Since there are 4 different outcomes to the testing procedure, we need at least 2 weighings, since one weighing will give us only 3 possible outcomes (a decision tree of height 1 has only 3 leaves). Here is how to find the counterfeit coin with 2 weighings. Let us call the coins A, B, C, and D. First compare coins A and B. If they balance, then the counterfeit is among the other two. In this case, compare C with A; if they balance, then D is counterfeit; if they do not, then C is counterfeit. On the other hand if A and B do not balance, then one of them is the counterfeit. Again compare C with A. If they balance, then B is the counterfeit; if they do not, then A is counterfeit.

9. Since there are 12 different outcomes to the testing procedure, we need at least 3 weighings, since 2 weighings will only give us 9 possible outcomes (a decision tree of height 2 has only 9 leaves). Here is one way to find the counterfeit coin with 3 weighings. Divide the coins into three groups of 4 coins each, and compare two of the groups. If they balance, then the counterfeit is among the other four coins. If they do not balance, then

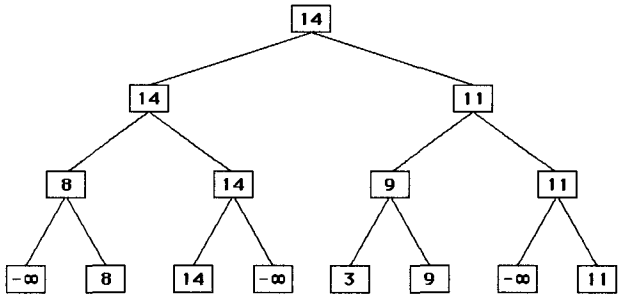
the counterfeit is among the four coins registering lighter. In either case we have narrowed the search to 4 coins. Now by Example 3 we can, in 2 more weighings, find the counterfeit among these four coins and four of the good ones.

11. By Theorem 1 in this section, at least $\lceil \log 4! \rceil$ comparisons are needed. Since $\log_2 24 \approx 4.6$, at least five comparisons are required. We can accomplish the sorting with five comparisons as follows. (This is essentially just merge sort, as discussed in Section 5.4.) Call the elements a, b, c, and d. First compare a and b; then compare c and d. Without loss of generality, let us assume that $a < b$ and $c < d$. (If not, then relabel the elements after these comparisons.) Next we compare a and c (this is our third comparison). Whichever is smaller is the smallest element of the set. Again without loss of generality, suppose $a < c$. Now we merely need to compare b with both c and d to completely determine the ordering. This takes two more comparisons, giving us the desired five in all.

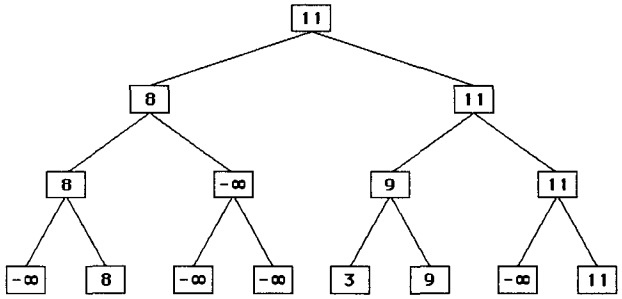
13. The first two steps are shown in the text. After 22 has been identified as the second largest element, we replace the leaf 22 by $-\infty$ in the tree and recalculate the winner in the path from the leaf where 22 used to be up to the root. The result is as shown here.



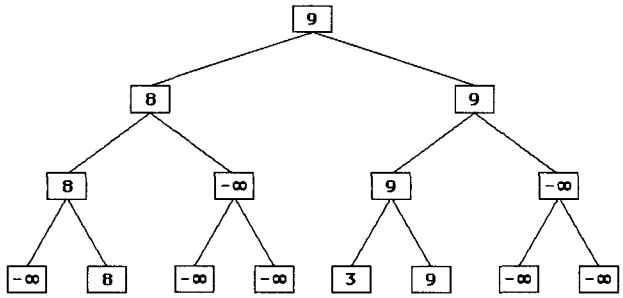
Now we see that 17 is the third largest element, so we repeat the process: replace the leaf 17 by $-\infty$ and recalculate. This gives us the following tree.



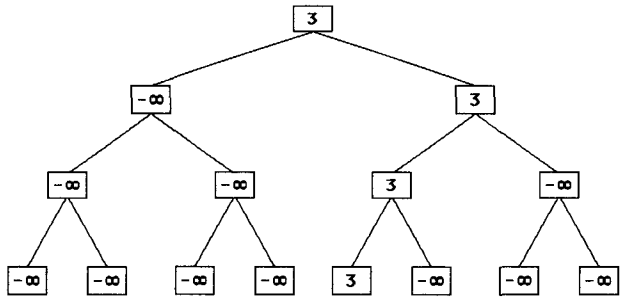
Thus we see that 14 is the fourth largest element, so we repeat the process: replace the leaf 14 by $-\infty$ and recalculate. This gives us the following tree.



Thus we see that 11 is the fifth largest element, so we repeat the process: replace the leaf 11 by $-\infty$ and recalculate. This gives us the following tree.



The process continues in this manner. The final tree will look like this, as we determine that 3 is the eighth largest element.



15. The heart of the matter is how to do the comparisons to work our way up the tree. We assume a data structure that allows us to access the left and right child of each vertex, as well as identify the root. We also need to keep track of which leaf is the winner of each contest so as to be able to find that leaf in one step. Each vertex of the tree, then, will have a value and a label; the value is the list element currently there, and the label is the name (i.e., location) of the leaf responsible for that value. We let $k = \lceil \log n \rceil$, which will be the height of the tree. The first part of the algorithm constructs the initial tree. The rest successively picks out the winners and recalculates the tree.

procedure *tournament sort*$(a_1, \ldots, a_n)$
$k := \lceil \log n \rceil$
build a binary tree of height k
for $i := 1$ **to** n
    set the value of the i^{th} leaf to be a_i and its label to be itself
for $i := n + 1$ **to** 2^k
    set the value of the i^{th} leaf to be $-\infty$ and its label to be itself
for $i := k - 1$ **downto** 0
    for each vertex v at level i
        set the value of v to the larger of the values of its children
        and its label to be the label of the child with the larger value
for $i := 1$ **to** n
    $c_i :=$ value at the root
    let v be the label of the root
    set the value of v to be $-\infty$
        while the label at the root is still v
            $v := parent(v)$
            set the value of v to the larger of the values of its children
            and its label to be the label of the child with the larger value
$\{c_1, \ldots, c_n$ is the list in nonincreasing order$\}$

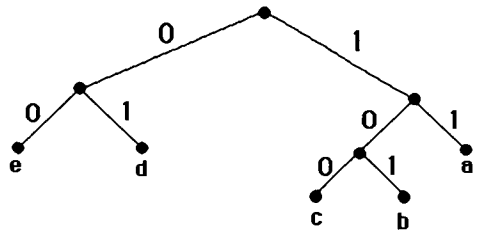
17. At each stage after the initial tree has been set up, only k comparisons are needed to recalculate the values at the vertices. (We are still assuming that $n = 2^k$ as in Exercise 16.) A leaf is replaced by $-\infty$, and, starting with that leaf's parent, one comparison is made for each vertex on the path up to the root. We can actually

get by with $k-1$ comparisons, because it does not take a comparison to determine that the new leaf's sibling beats it, so no comparison is needed for the new leaf's parent.

19. **a)** This is a prefix code, since no code is the first part of another.
b) This is not a prefix code, since, for instance, the code for a is the first part of the code for t.
c) This is a prefix code, since no code is the first part of another.
d) This is a prefix code, since no code is the first part of another.

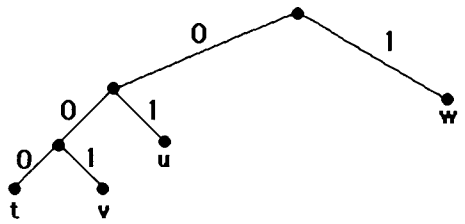
21. The code for a letter is simply the labels on the edges in the path from the root to that letter. Since the path from the root to a goes through the three edges leading left each time, all labeled 0, the code for a is 000. Similarly the codes for e, i, k, o, p and u are 001, 01, 1100, 1101, 11110 and 11111, respectively.

23. We follow Algorithm 2. Since b and c are the symbols of least weight, they are combined into a subtree, which we will call T_1 for discussion purposes, of weight $0.10 + 0.15 = 0.25$, with the larger weight symbol, c, on the left. Now the two trees of smallest weight are the single symbol a and either T_1 or the single symbol d (both have weight 0.25). We break the tie arbitrarily in favor of T_1, and so we get a tree T_2 with left subtree T_1 and right subtree a. (If we had broken the tie in the other way, our final answer would have been different, but it would have been just as correct, and the average number of bits to encode a character would be the same.) The next step is to combine e and d into a subtree T_3 of weight 0.55. And the final step is to combine T_2 and T_3. The result is as shown.



We see by looking at the tree that a is encoded by 11, b by 101, c by 100, d by 01, and e by 00. To compute the average number of bits required to encode a character, we multiply the number of bits for each letter by the weight of that latter and add. Since a takes 2 bits and has weight 0.20, it contributes 0.40 to the sum. Similarly b contributes $3 \cdot 0.10 = 0.30$. In all we get $2 \cdot 0.20 + 3 \cdot 0.10 + 3 \cdot 0.15 + 2 \cdot 0.25 + 2 \cdot 0.30 = 2.25$. Thus on the average, 2.25 bits are needed per character. Note that this is an appropriately weighted average, weighted by the frequencies with which the letters occur.

25. We proceed as in Exercise 23. The first step combines t and v (in either order—we have a choice here). At the second step we can combine this subtree either with u or with w. There are four possible answers in all, the one shown here and three more obtained from this one by swapping t and v, swapping u and w, or making both of these swaps.



