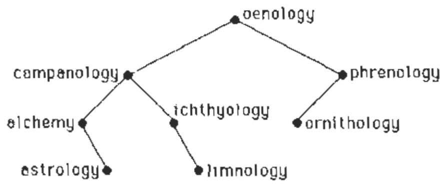
SECTION 10.2 Applications of Trees

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.

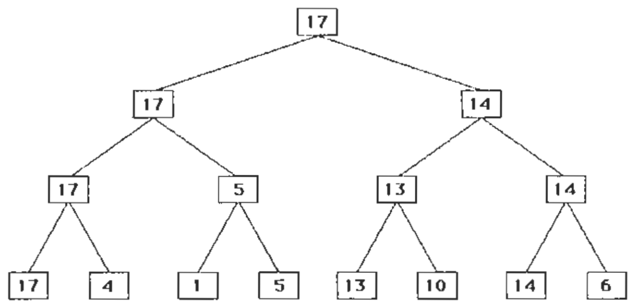
10. There are nine possible outcomes here: either there is no counterfeit, or else we need to name a coin (4 choices) and a type (lighter or heavier). Decision tree theory holds out hope that perhaps only two weighings are needed, but we claim that we cannot get by with only two. Suppose the first weighing 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.

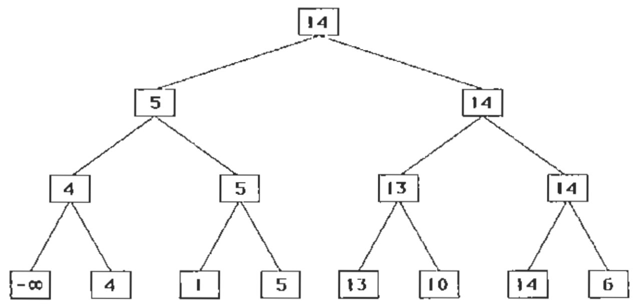
12. By Theorem 1 in this section, at least $\lceil \log 5! \rceil$ comparisons are needed. Since $\log_2 120 \approx 6.9$, at least seven comparisons are required. We can accomplish the sorting with seven comparisons as follows. Call the elements a, b, c, d, and e. First compare a and b; and compare 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.

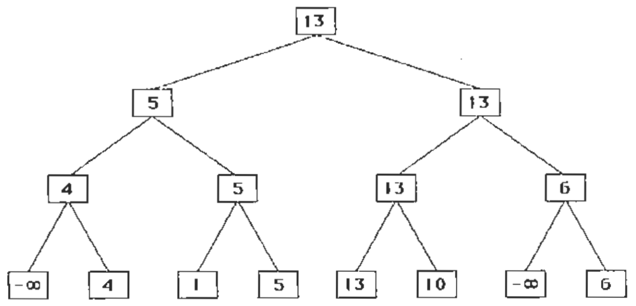
14. The first step builds the following tree.



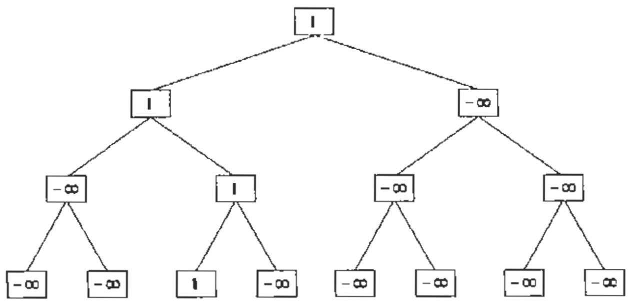
This identifies 17 as the largest element, so we replace the leaf 17 by $-\infty$ in the tree and recalculate the winner in the path from the leaf where 17 used to be up to the root. The result is as shown here.



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.



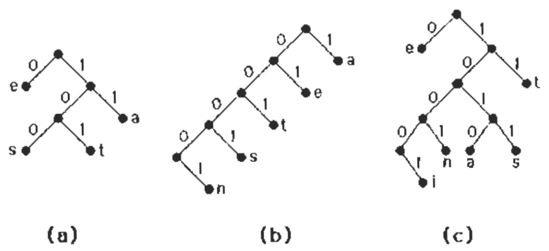
Thus we see that 13 is the third largest element, so we repeat the process: replace the leaf 13 by $-\infty$ and recalculate. The process continues in this manner. The final tree will look like this, as we determine that 1 is the eighth largest element.



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.

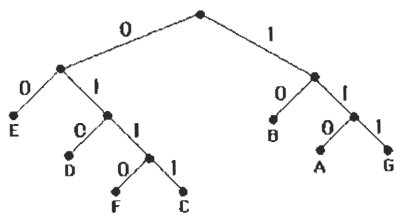
18. Following the hint we insert enough $-\infty$ values to make n a power of 2. This at most doubles n and so will not affect our final answer in big-Theta notation. By Exercise 16 we can build the initial tree using $n - 1$ comparisons. By Exercise 17 for each round after the first it takes $k = \log n$ 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.



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.

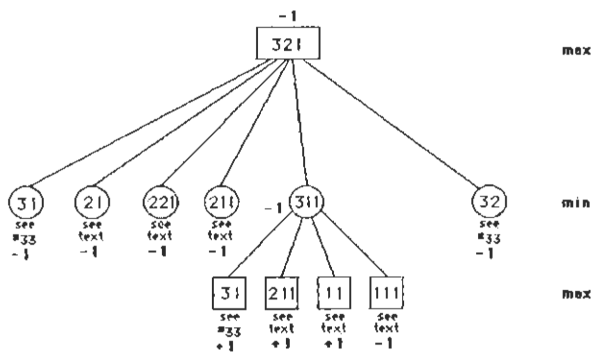
30. **a)** It is easy to construct this tree using the Huffman coding algorithm, as in previous exercises. We get A:0, B:10, C:11.

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.

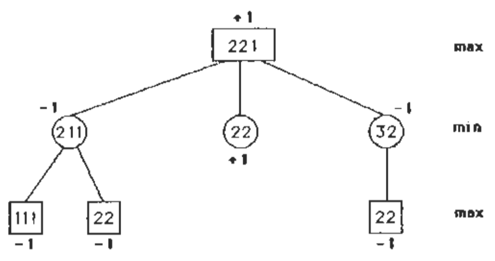
c) The average number of bits for part **(a)** is $1 \cdot 0.80 + 2 \cdot 0.19 + 2 \cdot 0.01 = 1.2000$ per symbol. The average number of bits for part **(b)** is $1 \cdot 0.6400 + 2 \cdot 0.1520 + 5 \cdot 0.0080 + 3 \cdot 0.1520 + 4 \cdot 0.0361 + 7 \cdot 0.0019 + 6 \cdot 0.0080 + 8 \cdot 0.0019 + 8 \cdot 0.0001 = 1.6617$ 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 22.

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 value(P, player) := payoff to first player
else if player = First then
begin {compute maximum of values of children}
          v := −∞
          for each legal move m for First
          begin {compute value of game at resulting position}
                    Q := (P followed by move m)
                    v′ := value(Q, Second)
                    if v′ > v then v := v′
          end
          value(P, player) := v
end
else { player = Second }
begin {compute minimum of values of children}
          v := ∞
          for each legal move m for Second
          begin {compute value of game at resulting position}
                    Q := (P followed by move m)
                    v′ := value(Q, Second)
                    if v′ < v then v := v′
          end
          value(P, player) := v
end
```