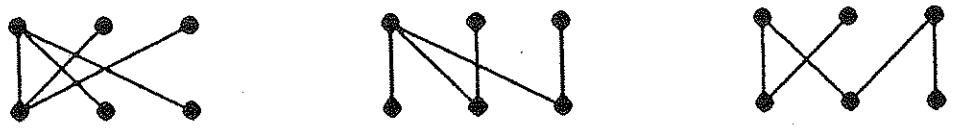
9. We claim that the three spanning trees shown here are the only ones, up to isomorphism.

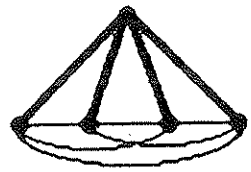


To see this, first note that these three are mutually nonisomorphic, since they all have different numbers of vertices of degree 1. Next suppose that no vertex has degree greater than 2 in the spanning tree. The only tree without a vertex of degree greater than 2 is a path, and we have included this as the right-most picture. Otherwise, suppose that v is a vertex of degree 3 in the spanning tree, say the upper left-hand vertex in the drawings above. Then the other two vertices in the same part as v are either both joined to the same vertex in the other part (the left-most picture), or they are joined to different vertices (the middle picture); there are no other possibilities.

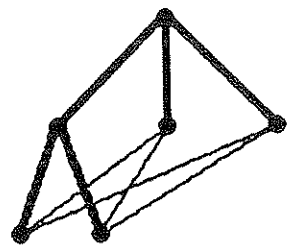
11. Suppose that C is a cut and T is a spanning tree. If the edges of C are removed, then the resulting graph is not connected, hence cannot contain T. In other words, not every edge of T is in the set of edges *not* in C. Therefore at least one edge of T was in C, as desired.

13. In each case we draw the spanning tree in the conventional form, using heavy lines. The thinner lines represent the edges in the graph that are not in the tree.

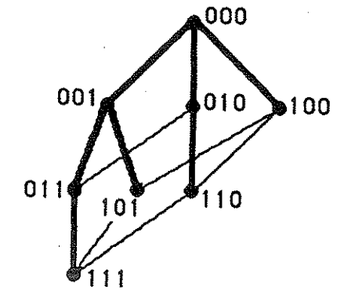
(a) Since every vertex is adjacent to the starting vertex, the spanning tree is just a star.



(b) The root of the tree is one vertex in one of the parts. All the vertices in the other part are adjacent to the starting vertex, so there are three vertices at level 1. The remaining two vertices in the first part are at level 2.



(c) We choose vertex 000 as the root (it represents the smallest number in base 2). It is adjacent to 001, 010, and 100, so these vertices are at level 1. Vertices 011 and 101 are both adjacent to the first of the level 1 vertices, and vertex 110 is adjacent to the second of them. Finally vertex 111 appears at level 3, adjacent to vertex 011. Note that no cross edges join vertices at the same level, since the graph is bipartite.



15. The breadth-first search spanning tree for K_n is a star, $K_{1,n-1}$, since in K_n every vertex is adjacent to the starting vertex. The depth-first search spanning tree for K_n is clearly a path.

17. Suppose that an edge uv of G is not used in the depth-first search spanning tree. We can assume without loss of generality that vertex u is encountered first during the search process. Since edge uv is not used, it must be the case that the search proceeded with other neighbors of u, whose numbers were less than v, and while searching from one of them, say v', encountered v. Thus v is in the subtree rooted at v', hence a descendant of v'. But v' is a descendant of u, so v is a descendant of u.

19. There is a tree on five vertices that is neither a path nor a star ($K_{1,4}$); it consists of a path of length 3, with the fifth vertex joined to one of the middle vertices of the path. There is no way that this tree can be either the depth-first nor the breadth-first search spanning tree of K_5, since we saw in Exercise 15 that these trees are a path and a star, respectively.