27. This is a computationally intensive exercise. We will not show the final picture here, because it is too complex. However, we show the steps below (read row by row), with the obvious notation that we are joining the tree

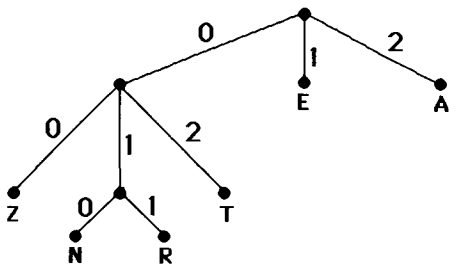
on the left of the plus sign to the tree on the right, to produce the new tree of a certain weight. The procedure is the same as in Exercise 23. (The sum of the frequencies was only 0.9999 due to round-off.)

$$\mathrm{Q} + \mathrm{Z} \rightarrow T_1(0.0014), \quad T_1 + \mathrm{J} \rightarrow T_2(0.0024), \quad T_2 + \mathrm{X} \rightarrow T_3(0.0039), \quad \mathrm{K} + T_3 \rightarrow T_4(0.0119)$$
$$T_4 + \mathrm{V} \rightarrow T_5(0.0221), \quad \mathrm{P} + \mathrm{B} \rightarrow T_6(0.0301), \quad \mathrm{F} + \mathrm{G} \rightarrow T_7(0.0391), \quad T_5 + \mathrm{Y} \rightarrow T_8(0.0432)$$
$$\mathrm{W} + \mathrm{C} \rightarrow T_9(0.0512), \quad T_6 + \mathrm{M} \rightarrow T_{10}(0.0578), \quad T_7 + \mathrm{U} \rightarrow T_{11}(0.0695), \quad \mathrm{D} + \mathrm{L} \rightarrow T_{12}(0.0828)$$
$$T_9 + T_8 \rightarrow T_{13}(0.0944), \quad T_{10} + \mathrm{R} \rightarrow T_{14}(0.1150), \quad \mathrm{N} + \mathrm{S} \rightarrow T_{15}(0.1290), \quad \mathrm{I} + \mathrm{H} \rightarrow T_{16}(0.1357)$$
$$\mathrm{O} + T_{11} \rightarrow T_{17}(0.1476), \quad T_{12} + \mathrm{A} \rightarrow T_{18}(0.1645), \quad T_{13} + \mathrm{T} \rightarrow T_{19}(0.1849)$$
$$\mathrm{E} + T_{14} \rightarrow T_{20}(0.2382), \quad T_{16} + T_{15} \rightarrow T_{21}(0.2647), \quad T_{18} + T_{17} \rightarrow T_{22}(0.3121)$$
$$T_{20} + T_{19} \rightarrow T_{23}(0.4231), \quad T_{22} + T_{21} \rightarrow T_{24}(0.5768), \quad T_{24} + T_{23} \rightarrow T_{25}(0.9999)$$

We show in the following table the resulting codes and the product of code length and frequency. Thus we see that the sum of the last column—the average number of bits required—is 4.2013. Note that this is slightly better than if we used five bits per letter (which we can do, since there are fewer than $2^5 = 32$ letters).

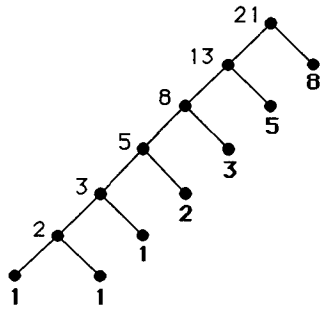
letter	code	length	frequency	product
A	0001	4	0.0817	0.3268
B	101001	6	0.0145	0.0870
C	11001	5	0.0248	0.1240
D	00000	5	0.0431	0.2155
E	100	3	0.1232	0.3696
F	001100	6	0.0209	0.1254
G	001101	6	0.0182	0.1092
H	0101	4	0.0668	0.2672
I	0100	4	0.0689	0.2756
J	110100101	9	0.0010	0.0090
K	1101000	7	0.0080	0.0560
L	00001	5	0.0397	0.1985
M	10101	5	0.0277	0.1385
N	0110	4	0.0662	0.2648
O	0010	4	0.0781	0.3124
P	101000	6	0.0156	0.0936
Q	1101001000	10	0.0009	0.0090
R	1011	4	0.0572	0.2288
S	0111	4	0.0628	0.2512
T	111	3	0.0905	0.2715
U	00111	5	0.0304	0.1520
V	110101	6	0.0102	0.0612
W	11000	5	0.0264	0.1320
X	11010011	8	0.0015	0.0120
Y	11011	5	0.0211	0.1055
Z	1101001001	10	0.0005	0.0050

29. Here $N = 6$ and $m = 3$, so $((N-1) \textbf{ mod } (m-1)) + 1 = 2$. Thus we start by combining the two symbols with smallest weights, **N** and **R**, into a subtree T_1 with weight 0.15. Next we combine the three items with smallest weights, namely **Z**, T_1, and **T**, into a subtree T_2 with weight 0.45. Finally we construct the tree, as shown here.



The codes, then, are as follows: A:2; E:1; N:010; R:011; T:02; Z:00.

31. We play around with this problem for small values of n, constructing the tree for the Huffman code, and we see that it will look like this (where $n = 6$).



The numbers in bold font are the frequencies of the symbols, and the numbers in plain font are the weights of the subtrees. Note that at each stage the tree constructed so far is paired with the next symbol. As we see from the picture (and can be proved rigorously by induction), the maximum number of bits used to encode a symbol is n.

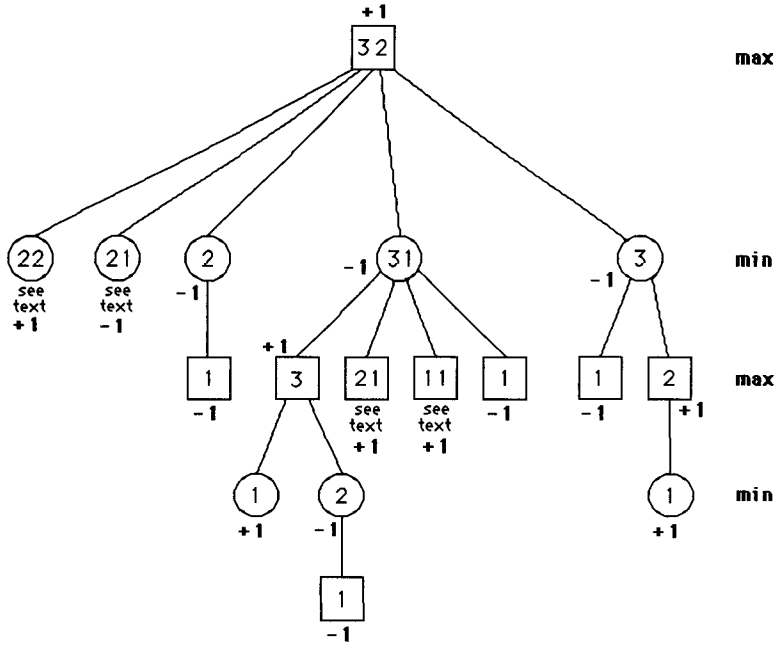
33. The procedure is similar to Examples 6 and 8. We draw the game tree, showing the positions within squares or circles. Since the tree is rather large, we have indicated in some places to "see text." Refer to Figure 9; the subtree rooted at these square or circle vertices is exactly the same as the corresponding subtree in Figure 9. Since the value of the root is $+1$, the first player wins the game following the optimal (minmax) strategy, by first moving to the position 22, and then leaving one pile with one stone at her second (final) move. (The figure is shown on the next page for spacing reasons.)

35. **a)** If we draw the game tree, then we see that the first player has only one winning move at her first play, namely to remove the three stones and win immediately. Thus the payoff is \$1.

b) If we draw the game tree, then we see that the first player has only one winning move at her first play, namely to remove two stones from the pile with four stones, leaving two piles of two stones each. Whatever the second player does, the first player must leave just one stone at her second move if she wants to win the game. Therefore the game will end in three moves, so the payoff to the first player is \$3.

c) If we draw the game tree we see that the first player is doomed to lose if the second player plays optimally. Since her payoff will be negative, she wants the game to end as quickly as possible. Therefore she should remove the pile of three stones, forcing her opponent to remove the pile of two stones if he wants to win the game. Therefore the game will end in two moves, so the payoff to the first player is $-\$3$.

NOTE: This is the figure for Exercise 33.



37. See the next page for the figures. Note that the second player (O) moves at the root of this subtree in each of parts **(a)** and **(b)**, and the first player (X) moves at the root of this subtree in part **(c)**. The squares and circles enclosing the positions have been suppressed for readability.

a) The value is 0, since the game must end in a draw.

b) The value is again 0, since O will choose the right branch.

c) The value is 1, since the first player will make her row of X's.

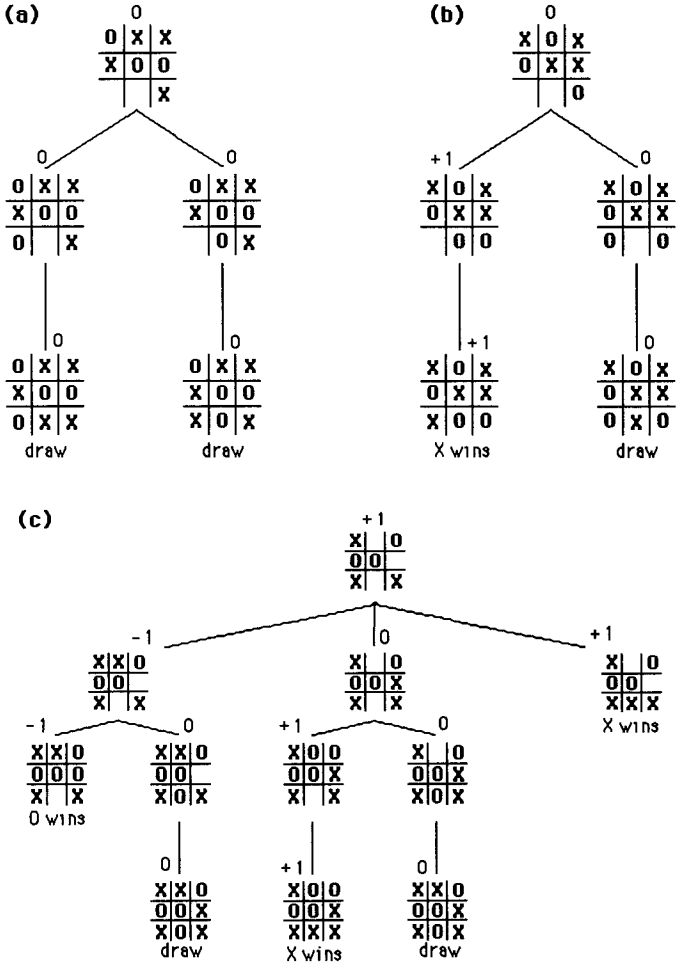
d) This is a trick question. This position cannot have occurred in a game. Note that X has three in a row, so the game was over at the point at which X made her third move. Therefore O did not make a third move, and this picture is impossible.

