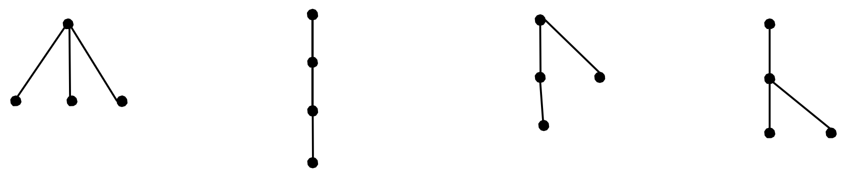
Sec. 11.1 12, 20, 21, 28

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).
12(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.

12(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.



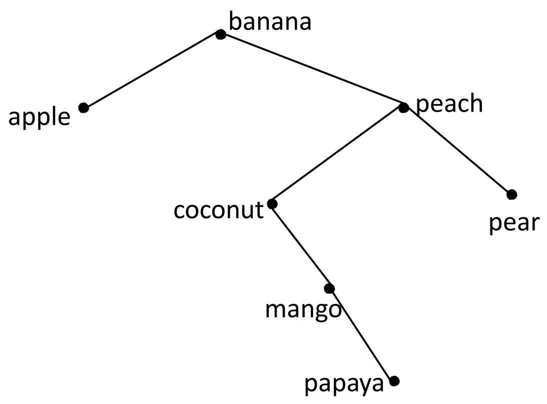
20 By Theorem 4(i), the answer is $[(m - 1)n + 1] / m = (2 \cdot 100 + 1) / 3 = 67$.

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.

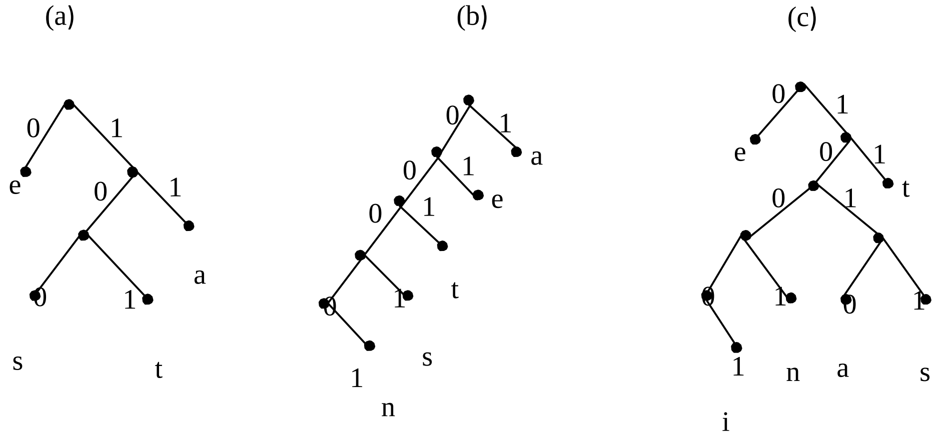
28 The tree has 1 vertex at level 0, m vertices at level 1, m^2 vertices at level 2, …, m^h vertices at level h. Therefore it has $1+m+m^2+\cdots+m^h = \dfrac{m^{h+1}-1}{m-1}$ vertices in all. The vertices at level h are the only leaves, so it has m^h leaves.

Sec.11.2 1, 20, 23

1 We first insert *banana* into the empty tree, giving us the tree with just a root, labeled *banana*. Next we inset *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.

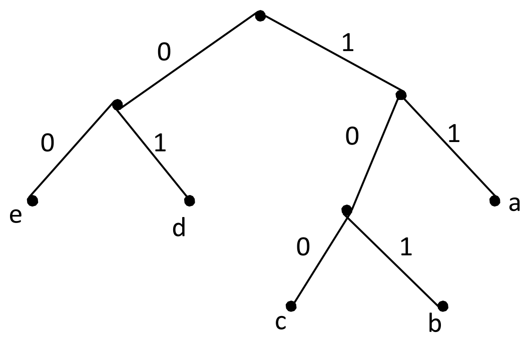


20 The constructions are straightforward.



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 purpose, of weight 0.10+0.15=0.25, with the largest 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