21. Since no edge is ever put into T that leads to a previously visited vertex, T can contain no cycles. To see that every vertex v will eventually be visited, however (and an edge to it included in the tree), we proceed by induction on the length of the shortest path from vertex 1 to vertex v. If this length is 0, then $v = 1$ and v is visited immediately. Now suppose that all vertices at a distance less than k from the root (vertex 1) have been visited, and suppose that v is a vertex at a distance k from the root. This means that there is a path of length k from the root to v; let u be the next to last vertex in such a

path. Then clearly u is at a distance $k-1$ from the root, so by the inductive hypothesis, it was visited. By the way the algorithm works, since v is adjacent to u, the algorithm will necessarily have visited v before leaving vertex u. Therefore v will be included in the tree as well.

23. We will find a spanning forest by finding a spanning tree in each component. We need to modify Algorithm 2 so that it does not terminate when the queue L becomes empty, but rather starts over with an unvisited vertex. In more detail, we let F accumulate the forest, one tree at a time. At each iteration, we search for an unvisited vertex, called *next*, and then proceed with the searching steps starting the queue with just *next* in it.

```
procedure breadth_first_search_forest(G : graph)
    {assume setting of Algorithm 2}
    F ← ∅ {the forest is originally empty}
    for i ← 1 to n do
        visited(i) ← false {no vertices have been visited yet}
    next ← 1 {first find the tree for the component containing vertex 1}
    while next > 0 do
        begin {this much is virtually unchanged}
            visited(next) ← true
            T ← ({next}, ∅)
            L ← (next)
            while L is not empty do
                begin
                    i ← first element of L
                    L ← L with i removed
                    for j ← 1 to n do
                        if (j is adjacent to i) ∧ not visited(j) then
                            begin
                                visited(j) ← true
                                add vertex j and edge ij to T
                                add vertex j to the end of L
                            end
                end
            F ← F ∪ T {throw T into the forest}
            next ← 0 {look for another component}
            for i ← n down to 1 do
                if not visited(i) then next ← i
        end
    return(F)
```

25. Vertices at distance 1 from vertex 1 are at level 1, since they get visited from vertex 1; vertices at distance 2 from vertex 1 are adjacent to vertices at distance 1 from vertex 1, so they must appear at level 2 as children of vertices at level 1, and so on.

27. **(a)** Add e to T. The result has a cycle. Let f be any edge of the cycle other than e. Then $T \cup \{e\} - \{f\}$ is again a tree, since it is connected and has the right number of edges.

(b) We need only show how, given a spanning tree T and another spanning tree $T' \neq T$, we can find a spanning tree T'' which is the same as T except that one edge of $T - T'$ has been replaced by an edge $e \in T' - T$ (this is nonempty since $T \neq T'$). Apply part (a), being careful to pick the edge f to be deleted from T not to lie in T' (this is possible, because not every edge in the cycle can lie in T').

29. We use the stack L to keep track of the vertices on which we are currently working. Initially L contains only vertex 1. When we encounter an unvisited vertex we push it onto the stack (as well as adding it and the appropriate edge to the tree). When we have finished with a vertex (i.e., when we exhaust the search, in the inner **while** loop below, for unvisited neighbors) we remove it from the stack and continue processing the vertex that then appears at the top of the stack. The algorithm always works on the top element of the stack. Our algorithm is somewhat inefficient, since every time a vertex comes to the top of the stack we start over searching for unvisited neighbors, but it does work correctly.

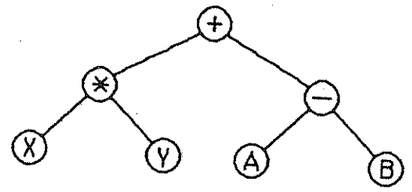
```
procedure iterative_DFS(G : connected graph)
    visited(1) ← true {start at vertex 1}
    for i ← 2 to n do
        visited(i) ← false {no other vertices have been visited}
    T ← ({1}, ∅) {the tree starts with no edges}
    L ← (1) {the stack has only vertex 1}
    while L ≠ ∅ do
        begin
            i ← first element of L {search from vertex i}
            j ← 1 {look for unvisited neighbor}
            found ← false
            while j ≤ n ∧ not found do
                begin
                    if j is adjacent to i ∧ not visited(j) then
                        begin {found an unvisited neighbor}
                            visited(j) ← true
                            found ← true
                            add j to the front of L
                                {processing will continue with j}
                            add j and ij to T
                        end
                    j ← j + 1 {try the next neighbor}
                end
            if not found then remove first element from L
                {we are finished with vertex i}
        end
    return(T)
```