39. We prove this by strong induction. For the basis step, when $n = 2$ stones are in each pile, the first player is up a creek. If she takes two stones from a pile, then the second player takes one stone from the remaining pile and wins. If she takes one stone from a pile, then the second player takes two stones from the other pile and again wins. Assume the inductive hypothesis that the second player can always win if the game starts with two piles of j stones for all j between 2 and k, inclusive, where $k \geq 2$, and consider a game with two piles containing $k + 1$ stones each. If the first player takes all the stones from one of the piles, then the second player takes all but one stone from the remaining pile and wins. If the first player takes all but one stone from one of the piles, then the second player takes all the stones from the other pile and again wins. Otherwise the first player takes some stones from one of the piles, leaving j stones in that pile, where $2 \leq j \leq k$, and $k+1$ stones in the other pile. The second player then takes the same number of stones from the larger pile, also leaving j stones there. At this point the game consists of two piles of j stones each, where $2 \leq j \leq k$. By the inductive hypothesis, the second player in that game, who is also the second player in our actual game, can win, and the proof by strong induction is complete.

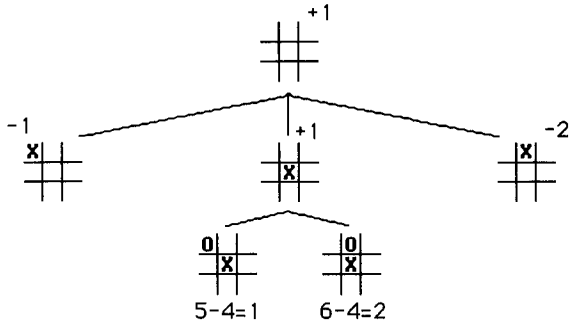
41. In the game of checkers, each player has 12 checkers (pieces), occupying the black squares nearest her in three rows on an 8×8 checkerboard. According to the rules, a checker may move to an adjacent black square diagonally, toward the other player. Thus only the front row of checkers can move at the start, and by looking at the board, we see that one of these four checkers has only one possible move, but the others each have two

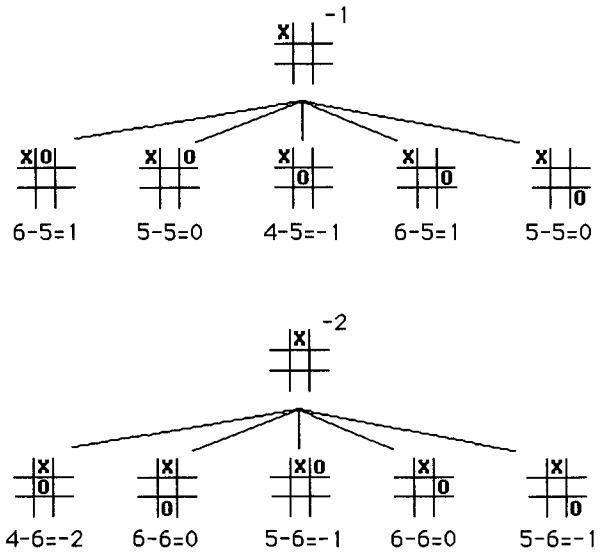
choices. This gives a total of seven moves for the first player, so the root of the game tree has seven children. For each move by the first player, the second player is free to move his front row in the same manner, so each vertex at Level 1 has seven children, giving $7 \cdot 7 = 49$ grandchildren of the root of this game tree.

NOTE: These are the figures for Exercise 37.



43. The game tree is too large to fit in one picture, so we show first the root, its children, and its grandchildren corresponding to a move that has only two responses (up to symmetry), and then the subtrees corresponding to the next move for O for each of the other two possible moves for X. See Figure 8 in the text. We label the second trees first, where we apply the evaluation function as described (calculation is shown), and assign the root of each subtree the minimum of the values of its children (again shown only up to symmetry). The value at the root of the tree is the maximum of the values of the three children, which is 1. Thus the first player, using this heuristic, should make her first move in the center.

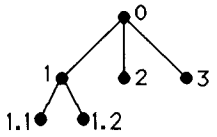




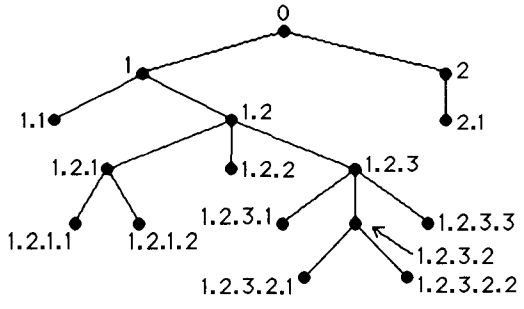
SECTION 11.3 Tree Traversal

Tree traversal is central to computer science applications. Trees are such a natural way to represent arithmetical and algebraic formulae, and so easy to manipulate, that it would be difficult to imagine how computer scientists could live without them. To see if you really understand the various orders, try Exercises 26 and 27. You need to make your mind work recursively for tree traversals: when you come to a subtree, you need to remember where to continue after processing the subtree. It is best to think of these traversals in terms of the recursive algorithms (shown as Algorithms 1, 2, and 3). A good bench-mark for testing your understanding of recursive definitions is provided in Exercises 30–34.

1. The root of the tree is labeled 0. The children of the root are labeled 1, 2, ..., from left to right. The children of a vertex labeled α are labeled $\alpha.1$, $\alpha.2$, ..., from left to right. For example, the two children of the vertex 1 here are 1.1 and 1.2. We completely label the tree in this manner, from the top down. See the figure. The lexicographic order of the labels is the preorder of the vertices: after each vertex come the subtrees rooted at its children, from left to right. Thus the order is $0 < 1 < 1.1 < 1.2 < 2 < 3$.



3. See the comments for the solution to Exercise 1. The order is $0 < 1 < 1.1 < 1.2 < 1.2.1 < 1.2.1.1 < 1.2.1.2 < 1.2.2 < 1.2.3 < 1.2.3.1 < 1.2.3.2 < 1.2.3.2.1 < 1.2.3.2.2 < 1.2.3.3 < 2 < 2.1$.



5. The given information tells us that the root has two children. We have no way to tell how many vertices are in the subtree of the root rooted at the first of these children. Therefore we have no way to tell how many vertices are in the tree.

7. In preorder, the root comes first, then the left subtree in preorder, then the right subtree in preorder. Thus the preorder is a, followed by the vertices of the left subtree (the one rooted at b) in preorder, then c. Recursively, the preorder in the subtree rooted at b is b, followed by d, followed by the vertices in the subtree rooted at e in preorder, namely e, f, g. Putting this all together, we obtain the answer a, b, d, e, f, g, c.

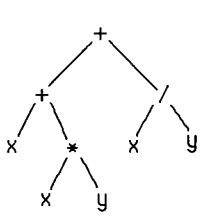
9. See the comments in the solution to Exercise 7 for the procedure. The only difference here is that some vertices have more than two children: after listing such a vertex, we list the vertices of its subtrees, in preorder, from left to right. The answer is $a, b, e, k, l, m, f, g, n, r, s, c, d, h, o, i, j, p, q$.

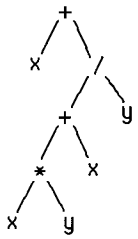
11. Inorder traversal requires that the left-most subtree be traversed first, then the root, then the remaining subtrees (if any) from left to right. Applying this principle, we see that the list must start with the left subtree in inorder. To find this, we need to start with *its* left subtree, namely d. Next comes the root of that subtree, namely b, and then the right subtree in inorder. This is i, followed by the root e, followed by the subtree rooted at j in inorder. This latter listing is m, j, n, o. We continue in this manner, ultimately obtaining: $d, b, i, e, m, j, n, o, a, f, c, g, k, h, p, l$.

13. In postorder, the root comes last, following the left subtree in postorder and the right subtree in postorder. Thus the postorder is the vertices of the left subtree (the one rooted at b) in postorder, then c, then a. Recursively, the postorder in the subtree rooted at b is d, followed by the vertices in the subtree rooted at e in postorder, namely f, g, e, followed by b. Putting this all together, we obtain the answer d, f, g, e, b, c, a.

15. This is just like Exercises 13 and 14. Note that all subtrees of a vertex are completed before listing that vertex. The answer is $k, l, m, e, f, r, s, n, g, b, c, o, h, i, p, q, j, d, a$.

17. **a)** For the first expression, we note that the outermost operation is the second addition. Therefore the root of the tree is this plus sign, and the left and right subtrees are the trees for the expressions being added. The first operand is the sum of x and xy, so the left subtree has a plus sign for its root and the tree for the expressions x and xy as its subtrees. We continue in this manner until we have drawn the entire tree. The second tree is done similarly. Note that the only difference between these two expressions is the placement of parentheses, and yet the expressions represent quite different operations, as can be seen from the fact that the trees are quite different.





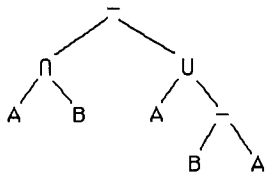
b) We can read off the answer from the picture we have just drawn simply by listing the vertices of the tree in preorder: First list the root, then the left subtree in preorder, then the right subtree in preorder. Therefore the answer is $++x*xy/xy$. Similarly, the second expression in prefix notation is $+x/+*xyxy$.

c) We can read off the answer from the picture we have just drawn simply by listing the vertices of the tree in postorder: First list the left subtree in postorder, then the right subtree in postorder, then the root. Therefore the answer is $xxy*+xy/+$. Similarly, the second expression in postfix notation is $xxy*x+y/+$.

d) The infix expression is just the given expression, fully parenthesized, with an explicit symbol for multiplication. Thus the first is $((x + (x * y)) + (x/y))$, and the second is $(x + (((x * y) + x)/y))$. This corresponds to traversing the tree in inorder, putting in a left parenthesis whenever we go down to a left child and putting in a right parenthesis whenever we come up from a right child.

19. This is similar to Exercise 17, with set operations rather than arithmetic ones.

a) We construct the tree in the same way we did there, noting, for example, that the first minus is the outermost operation.



b) The prefix expression is obtained by traversing the tree in preorder: $- \cap A\, B \cup A - B\, A$.

c) The postfix expression is obtained by traversing the tree in postorder: $A\, B \cap A\, B\, A - \cup -$.

d) This is already in fully parenthesized infix notation except for needing an outer set of parentheses: $((A \cap B) - (A \cup (B - A)))$.

21. Either of the four operators can be the outermost one, so there are four cases to consider. If the first operator is the outermost one, then we need to compute the number of ways to fully parenthesize $B - A \cap B - A$. Here there are 5 possibilities: 1 in which the "$\cap$" symbol is the outermost operator and 2 with each of the "$-$" symbols as the outermost operator. If the second operator in our original expression is the outermost one, then the only choice is in the parenthesization of the second of its operands, and there are 2 possibilities. Thus there are a total 7 ways to parenthesize this expression if either of the first two operators are the outermost one. By symmetry there are another 7 if the outermost operator is one of the last two. Therefore the answer to the problem is 14.