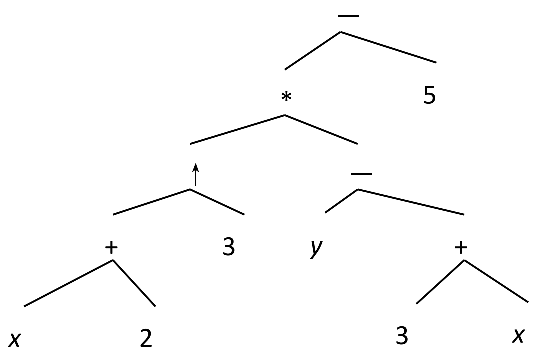
and T_3. The result is shown as below. We see by looking at the tree that *a* is encoded by 11, *b* by 101, *c* by 100, *d* by 01, and *e* by 00. To compute the average number of bits required to encode a character, we multiply the number of bits for each letter by the weight of that letter and add. Since *a* takes 2 bits and has weight 0.20, it contributes 0.40 to the sum. Similarly *b* contributes 3 •0.10 =0.30. In all we get 2 •0.20 + 3 • 0.10 +3 • 0.15 + 2 • 0.25 + 2 • 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.



Sec. 11.3 8,
16

8 In preorder, the root comes first, then list 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.*

16(a) We build the tree from the top down while analyzing the expression by identifying the outermost operation at each stage. The outmost operation in this expression is the final subtraction. Therefore the tree has – 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.



16(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 —∗↑+x23—y

+3x5.

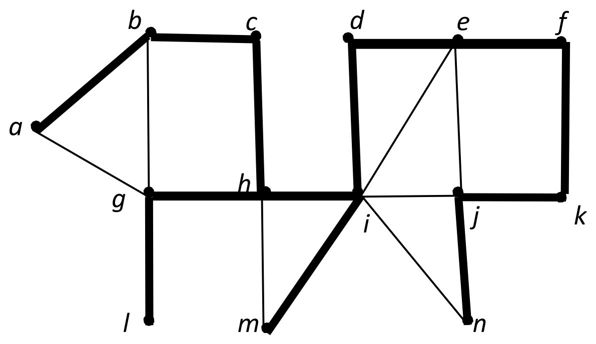
16(c) We can read off the answer from the picture we have just drawn simply by listing the vertices of the tree in postorder: x2+3 ↑ y3x+－∗5－.

16(d) The infix expression is just the given expression, fully parenthesized: ((((*x*+2) ↑ 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.

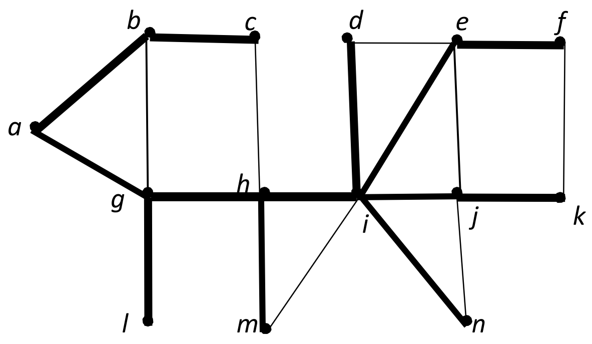
Sec. 11.4
4, 14, 16(14), 31

4 We can remove these edges to produce a spanning tree: {*a*,*i*}, {*b*,*i*}, {*b*,*j*}, {*c*,*d*}, {*c*,*j*}, {*d*,*e*}, {*e*,*j*}, {*f*,*i*}, {*f*,*j*}, and {*g*,*i*}. There are many other possible answers, corresponding to different choices of edges to remove.

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(14) If we start at vertex *a* and **use alphabetical order,** then the breadth-first search spanning tree is **unique**. We first fan out from vertex *a*, picking up the edges {*a*,*b*} and {*a*,*g*}. Then we fan out from *b* to get edges {*b*,*c*} and {*b*,*g*}. Next we fan out from *g* to get edges {*g*,*l*} and {*g*,*h}*. This process continues until we have the entire tree shown in heavy lines below.



31 Assume that the graph has vertices v_1, v_2, …, 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 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.

Sec. 11.5
3,7,12

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.

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.

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.