SECTION 9.3 Tree Traversal

1. **(a)** The root a comes first. Then we need to list the vertices in the first (left-most) immediate subtree in preorder. This begins with its root b, then b's only child e, then e's first child h, e's second child i, and the last immediate subtree of the tree rooted at e in preorder, namely j, o. This finishes the subtree rooted at b, so we next list the subtree rooted at c, namely just c itself, and finally the subtree rooted at d, which in preorder is f, k, l, m, n, g. Putting this all together we have the preorder: a, b, e, h, i, j, o, c, d, f, k, l, m, n, g.

   (b) The root a comes last. Before that we need to list all the immediate subtrees in postorder. For the first of these, we will have b last, preceded by e, which is preceded by h, i, o, and j, in that order. Thus the sequence will start h, i, o, j, e, b, and will end with a. The rest of the listing is handled in a similar manner, giving the answer: h, i, o, j, e, b, c, k, l, m, n, f, g, d, a.

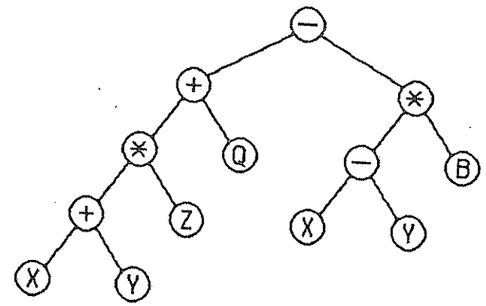
3. **(a)** Fully parenthesized this expression is $((X*Y)+(A-B))$. In particular the outermost operation is $+$, and the two operands are each the result of applying a binary operator to two operands. The expression tree reflects this analysis.



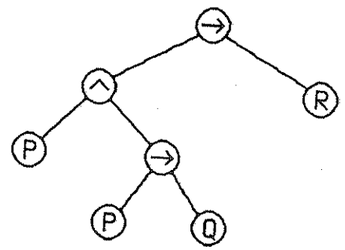
(b) Fully parenthesized by the precedence rules (in particular, the rule that additions and subtractions are performed from left to right), this expression is

$$((((X+Y)*Z)+Q)-((X-Y)*B)).$$

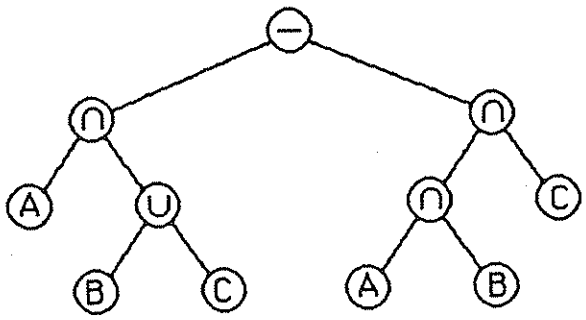
Thus we see that the outermost operator is the minus. Its first operand is the result of the second addition operation, and so on. The expression tree makes it all clear.



(c) The outermost operation here is the second implication. Its right operand is just R, but its first operand is the conjunction of two expressions, the second of which is an implication. The tree shows this structure.



(d) The following tree captures this calculation.



5. In each case we just traverse the tree in postorder and write down what we see.
   (a) $XY*AB-+$
   (b) $XY+Z*Q+XY-B*-$
   (c) $PPQ\rightarrow\wedge R\rightarrow$
   (d) $ABC\cup\cap AB\cap C\cap -$