23. We show how to do these exercises by successively replacing the first occurrence of an operator immediately followed by two operands with the result of that operation. (This is an alternative to the method suggested in the text, where the *last* occurrence of an operator, which is necessarily preceded by two operands, is acted upon first.) The final number is the value of the entire prefix expression. In part **(a)**, for example, we first replace / 8 4 by the result of dividing 8 by 4, namely 2, to obtain $- * 2\ 2\ 3$. Then we replace $* 2\ 2$ by the result of multiplying 2 and 2, namely 4, to obtain the third line of our calculation. Next we replace $-\ 4\ 3$ by its answer, 1, which is the final answer.

a)

$$- * 2\ /\ 8\ 4\ 3$$
$$- * 2\ 2\ 3$$
$$-\ 4\ 3$$
$$1$$

b)

$$\uparrow - * 3\ 3 * 4\ 2\ 5$$
$$\uparrow - 9 * 4\ 2\ 5$$
$$\uparrow - 9\ 8\ 5$$
$$\uparrow 1\ 5$$
$$1$$

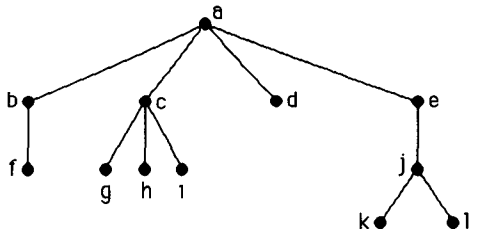
c)

$$+ - \uparrow 3\ 2 \uparrow 2\ 3 \,/\, 6 - 4\ 2$$
$$+ - 9 \uparrow 2\ 3 \,/\, 6 - 4\ 2$$
$$+ - 9\ 8 \,/\, 6 - 4\ 2$$
$$+ 1 \,/\, 6 - 4\ 2$$
$$+ 1 \,/\, 6\ 2$$
$$+ 1\ 3$$
$$4$$

d)

$$* + 3 + 3 \uparrow 3 + 3\ 3\ 3$$
$$* + 3 + 3 \uparrow 3\ 6\ 3$$
$$* + 3 + 3\ 729\ 3$$
$$* + 3\ 732\ 3$$
$$* 735\ 3$$
$$2205$$

25. We slowly use the clues to fill in the details of this tree, shown below. Since the preorder starts with a, we know that a is the root, and we are told that a has four children. Next, since the first child of a comes immediately after a in preorder, we know that this first child is b. We are told that b has one child, and it must be f, which comes next in the preorder. We are told that f has no children, so we are now finished with the subtree rooted at b. Therefore the second child of a must be c (the next vertex in preorder). We continue in this way until we have drawn the entire tree.



27. We prove this by induction on the length of the list. If the list has just one element, then the statement is trivially true. For the inductive step, consider the end of the list. There we find a sequence of vertices, starting with the last leaf and ending with the root of the tree, each vertex being the last child of its successor in the list. We know where this sequence starts, since we are told the number of children of each vertex: it starts at the last leaf in the list. Now remove this leaf, and decrease the child count of its parent by 1. The result is the postorder and child counts of a tree with one fewer vertex. By the inductive hypothesis we can uniquely determine this smaller tree. Then we can uniquely determine where the deleted vertex goes, since it is the last child of its parent (whom we know).

29. In each case the postorder is c, d, b, f, g, h, e, a.

31. We prove this by induction on the recursive definition, in other words, on the length of the formula, i.e., the total number of symbols and operators. The only formula of length 1 arises from the base case of the recursive definition (part (i)), and in that case we have one symbol and no operators, so the statement is true. Assume

that the statement is true for formulae of length less than $n > 1$, and let F be a formula of length n. Then F arises from part (ii) of the definition, so F consists of $*XY$, for some operator $*$ and some formulae X and Y. By the inductive hypothesis, the number of symbols in X exceeds the number of operators there by 1, and the same holds for Y. If we add and note that there is one more operator in F than in X and Y combined, then we see that the number of symbols in F exceeds the number of operators in F by 1, as well.

33. Any string of length n, using these six characters, is a well-formed formula as long as two conditions are met: if we read the string from left to right, the number of symbols is always at least 1 greater than the number of operators; and in all there is one more symbol than operator. We are asked to write down six such strings, with $n \geq 7$. One such set is $xxxx+++$, $xxxxx++++$, $xx+xx++$, $xxxx+xx++++$, $xxx+xx+++$, and $xx+xxx+++$.

SECTION 11.4 Spanning Trees

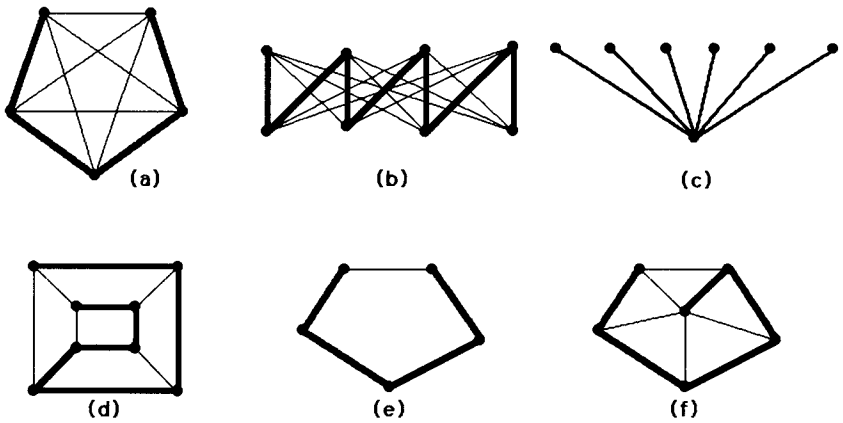
The spanning tree algorithms given here provide systematic methods for searching through graphs, and they are the foundation of many other, more complicated, algorithms. The concept of a spanning tree is quite simple and natural, of course: the problem comes with finding spanning trees efficiently. The reader should pay attention to the exercises on backtracking (Exercises 26–30) to get a feel both for the ideas behind it and for its inefficiency for large problems.

1. The graph has m edges. The spanning tree has $n-1$ edges. Therefore we need to remove $m-(n-1)$ edges.

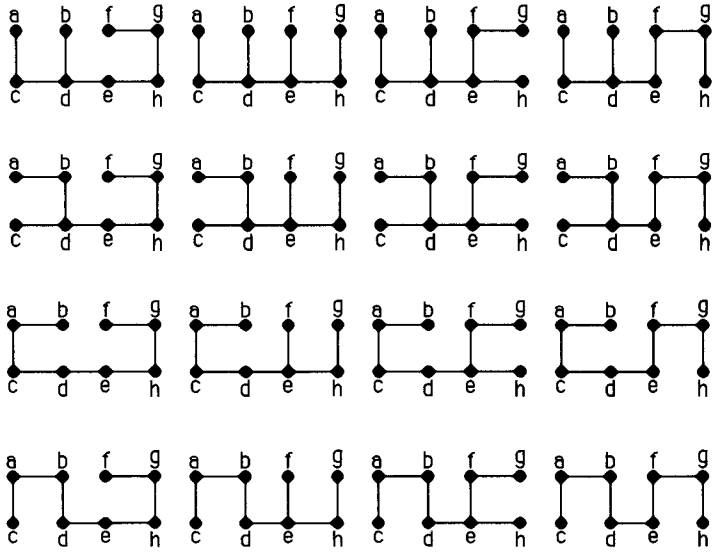
3. We have to remove edges, one at a time. We can remove any edge that is part of a simple circuit. The answer is by no means unique. For example, we can start by removing edge $\{a,d\}$, since it is in the simple circuit $adcba$. Then we might choose to remove edge $\{a,g\}$. We can continue in this way and remove all of these edges: $\{b,e\}$, $\{b,f\}$, $\{b,g\}$, $\{d,e\}$, $\{d,g\}$, and $\{e,g\}$. At this point there are no more simple circuits, so we have a spanning tree.

5. This is similar to Exercise 3. Here is one possible set of removals: $\{a,b\}$, $\{a,d\}$, $\{a,f\}$, $\{b,c\}$, $\{b,d\}$, $\{c,d\}$, $\{d,e\}$, $\{d,g\}$, $\{d,j\}$, $\{e,g\}$, $\{e,j\}$, $\{f,g\}$, $\{h,j\}$, $\{h,k\}$, and $\{j,l\}$. As a check, note that there are 12 vertices and 26 edges. A spanning tree must have 11 edges, so we need to remove 15 of them, as we did.

7. In each case we show the original graph, with a spanning tree in heavier lines. These were obtained by trial and error. In each case except part **(c)**, our spanning tree is a simple path (but other answers are possible). In part **(c)**, of course, the graph is its own spanning tree.



9. We can remove any one of the four edges in the square on the left, together with any one of the four edges in the square on the right. Therefore there are $4 \cdot 4 = 16$ different spanning trees, shown here.



11. We approach this problem in a rather ad hoc way.

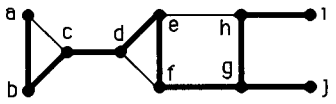
a) Every pair of edges in K_3 forms a spanning tree, so there are $C(3,2) = 3$ such trees.

b) There are 16 spanning trees; careful counting is required to see this. First, let us note that the trees can take only two shapes: the star $K_{1,3}$ and the simple path of length 3. There are 4 different spanning trees of the former shape, since any of the four vertices can be chosen as the vertex of degree 3. There are $P(4,4) = 24$ orders in which the vertices can be listed in a simple path of length 3, but since the path can be traversed in either of two directions to yield the same tree, there are only 12 trees of this shape. Therefore there are $4 + 12 = 16$ spanning trees of K_4 altogether.

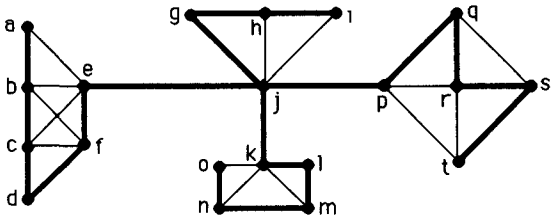
c) Note that $K_{2,2} = C_4$. A tree is determined simply by deciding which of the four edges to remove. Therefore there are 4 spanning trees.

d) By the same reasoning as in part **(c)**, there are 5 spanning trees.