7. We work from the right, successively replacing the last operator and its two following operands by the result of that operation.
   (a) $*+3-42+63 = *+3-429 = *+329 = *59 = 45$
   (b) $\cup\cap\{1,2,3,4\}\cup\{1\}\{3,5\}\{6,7\} = \cup\cap\{1,2,3,4\}\{1,3,5\}\{6,7\} = \cup\{1,3\}\{6,7\} = \{1,3,6,7\}$
   (c) $\vee F\wedge\wedge\vee F\wedge\rightarrow TFTFT = \vee F\wedge\wedge\vee F\wedge FTFT = \vee F\wedge\wedge\vee FFFT = \vee F\wedge\wedge FFT = \vee F\wedge FT = \vee FF = F$

9. One way to approach these is to draw the expression tree (Exercise 8) and then write down the vertices in postorder. An alternative approach is to move the first operator to the end, find the two operands (i.e., the two complete prefix expressions following the initial operator), and recursively apply this procedure to both of them. The base case is that a constant is left alone.
   (a) $342-+63+*$
   (b) $\{1,2,3,4\}\{1\}\{3,5\}\cup\cap\{6,7\}\cup$
   (c) $FFTF\rightarrow T\wedge\vee F\wedge T\wedge\vee$

11. We work from the left, successively replacing the first operator and its two preceding operands by the result of that operation.
   (a) $33*44*+ = 944*+ = 916+ = 25$
   (b) $\{1,2,3\}\{4,5\}\cap\{5,6\}\cup = \emptyset\{5,6\}\cup = \{5,6\}$
   (c) $FFFF\rightarrow\rightarrow\rightarrow = FFT\rightarrow\rightarrow = FT\rightarrow = T$

13. One way to approach these is to draw the expression tree (Exercise 12) and then write down the vertices in preorder. An alternative approach is to move the last operator to the front, find the two operands (i.e., the two complete postfix expressions preceding the final operator), and recursively apply this procedure to both of them. The base case is that a constant is left alone.

(a) $+*33*44$ **(b)** $\cup\cap\{1,2,3\}\{4,5\}\{5,6\}$ **(c)** $\rightarrow \mathrm{F} \rightarrow \mathrm{F} \rightarrow \mathrm{F}\,\mathrm{F}$

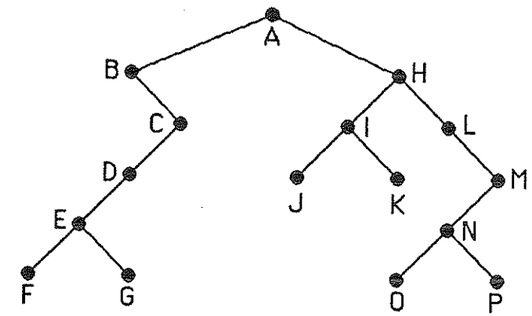
15. We mimic the idea of preorder traversal of an ordered tree, except that to process a vertex means to print it if it is a leaf (and do nothing if it is not a leaf).

```
procedure leaves(r : root of ordered tree)
    if r has no children then print(r)
       else for each child v of r, in order, do
          call leaves(v)
    return
```

17. We need to consider both subtrees, and the most convenient way to do so is by giving a name to their heights. If one or the other of the subtrees is empty, then setting its height to -1 has the right effect.

```
procedure height(r : root of binary tree)
    if r has a left child then left_ ← heightheight(left_child(r))
       else left_ ← height − 1
    if r has a right child then right_ ← heightheight(right_child(r))
       else right_ ← height − 1
    return(1 + max(left_height, right_height))
```

19. Since A comes first in preorder, it must be the root of the tree. We find A in the middle of the inorder listing and know immediately that the vertices listed to its left (B, F, E, G, D, and C) must be in the left subtree, while the remaining vertices must be in the right subtree. Now we know the preorder and inorder listing of both subtrees, so we continue recursively in this manner until we have drawn the entire tree.



21. We can show this with a simple example. The tree with root A and left child B has the same preorder and postorder as the tree with root A and right child B.

23. **(a)** If T is empty, then T^R is empty. If T has root r, then T^R has root r, with left subtree T_1^R, where T_1 is the right subtree of T (if any), and right subtree T_2^R, where T_2 is the left subtree of T (if any).

(b) To construct the tree, we must create a new vertex for each vertex of the given tree. We can work recursively, worrying only about the root. If the root is empty, there is nothing to do. Otherwise, we create a new vertex r_0 for the root, recursively construct the reverse of the right subtree and make it the left subtree of r_0, and recursively construct the reverse of the left subtree and make it the right subtree of r_0. In pseudocode our algorithm appears as follows.

```
procedure reverse(r : root of extended binary tree)
    if r = ∅ then return(∅)
        else
            begin
                create a new vertex r0
                make reverse(right_child(r)) the left child of r0
                make reverse(left_child(r)) the right child of r0
                return(r0)
            end
```

25. A variable name or a constant is a prefix expression. If α_1 and α_2 are prefix expressions and θ is an operator, then $\theta\,\alpha_1\,\alpha_2$ is a prefix expression. A variable name or a constant is a postfix expression. If α_1 and α_2 are postfix expressions and θ is an operator, then $\alpha_1\,\alpha_2\,\theta$ is a postfix expression. (We assumed here that only binary operations were used. Otherwise we need to modify the definition slightly, so that if θ is an n-ary operator, then $\theta\,\alpha_1\,\alpha_2\,\ldots\,\alpha_n$ is the prefix expression created by the recursive part, and $\alpha_1\,\alpha_2\,\ldots\,\alpha_n\,\theta$ is the analogous postfix expression.)

27. We can read these answers from the expression trees, or just form them directly, keeping in mind that an operator immediately precedes its operand or operands.

(a) $*-4\ominus 3\,7$ (where $\ominus$ is unary minus)

(b) $\surd+**A\,A*B\,B$

(c) $\surd+\Uparrow A\Uparrow B$ (where $\Uparrow$ is squaring)

(d) $\cup\cap A\ominus B\,C$ (where $\ominus$ is complementation)

(e) $\rightarrow\vee P\neg P\,Q$

29. The problem is that it is not clear where to put the operator, since there is no notion of "between" with only one operand. For example, for part (c) of Exercise 26, do we write $(\surd((A\Uparrow)+(B\Uparrow)))$ or $(\surd((\Uparrow A)+(\Uparrow B)))$ or what? We are forced to use a hybrid notation—part infix and part either prefix or postfix.

31. This, like most orally unambiguous statements involving calculations, is in prefix notation. The operator of squaring is mentioned before the things being squared ("the square of ..."), and the addition operator is mentioned before the summands ("the sum of ...").

33. Essentially all we want to do is to perform a breadth-first search on the tree. The following procedure is even simpler than the breadth-first search algorithm from Section 9.2, however, since there is no need to keep track of which vertices have been visited (we know that the children of a vertex have never been seen when we are looking at that vertex).

```
procedure level_process(r : root of ordered tree)
    { L is a list of vertices waiting to be processed }
    L ← (r)
    while L is not empty do
        begin
            v ← first element of L
            remove v from L
            process v
            for each child c of v from left to right do
                add c to the end of L
        end
    return
```

35. There are a few base cases to handle—if the root has no children (automatically an AVL-tree), or if there is only a right child (an AVL-tree if the right subtree has height 0), or vice versa. In the general case, three things must be satisfied: The heights of the subtrees must differ by at most 1, and each subtree must be an AVL-tree. The following procedure incorporates these observations in a recursive algorithm.

```
procedure AVL(r : root of binary tree)
    if r has no children then return(true)
        else if r has no left child then
                return(truth value of height(right_child(v) = 0))
            else if r has no right child then
                    return(truth value of height(left_child(v) = 0))
                else return(truth value of
                    |height(left_child(v)) − height(right_child(v))| ≤ 1
                        ∧ AVL(left_child(v)) ∧ AVL(right_child(v)))
```

37. Let x be the first vertex in the first list: It needs to be the root of the tree. Find x in the second list, and let L_1 and L_2 be the lists of vertices that precede and follow x in the second list, respectively. A necessary condition that the tree exists is that the first list consist of the vertices in L_1, in some order, followed by the vertices in L_2, in some order. If the condition is met, then the left and right subtrees of x are determined recursively by L_1 and L_2 and their preorder arrangements given in the first list.