13. If we start at vertex a and use alphabetical order, then the depth-first search spanning tree is unique. We start at vertex a and form the path shown in heavy lines to vertex i before needing to backtrack. There are no unreached vertices from vertex h at this point, but there is an unreached vertex (j) adjacent to vertex g. Thus the tree is as shown in heavy lines.



15. The procedure is the same as in Exercise 13. The spanning tree is shown in heavy lines.

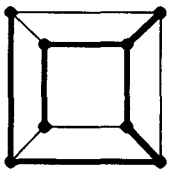


17. **a)** We start at the vertex in the middle of the wheel and visit a neighbor—one of the vertices on the rim. From there we move to an adjacent vertex on the rim, and so on all the way around until we have reached every vertex. Thus the resulting spanning tree is a path of length 6.

b) We start at any vertex, visit a neighbor, then a new neighbor, and so on until we have reached every vertex. Thus the resulting spanning tree is a path of length 5.

c) We start at a vertex in the part with four vertices, move to a vertex in the part with three vertices, then back to a vertex in the larger part, and so on. The resulting spanning tree is a path of length 6.

d) Depending on what order we choose to visit the vertices, we may or may not need to backtrack in doing this depth-first search. In most cases, the resulting tree is a path of length 7, but we could, for example, come up with the following tree.



19. With breadth-first search, the initial vertex is the middle vertex, and the n spokes are added to the tree as this vertex is processed. Thus the resulting tree is $K_{1,n}$. With depth-first search, we start at the vertex in the middle of the wheel and visit a neighbor—one of the vertices on the rim. From there we move to an adjacent vertex on the rim, and so on all the way around until we have reached every vertex. Thus the resulting spanning tree is a path of length n.

21. With breadth-first search, we fan out from a vertex of degree m to all the vertices of degree n as the first step. Next a vertex of degree n is processed, and the edges from it to all the remaining vertices of degree m are added. The result is what is called a "double star": a $K_{1,n-1}$ and a $K_{1,m-1}$ with their centers joined by an edge. With depth-first search, we travel back and forth from one partite set to the other until we can go no further. If $m = n$ or $m = n - 1$, then we get a path of length $m + n - 1$. Otherwise, the path ends while some vertices in the larger partite set have not been visited, so we back up one link in the path to a vertex v and then successively visit the remaining vertices in that set from v. The result is what is called a "broom": a path with extra pendant edges coming out of one end of the path (our vertex v).

23. The question is simply asking for a spanning tree of the graph shown. There are of course many such spanning trees. One that the airline would probably not like to choose is the tree that consists of the path Bangor, Boston, New York, Detroit, Chicago, Washington, Atlanta, St. Louis, Dallas, Denver, San Diego, Los Angeles, San Francisco, Seattle. All the other 18 flights, then, would be discontinued. This set of flights would not be useful to a New York to Washington flyer, for example, nor to one who wants to fly from Chicago to Seattle. A more practical approach would be to build the tree up from nothing, using key short flights, such as the one between Detroit and Chicago. After 13 such edges had been chosen, without creating any simple circuits, we would have the desired spanning tree, and the other flights would be discontinued.

25. We prove this statement by induction on the length of a shortest path from v to u. If this length is 0, then $v = u$, and indeed, v is at the root of the tree. Assume that the statement is true for all vertices w for which a shortest path to v has length n, and let u be a vertex for which a shortest path has length $n + 1$. Clearly u cannot be at a level less than $n + 1$, because then a shorter path would be evident in the tree itself. On the other hand, if we let w be the penultimate vertex in a shortest path from v to u of length $n + 1$, then by the inductive hypothesis, we know that w is at level n of the tree. Now when vertex w was being processed by the breadth-first search algorithm, either u was already in the tree (and therefore adjacent to a vertex at level at most n) or u was one of the vertices put into the tree adjacent to w. In either case, u is adjacent to a vertex at level at most n and therefore is at level at most $n + 1$.

27. Label the squares of the $n \times n$ chessboard with coordinates (i, j), where i and j are integers from 1 to n, inclusive.

a) For the 3×3 board, we start our search by placing a queen in square $(1,1)$. The only possibility for a queen in the second column is square $(3,2)$. Now there is no place to put a queen in the third column. Therefore we backtrack and try placing the first queen in square $(2,1)$. This time there is no place to put a queen in the second column. By symmetry, we need not consider the initial choice of a queen in square $(3,1)$ (it will be just like the situation for the queen in square $(1,1)$, turned upside down). Therefore we have shown that there is no solution.

b) We start by placing a queen in square $(1,1)$. The first place a queen might then reside in the second column is square $(3,2)$, so we place a queen there. Now the only free spot in the third column is $(5,3)$, the only free spot in the fourth column is $(2,4)$, and the only free spot in the fifth column is $(4,5)$. This gives us a solution. Note that we were lucky and did not need to backtrack at all to find this solution.

c) The portion of the decision tree corresponding to placing the first queen in square $(1,1)$ is quite large here, and it leads to no solution. For example, the second queen can be in any of the squares $(3,2)$, $(4,2)$, $(5,2)$, or $(6,2)$. If the second queen is in square $(3,2)$, then the third can be in squares $(5,3)$ or $(6,3)$. After several backtracks we find that there is no solution with one queen in square $(1,1)$. Next we try square $(2,1)$ for the first queen. After a few more backtracks, we are led to the solution in which the remaining queens are in squares $(4,2)$, $(6,3)$, $(1,4)$, $(3,5)$ and $(5,6)$.

29. Assume that the graph has vertices $v_1, v_2, \ldots, v_n$. In looking for a Hamilton circuit we may as well start building a path at v_1. The general step is as follows. We extend the path if we can, to a new vertex (or to v_1 if this will complete the Hamilton circuit) adjacent to the vertex we are at. If we cannot extend the path any further, then we backtrack to the last previous vertex in the path and try other untried extensions from that vertex. The procedure for Hamilton paths is the same, except that we have to try all possible starting vertices, and we do not allow a return to the starting vertex, stopping instead when we have a path of the right length.

31. We know that every component of the graph has a spanning tree. The union of these spanning trees is clearly a spanning forest for the graph, since it contains every vertex and two vertices in the same component are joined by a path in the spanning tree for that component.

33. First we claim that the spanning forest will use $n - c$ edges in all. To see this, let n_i be the number of vertices in the i^{th} component, for $i = 1, 2, \ldots, c$. The spanning forest uses $n_i - 1$ edges in that component. Therefore the spanning forest uses $\sum(n_i - 1) = \left(\sum n_i\right) - c = n - c$ edges in all. Thus we need to remove $m - (n - c)$ edges to form a spanning forest.

35. We execute the breadth-first search algorithm, starting with the vertex v_1 from which we wish to find the shortest path. Each vertex v is assigned a value, $L(v)$, which will be the length of a shortest path from v_1 to v. We initialize $L(v_1) = 0$, and as each vertex w is added to the tree in Algorithm 2, we set $L(w) = 1 + L(v)$.

37. We carry out the breadth-first search algorithm, marking each vertex as we encounter it. The set of vertices encountered (and all the edges joining those vertices) form one component. Once BFS has concluded we look for an unmarked vertex and repeat the process starting from an unmarked vertex; this gives us a second component. When there are no more unmarked vertices, we are done. This works just as well for DFS.

39. A connected simple graph has only one spanning tree if the graph is itself a tree (clearly its only spanning tree is itself in this case). On the other hand, if a connected simple graph is not a tree, then it has a simple circuit containing $k \geq 3$ edges, and one can find a spanning tree of the graph containing any $k - 1$ of these edges but not the other (see Exercise 23 in Section 11.5).

41. In effect we use the depth-first search algorithm on each component. In more detail, once that procedure wants to stop, have it search through the list of vertices in the graph to find one that is not yet in the forest. If there is such a vertex, then repeat the process starting from that vertex. Continue this until all the vertices have been included in the forest.

43. If an edge uv is not followed while we are processing vertex u during the depth-first search process, then it can only be the case that the vertex v had already been visited. There are two cases. If vertex v was visited after we started processing u, then, since we are not finished processing u yet, v must appear in the subtree rooted at u (and hence must be a descendant of u). On the other hand, if the processing of v had already begun before we started processing u, then why wasn't this edge followed at that time? It must be that we had not finished processing v, in other words, that we are still forming the subtree rooted at v, so u is a descendant of v, and hence v is an ancestor of u.

45. Certainly these two procedures produce the identical spanning trees if the graph we are working with is a tree itself, since in this case there is only one spanning tree (the whole graph). This is the only case in which that happens, however. If the original graph has any other edges, then by Exercise 43 they must be back edges and hence join a vertex to an ancestor or descendant, whereas by Exercise 34, they must connect vertices at the same level or at levels that differ by 1. Clearly these two possibilities are mutually exclusive. Therefore there can be no edges other than tree edges if the two spanning trees are to be the same.

47. Since the edges not in the spanning tree are not followed in the process, we can ignore them. Thus we can assume that the graph was a rooted tree to begin with. The basis step is trivial (there is only one vertex), so we assume the inductive hypothesis that breadth-first search applied to trees with n vertices have their vertices visited in order of their level in the tree and consider a tree T with $n+1$ vertices. The last vertex to be visited during breadth-first search of this tree, say v, is the one that was added last to the list of vertices waiting to be processed. It was added when its parent, say u, was being processed. We must show that v is at the lowest (bottom-most, i.e., numerically greatest) level of the tree. Suppose not; say vertex x, whose parent is vertex w, is at a lower level. Then w is at a lower level than u. Clearly v must be a leaf, since any child of v could not have been seen before v is seen. Consider the tree T' obtained from T by deleting v. By the inductive hypothesis, the vertices in T' must be processed in order of their level in T' (which is the same as their level in T, and the absence of v in T' has no effect on the rest of the algorithm). Therefore u must have been processed before w, and therefore v would have joined the waiting list before x did, a contradiction. Therefore v is at the bottom-most level of the tree, and the proof is complete.

49. We modify the pseudocode given in Algorithm 2 by initializing m to be 0 at the beginning of the algorithm, and adding the statements "$m := m + 1$" and "assign m to vertex v" after the statement that removes vertex v from L.

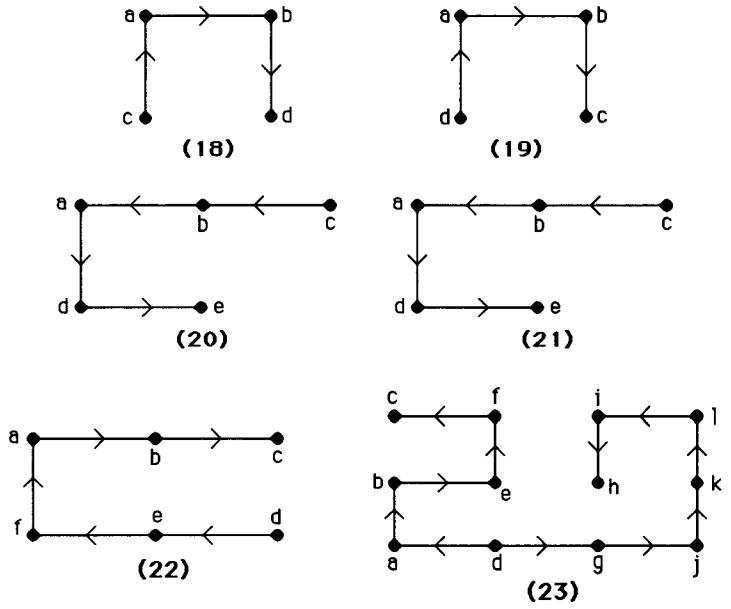
51. This is similar to Exercise 43. If a directed edge uv is not followed while we are processing its tail u during the depth-first search process, then it can only be the case that its head v had already been visited. There are three cases. If vertex v was visited after we started processing u, then, since we are not finished processing u yet, v must appear in the subtree rooted at u (and hence must be a descendant of u), so we have a forward edge. Otherwise, the processing of v must have already begun before we started processing u. If it had not yet finished (i.e., we are still forming the subtree rooted at v), then u is a descendant of v, and hence v is an ancestor of u (we have a back edge). Finally, if the processing of v had already finished, then by definition we have a cross edge.

53. There are five trees here, so there are $C(5,2) = 10$ questions. Let T be the tree in Figure 3c, and let T_1 through T_4 be the trees in Figure 4, reading from left to right. We will discuss one of the pairs at length and

simply report the other answers. Let $d(T, T_1)$ denote the distance between trees T and T_1. Note that tree T_1 has edges $\{a, e\}$, $\{c, g\}$, and $\{e, f\}$ that tree T does not. Since the trees have the same number of edges, there must also be 3 edges in T that are not in T_1 (we do not need to list them). Therefore $d(T, T_1) = 6$. Similarly we have $d(T, T_2) = 4$, $d(T, T_3) = 4$, $d(T, T_4) = 2$, $d(T_1, T_2) = 4$, $d(T_1, T_3) = 4$, $d(T_1, T_4) = 6$, $d(T_2, T_3) = 4$, $d(T_2, T_4) = 2$, and $d(T_3, T_4) = 4$.

55. Let $e_1 = \{u, v\}$. The graph $T_2 \cup \{e_1\}$ contains a (unique) simple circuit C containing edge e_1. Now $T_1 - \{e_1\}$ has two components, one of which contains u and the other of which contains v. We travel along the circuit C, starting at u and not using edge e_1 first, until we first reach a vertex in the component of $T_1 - \{e_1\}$ that contains v; obviously we must reach such a vertex eventually, since we eventually reach v itself. The edge we last traversed is e_2. Clearly $T_2 \cup \{e_1\} - \{e_2\}$ is a tree, since e_2 is on C. On the other hand, $T_1 - \{e_1\} \cup \{e_2\}$ is also a tree, since e_2 reunites the two components of $T_1 - \{e_1\}$.

57. Rooted spanning trees are easy to find in all six figures, as these pictures show. There are of course many other possible correct answers. In the first five cases the tree is a path. In the last case, the root is d.



59. By Exercise 16 in Section 10.5, we know that such a directed graph has an Euler circuit. Now we traverse the Euler circuit, starting at some vertex v (which will be our root), and delete from the circuit every edge that has as its terminal vertex a vertex we have already visited on this traversal. The graph that remains is a rooted spanning tree; there is a path from the root to every other vertex, and there can be no simple circuits.

61. According to Exercise 60, a directed graph contains a circuit if and only if there are any back edges. We can detect back edges as follows. Add a marker on each vertex v to indicate what its status is: not yet seen (the initial situation), seen (i.e., put into T) but not yet finished (i.e., *visit*(v) has not yet terminated), or finished (i.e., *visit*(v) has terminated). A few extra lines in Algorithm 1 will accomplish this bookkeeping. Then to determine whether a directed graph has a circuit, we just have to check when looking at edge uv whether the status of v is "seen." If that ever happens, then we know there is a circuit; if not, then there is no circuit.

SECTION 11.5 Minimum Spanning Trees

The algorithms presented here are not hard, once you understand them. The two algorithms are almost identical, the only real difference being in the set of edges available for inclusion in the tree at each step. In Prim's algorithm, only those edges that are adjacent to edges already in the tree (and not completing simple circuits) may be added (so that, as a result, all the intermediate stages are trees). In Kruskal's algorithm, any edge that does not complete a simple circuit may be added (so that the intermediate stages may be forests and not trees). The reader might try to discover other methods for finding minimum spanning trees, in addition to the ones in this section.

1. We want a minimum spanning tree in this graph. We apply Kruskal's algorithm and pave the following edges: Oasis to Deep Springs, Lida to Gold Point, Lida to Goldfield, Silver Peak to Goldfield, Oasis to Dyer, Oasis to Silver Peak, Manhattan to Tonopah, Goldfield to Tonopah, Gold Point to Beatty, and Tonopah to Warm Springs. At each stage we chose the minimum weight edge whose addition did not create a simple circuit (Kruskal's algorithm).

3. We start with the minimum weight edge $\{e, f\}$. The least weight edges incident to the tree constructed so far are edges $\{c, f\}$ and $\{e, h\}$, each with weight 3, so we add one of them to the tree (we will break ties using alphabetical order, so we add $\{c, f\}$). Next we add edge $\{e, h\}$, and then edge $\{h, i\}$, which has a smaller weight but has just become eligible for addition. The edges continue to be added in the following order (note that ties are broken using alphabetical order): $\{b, c\}$, $\{b, d\}$, $\{a, d\}$, and $\{g, h\}$. The total weight of the minimum spanning tree is 22.

5. Kruskal's algorithm will have us include first the links from Atlanta to Chicago, then Atlanta to New York, then Denver to San Francisco (the cheapest links). The next cheapest link, from Chicago to New York, cannot be included, since it would form a simple circuit. Therefore we next add the link from Chicago to San Francisco, and our network is complete.

7. The edges are added in the following order (with Kruskal's algorithm, we add at each step the shortest edge that will not complete a simple circuit): $\{e, f\}$, $\{a, d\}$, $\{h, i\}$, $\{b, d\}$, $\{c, f\}$, $\{e, h\}$, $\{b, c\}$, and $\{g, h\}$. The total weight of the minimum spanning tree is 22.

9. A graph with one edge obviously cannot be the solution, and a graph with two edges cannot either, since a simple connected graph with two edges must be a tree. On the other hand, if we take a triangle (K_3) and weight all the edges equally, then clearly there are three different minimum spanning trees.

11. If we simply replace each of the occurrences of the word "minimum" with the word "maximum" in Algorithm 1, then the resulting algorithm will find a maximum spanning tree.

13. We use an analog of Kruskal's algorithm, adding at each step an edge of greatest weight that does not create a simple circuit. The answer here is unique. It uses edges $\{a, c\}$, $\{b, d\}$, $\{b, e\}$, and $\{c, e\}$.

15. There are numerous possible answers. One uses edges $\{a, d\}$, $\{b, f\}$, $\{c, g\}$, $\{d, p\}$, $\{e, f\}$, $\{f, j\}$, $\{g, k\}$, $\{h, l\}$, $\{i, j\}$, $\{i, m\}$, $\{j, k\}$, $\{j, n\}$, $\{k, l\}$, $\{k, o\}$, and $\{o, p\}$, obtained by choosing at each step an edge of greatest weight that does not create a simple circuit.

17. If we want a second "shortest" spanning tree (which may, of course, have the same weight as the "shortest" tree), then we need to use at least one edge not in some minimum spanning tree T that we have found. One way to force this is, for each edge e of T, to apply a minimum spanning tree algorithm to the graph with e deleted, and then take a tree of minimum weight among all of these. It cannot equal T, and so it must be a second "shortest" spanning tree.

19. The proof that Prim's algorithm works shows how to take any minimum spanning tree T and, if T is not identical to the tree constructed by Prim's algorithm, to find another minimum spanning tree with even more edges in common with the Prim tree than T has. The core of the proof is in the last paragraph, where we add an edge e_{k+1} to T and delete an edge e. Now if all the edges have different weights, then the result of this process is not another minimum spanning tree but an outright contradiction. We conclude that there are no minimum spanning trees with any edges not in common with the Prim tree, i.e., that the only minimum spanning tree is the Prim tree.

21. We simply apply Kruskal's algorithm, starting not from the empty tree but from the tree containing these two edges. We can add the following edges to form the desired tree: $\{c,d\}$, $\{k,l\}$, $\{b,f\}$, $\{c,g\}$, $\{a,b\}$, $\{f,j\}$, $\{a,e\}$, $\{g,h\}$, and $\{b,c\}$.

23. The algorithm is identical to Kruskal's algorithm (Algorithm 2), except that we replace the statement "$T :=$ empty graph" by the assignment to T initially of the specified set of edges, and, instead of iterating from 1 to $n-1$, we iterate from 1 to $n-1-s$, where s is the number of edges in the specified set. It is assumed that the specified set of edges forms no simple circuits.

25. **a)** First we need to find the least expensive edges incident to each vertex. These are the links from New York to Atlanta, Atlanta to Chicago, and Denver to San Francisco. The algorithm tells us to choose all of these edges. At the end of this first pass, then, we have a forest of two trees, one containing the three eastern cities, the other containing the two western cities. Next we find the least expensive edge joining these two trees, namely the link from Chicago to San Francisco, and add it to our growing forest. We now have a spanning tree, and the algorithm has finished. Note, incidentally, that this is the same spanning tree that we obtained in Example 1; by the result of Exercise 19, since the weights in this graph are all different, there was only one minimum spanning tree.

b) On the first pass, we choose all the edges that are the minimum weight edges at each vertex. This set consists of $\{a,b\}$, $\{b,f\}$, $\{c,d\}$, $\{a,e\}$, $\{c,g\}$, $\{g,h\}$, $\{i,j\}$, $\{f,j\}$, and $\{k,l\}$. At this point the forest has three components. Next we add the lowest weight edges connecting these three components, namely $\{h,l\}$ and $\{b,c\}$, to complete our tree.

27. Let $e_1, e_2, \ldots, e_{n-1}$ be the edges of the tree S chosen by Sollin's algorithm in the order chosen (arbitrarily order the edges chosen at the same stage). Let T be a minimum spanning tree that contains all the edges $e_1, e_2, \ldots, e_k$ for as large a k as possible. Thus $0 \leq k \leq n-1$. If $k = n-1$, then $S = T$ and we have shown that S is a minimum spanning tree. Otherwise we will construct another minimum spanning tree T' which contains edges $e_1, e_2, \ldots, e_k, e_{k+1}$, contradicting the choice of T and completing the proof.

Let S' be the forest constructed by Sollin's algorithm at the stage before e_{k+1} is added to S. Let u be the endpoint of e_{k+1} that is in a component C of S' responsible for the addition of e_{k+1} (i.e., so that e_{k+1} is the minimum weight edge incident to C). Let v be the other endpoint of e_{k+1}. We let P be the unique simple path from u to v in T. We follow P until we come to the first edge e' not in S'. Thus e' is also incident to C. Since the algorithm chose to add e_{k+1} on behalf of C, we know that $w(e_{k+1}) \leq w(e')$, and that e' was added on behalf of the component of its other endpoint. Now if e' is not in $\{e_1, \ldots, e_k\}$, then we go on to the next paragraph. Otherwise, we continue following P until we come to the first edge e'' not in S'. Thus e'' is also incident to C', but was added on behalf of another component C'', and $w(e') \leq w(e'')$. We continue in this way until we come to an edge $e^{(r)}$ not in $\{e_1, \ldots, e_k\}$.

Finally, let $T' = T \cup \{e_{k+1}\} - \{e^{(r)}\}$. Then by stringing together the inequalities we have obtained along the way, we know that $w(e_{k+1}) \leq w(e^{(r)})$. It follows that $w(T') \leq w(T)$, so T' is again a minimum spanning tree. Furthermore, T' contains $e_1, e_2, \ldots, e_k, e_{k+1}$, and we have our desired contradiction.

29. Suppose that there are r trees in the forest at some intermediate stage of Sollin's algorithm. Each new tree formed during this stage will then contain at least two of the old trees, so there are at most $r/2$ new trees. In other words, we have to reduce the number of trees by at least $r - (r/2) = r/2$. Since each edge added at this stage reduces the number of trees by exactly one, we must add at least $r/2$ edges. Finally, the number of edges added is of course an integer, so it is at least $\lceil r/2 \rceil$.

31. This follows easily from Exercises 29 and 30. After k stages of Sollin's algorithm, since the number of trees begins at n and is at least halved at each stage, there are at most $n/2^k$ trees. Thus if $n \leq 2^k$, then this quantity is less than or equal to 1, so the algorithm has terminated. In other words, the algorithm terminates after at most k stages if $k \geq \log n$, which is what we wanted to prove.

33. Suppose by way of contradiction that a minimum spanning tree T contains edge $e = uv$ that is the maximum weight edge in simple circuit C. Delete e from T. This creates a forest with two components, one containing u and the other containing v. Follow the edges of the path $C - \{e\}$, starting at u. At some point this path must jump from the component of $T - \{e\}$ containing u to the component of $T - \{e\}$ containing v, say using edge f. This edge cannot be in T, because e can be the only edge of T joining the two components (otherwise there would be a simple circuit in T). Because e is the edge of greatest weight in C, the weight of f is smaller. The tree formed by replacing e by f in T therefore has smaller weight, a contradiction.

35. The reverse-delete algorithm must terminate and produce a spanning tree, because the algorithm never disconnects the graph and upon termination there can be no more simple circuits. The edge deleted at each stage of the algorithm must have been the edge of maximum weight in whatever circuits it was a part of. Therefore by Exercise 33 it cannot be in any minimum spanning tree. Since only edges that could not have been in any minimum spanning tree have been deleted, the result must be a minimum spanning tree. Actually the assumption that the edge weights all be distinct can be avoided; see [KlTa05] for details.

GUIDE TO REVIEW QUESTIONS FOR CHAPTER 11

1. **a)** See p. 746. **b)** See p. 746.

2. No, for each ordered pair of vertices u and v, there is a unique simple path from u to v.

3. See Examples 5–8 in Section 11.1.

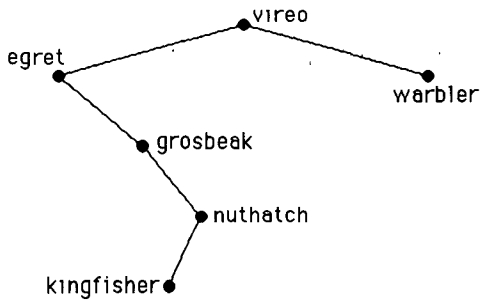
4. **a)** See p. 747. **b)** See p. 747. **c)** See p. 748.
d) Figure 8a in Section 11.1 is such a tree. Its root is a. The parent of each vertex is the vertex immediately above it; thus the parent of b is a, the parent of c is also a, the parent of d is b, and so on. The children of a vertex are the vertices immediately below it; thus the children of a are b and c, the children of b are d and e, the only child of h is j, e has no children, and so on. The internal vertices are the ones with children: a, b, c, d, h, i, and l. The leaves are the vertices without children: e, f, g, j, k, and m.

5. **a)** $n - 1$ **b)** If c is the number of components, then $e = n - c$.

6. **a)** See p. 748. **b)** $mi + 1$; $(m-1)i + 1$; see Theorem 4 in Section 11.1.

7. **a)** See p. 753. **b)** See p. 753. **c)** between 1 and m^h, inclusive

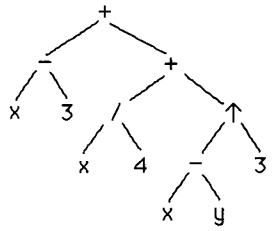
8. **a)** See pp. 757–758. **b)** Repeatedly apply Algorithm 1 in Section 11.2 to insert items one by one.
c) If we insert the items in the order given, we obtain the following tree.



9. **a)** See p. 762. **b)** See pp. 762–763.

10. **a)** See p. 773, 775 and 776. **b)** See Examples 2–4 in Section 11.3.

11. **a)** Build the expression tree. Its preorder traversal gives prefix form; its postorder traversal gives postfix form; and its inorder traversal gives infix form (if we assume that there are only binary operations, and if we put a pair of parentheses around the expression for every subtree that is not a leaf).
b) Here is the expression tree.



c) prefix: $+-x3+/x4\uparrow-xy3$; postfix: $x3-x4/xy-3\uparrow++$

12. See Theorem 1 in Section 11.2 and the discussion preceding it.

13. **a)** See pp. 763–764.
b) The answer is not unique, because there is a choice of whether to make the subtree containing A and B or the leaf C the left subtree at the second step. Assuming we choose the former, the code for A is 000; that for B is 001; that for C is 01; and the code for D is 1.

14. The tree is too large to draw here. (It can be completed by referring to the solution to Exercise 33 in Section 11.2.) The children of the root are the positions 4, 31, 21, 11, and 1. Each of these has value -1, except for the position with just one pile with one stone. That is a winning position for the first player, since it is a leaf and the second player has no move. So the first player wins the game by removing the pile of four stones at her first move. Thus the value of the root of the tree is 1.

15. **a)** See p. 785. **b)** all connected ones **c)** road-plowing, communication network reinforcement

16. **a)** See pp. 787–789 (depth-first search) and pp. 789–791 (breadth-first search).
b) See Exercises 15 and 16 in Section 11.4.

17. **a)** See Example 6 in Section 11.4. **b)** Apply this algorithm to W_5.

18. **a)** See p. 798. **b)** minimum cost communications network, shortest connecting road configuration

19. **a)** See Algorithms 1 and 2 in Section 11.5. **b)** See Examples 2 and 3 in Section 11.5.

SUPPLEMENTARY EXERCISES FOR CHAPTER 11

1. There are of course two things to prove here. First let us assume that G is a tree. We must show that G contains no simple circuits (which is immediate by definition) and that the addition of an edge connecting two nonadjacent vertices produces a graph that has exactly one simple circuit. Clearly the addition of such an edge $e = \{u, v\}$ produces a graph with a simple circuit, namely u, e, v, P, u, where P is the unique simple path joining v to u in G. Since P is unique, moreover, this is the only simple circuit that can be formed.

To prove the converse, suppose that G satisfies the given conditions; we want to prove that G is a tree, in other words, that G is connected (since one of the conditions is already that G has no simple circuits). If G is not connected, then let u and v lie in separate components of G. Then edge $\{u, v\}$ can be added to G without the formation of any simple circuits, in contradiction to the assumed condition. Therefore G is indeed a tree.

3. Let P be a longest simple path in a given tree T. This path has length at least 1 as long as T has at least one edge, and such a longest simple path exists since T is finite. Now the vertices at the ends of P must both have degree 1 (i.e., be pendant vertices), since otherwise the simple path P could be extended to a longer simple path.

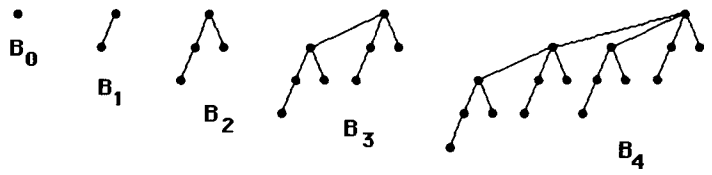
5. Since the sum of the degrees of the vertices is twice the number of edges, and since a tree with n vertices has $n - 1$ edges, the answer is $2n - 2$.

7. One way to prove this is simply to note that the conventional way of drawing rooted trees provides a planar embedding. Another simple proof is to observe that a tree cannot contain a subgraph homeomorphic to K_5 or $K_{3,3}$, since these must contain simple circuits. A third proof, of the fact that a tree can be embedded in the plane with straight lines for the edges, is by induction on the number of vertices. The basis step (a tree with one vertex) is trivial (there are no edges). If tree T contains $n + 1$ vertices, then delete one vertex of degree 1 (which exists by Exercise 3), embed the remainder (by the inductive hypothesis), and then draw the deleted vertex and reattach it with a short straight line to the proper vertex.

9. Since the colors in one component have no effect on the colors in another component, it is enough to prove this for connected graphs, i.e., trees. In order to color a tree with 2 colors, we simply view the tree as rooted and color all the vertices at even-numbered levels with one color and all the vertices at odd-numbered levels with another color. Since every edge connects vertices at adjacent levels, this coloring is proper.

11. A B-tree of degree k and height h has the most leaves when every vertex not at level h has as many children as possible, namely k children. In this case the tree is simply the complete k-ary tree of height h, so it has k^h leaves. Thus the best upper bound for the number of leaves of a B-tree of degree k and height h is k^h. To obtain a lower bound, we want to have as few leaves as possible. This is accomplished when each vertex has as few children as possible. The root must have at least 2 children. Each other vertex not at level h must have at least $\lceil k/2 \rceil$ children. Thus there are $2\lceil k/2 \rceil^{h-1}$ leaves, so this is our best lower bound. (Of course if $h = 0$, then the tree has exactly 1 leaf.)

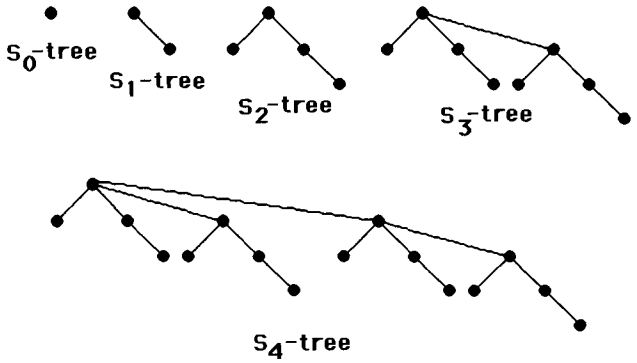
13. We follow the instructions, constructing each tree from the previous trees as indicated.



15. Since B_{k+1} is formed from two copies of B_k, one shifted down one level, the height increases by 1 as k increases by 1. Since B_0 had height 0, it follows by induction that B_k has height k.

17. Since the root of B_{k+1} is the root of B_k with one additional child (namely the root of the other B_k), the degree of the root increases by 1 as k increases by 1. Since B_0 had a root with degree 0, it follows by induction that B_k has a root with degree k.

19. We follow the recursive definition in drawing these trees. For example, we obtain S_4 by taking a copy of S_3 (on the left), adding one more child of the root (the right-most one), and putting a copy of S_3 rooted at this new child.



21. We prove this by induction on k. If $k = 0$ or 1, then the result is trivial. Assume the inductive hypothesis that the S_{k-1}-tree can be formed in the manner indicated, and let T be an S_k-tree. Then T consists of a copy of an S_{k-1} tree T_{k-1} with root r_{k-1}, together with another copy of an S_{k-1} tree whose root is made a child of r_{k-1}. Now by the inductive hypothesis, this latter S_{k-1}-tree can be formed from a handle v and disjoint trees T_0, T_1, ..., T_{k-2} by connecting v to r_0 and r_i to r_{i+1} for $i = 0, 1, \ldots, k-3$. Since our tree T is formed by then joining r_{k-2} to r_{k-1}, our tree is formed precisely in the manner desired, and the proof by induction is complete.

23. Essentially we just want to do a breadth-first search of the tree, starting at the root, and considering the children of a vertex in the order from left to right. Thus the algorithm is to perform breadth-first search, starting at the root of the tree. Whenever we encounter a vertex, we print it out.

25. First we determine the universal addresses of all the vertices in the tree. The root has address 0. For every leaf address, we include an address for each prefix of that address. For example, if there is a leaf with address 4.3.7.3, then we include vertices with addresses 4, 4.3, and 4.3.7. The tree structure is constructed by making all the vertices with positive integers as their addresses the children of the root, in order, and all the vertices with addresses of the form $A.i$, where A is an address and i is a positive integer, the children of the vertex whose address is A, in order by i.

27. For convenience, let us define a "very simple circuit" to be a set of edges that form a circuit all of whose vertices are distinct except that (necessarily) the last vertex is the same as the first. Thus a cactus is a graph in which every edge is in no more than one very simple circuit.

a) This is a cactus. The edge at the top is in no simple circuit. The three edges at the bottom are only in the triangle they form.

b) This is not a cactus, since the edge in the upper right-hand corner, for instance, is in more than one very simple circuit: a triangle and a pentagon.

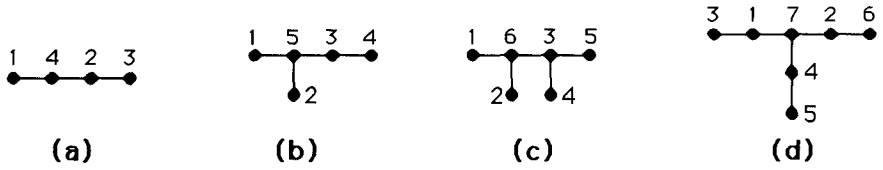
c) This is a cactus. The edges in each of the three triangles are each in exactly one very simple circuit.

29. Adding a very simple circuit (see the solution to Exercise 27 for the definition) does not give the graph any more very simple circuits, except for the one being added. Thus the new edges are each in exactly one very simple circuit, and the old edges are still in no more than one.

31. There is clearly a spanning tree here which is a simple path $(a, b, c, f, e, d$, for instance); since each degree is 1 or 2, this spanning tree meets the condition imposed.

33. There is clearly a spanning tree here which is a simple path $(a, b, c, f, e, d, i, h, g$, for instance); since each degree is 1 or 2, this spanning tree meets the condition imposed.

35. We need to label these trees so that they satisfy the condition. We work by trial and error, using some common sense. For example, the labels 1 and n need to be adjacent in order to obtain the difference $n - 1$.



37. We count the caterpillars by drawing them all, using the length of the longest path to organize our work. In fact every tree with six vertices is a caterpillar. They are the five trees shown with heavy lines in our solution to Exercise 33c in Section 11.1, together with the star $K_{1,5}$. Thus the answer is 6.

39. **a)** The frequencies of the bits strings are 0.81 for 00, 0.09 for 01 and for 10, and 0.01 for 11. The resulting Huffman code uses 0 for 00, 11 for 01, 100 for 10, and 101 for 11. (The exact coding depends on how ties were broken, but all versions are equivalent.) Thus in a string of length n, the average number of bits required to send two bits of the message is $1 \cdot 0.81 + 2 \cdot 0.09 + 3 \cdot 0.09 + 3 \cdot 0.01 = 1.29$, so the average number of bits required to encode the string is $1.29 \cdot (n/2) = 0.645n$.

b) The frequencies of the bits strings are 0.729 for 000, 0.081 for 001, 010, and 100, 0.009 for 011, 101, and 110, and 0.001 for 111. The resulting Huffman code uses 0 for 000, 100 for 001, 101 for 010, 110 for 100, 11100 for 011, 11101 for 101, 11110 for 110, and 11111 for 111. (The exact coding depends on how ties were broken, but all versions are equivalent.) Thus in a string of length n, the average number of bits required to send three bits of the message is $1 \cdot 0.729 + 3 \cdot 3 \cdot 0.081 + 5 \cdot 3 \cdot 0.009 + 5 \cdot 0.001 = 1.598$, so the average number of bits required to encode the string is $1.598 \cdot (n/3) \approx 0.533n$.

41. Let T be a minimum spanning tree. If T contains e, then we are done. If not, then adding e to T creates a simple circuit, and then deleting another edge e' of this simple circuit gives us another spanning tree T'. Since $w(e) \leq w(e')$, the weight of T' is no larger than the weight of T. Therefore T' is a minimum spanning tree containing e.

43. The proof uses the same idea as in the solution to Exercise 18 in Section 11.5. Suppose that edge e is the edge of least weight incident to vertex v, and suppose that T is a spanning tree that does not include e. Add e to T, and delete from the simple circuit formed thereby the other edge of the circuit that contains v. The result will be a spanning tree of strictly smaller weight (since the deleted edge has weight greater than the weight of e). This is a contradiction, so T must include e.

45. Because paths in trees are unique, an arborescence T of a directed graph G is just a subgraph of G that is a tree rooted at r, containing all the vertices of G, with all the edges directed away from the root. Thus the in-degree of each vertex other than r is 1. To show the converse, it is enough to show that for each $v \in V$ there is a unique directed path from r to v. Because the in-degree of each vertex other than r is 1, we can

follow the edges of T backwards from v. This path can never return to a previously visited vertex, because that would create a simple circuit. Therefore the path must eventually stop, and it can stop only at r, the vertex whose in-degree is not necessarily 1 (in fact r has to have in-degree 0). Following this path forward gives the path from r to v required by the definition of arborescence.

47. **a)** We just run the breadth-first search algorithm, starting from v and respecting the directions of the edges, marking each vertex that we encounter as reachable.

b) Clearly running breadth-first search on G^{conv}, again starting at v, respecting the directions of the edges, and marking each vertex that we encounter, will identify all the vertices from which v is reachable.

c) By definition, the strong component of G containing v consists of all vertices w such that w is reachable from v and v is reachable from w. Therefore the set of vertices marked by both of the algorithms above constitute the vertices in the strong component of G containing v. (The edges in this strong component, of course, are just all the edges joining vertices in this strong component.) So choose a vertex v_1 and find the strong component containing this vertex (it might be as small as $\{v_1\}$). Then choose another vertex v_2 not yet in a strong component and find the strong component of v_2. Repeat until all vertices have been included. Recall that by Exercise 17 in Section 10.4, strong components are pairwise disjoint.

WRITING PROJECTS FOR CHAPTER 11

Books and articles indicated by bracketed symbols below are listed near the end of this manual. You should also read the general comments and advice you will find there about researching and writing these essays.

1. You can probably find something useful in [BiLl]. Also, take a look at [WhCl].

2. Look up "phylogenetic tree" on the Web.

3. This Wikipedia site might be a place to start:
`en.wikipedia.org/wiki/Hierarchical_clustering_of_networks`

4. Consult books on data structures or algorithms, such as [Kr] or [Ma2].

5. There is a relevant article in [De2], pp. 290–295.

6. See hints for Writing Project 4.

7. A Web search will turn up some good expositions.

8. An elementary exposition can be found in [Gr2]. See also algorithm or data structures texts, such as [Sm].

9. A Web search will turn up some good sites, including one by Dr. A. N. Walker.

10. See books on parallel processing, such as [Le2], to learn what meshes of trees are and how they are applied.

11. Books are beginning to appear in this new field; see [Ma5], for instance.

12. Good books on algorithms will contain material for this. See [CoLe], for example.

13. Good books on algorithms will contain material for this. See [CoLe], for example.

14. Each search company wants to keep the details of its techniques secret, but you should be able to find some general information on the Web. See also [Sz].

15. This problem is much more difficult than the plain minimum spanning tree problem. You may have to go to the research literature here (search the *Mathematical Reviews*, database available on the Web as MathSciNet, for keywords "spanning tree" and "constrained" or "degree").

16. Any good data structures or algorithms book would have loads of material on this topic, including those mentioned in Writing Project 4 (or others given in the reference list). See also volume 3 of the classic [Kn].

17. See the article [GrHe].

18. See books on random graph theory ([Bo2] or [Pa1]), or the article [Ti].