CHAPTER 10

Trees

A *tree* is a nonlinear data structure that models a hierarchical organization. The characteristic features are that each element may have several successors (called its "children") and every element except one (called the "root") has a unique predecessor (called its "parent"). Trees are common in computer science: Computer file systems are trees, the inheritance structure for Java classes is a tree, the run-time system of method invocations during the execution of a Java program is a tree, the classification of Java types is a tree, and the actual syntactical definition of the Java programming language itself forms a tree.

TREE DEFINITIONS

Here is the recursive definition of an (unordered) tree:

> **A *tree* is a pair (*r*, *S*), where *r* is a node and *S* is a set of disjoint trees, none of which contains *r*.**

The node *r* is called the *root* of the tree *T*, and the elements of the set *S* are called its *subtrees*. The set *S*, of course, may be empty. The restriction that none of the subtrees contains the root applies recursively: *r* cannot be in any subtree or in any subtree of any subtree.

Note that this definition specifies that the second component of a tree be a *set* of subtrees. So the order of the subtrees is irrelevant. Also note that a set may be empty, so (*r*, ∅) qualifies as a tree. This is called a *singleton tree*. But the empty set itself does not qualify as an unordered tree.

EXAMPLE 10.1 Equal Unordered Trees

The two trees shown in Figure 10.1 are equal. The tree on the left has root **a** and two subtrees B and C, where B = (**b**, ∅), C = (**c**, {D}), and D is the subtree D = (**d**, ∅). The tree on the right has the same root **a** and the same set of subtrees {B, C} = {C, B}, so (**a**, {B, C}) = (**a**, {C, B}).

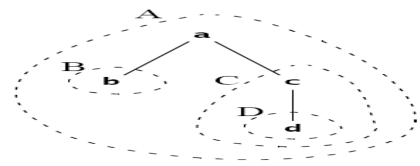

Figure 10.1 Equal trees

The elements of a tree are called its *nodes*. Technically, each node is an element of only one subtree, namely the tree of which it is the root. But indirectly, trees consist of nested subtrees, and each node is considered to be an element of every tree in which it is nested. So **a**, **b**, **c**, and **d** are all considered to be nodes of the tree A shown Figure 10.2. Similarly, **c** and **d** are both nodes of the tree C.

The *size* of a tree is the number of nodes it contains. So the tree A shown in Figure 10.2 has size 4, and C has size 2. A tree of size 1 is called a *singleton*. The trees B and D shown here are singletons.

If $T = (x, S)$ is a tree, then x is the root of T and S is its set of subtrees $S = \{T_1, T_2, \ldots, T_n\}$. Each subtree T_j is itself a tree with its own root r_j. In this case, we call the node r the *parent* of each node r_j, and we call the r_j the *children* of r. In general, we say that two nodes are *adjacent* if one is the parent of the other.

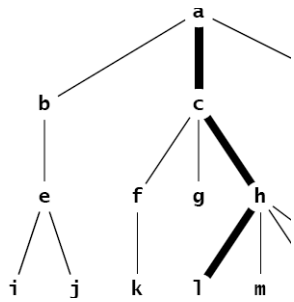

Figure 10.2 Subtrees

A node with no children is called a *leaf*. A node with at least one child is called an *internal node*.

A *path* in a tree is a sequence of nodes $(x_0, x_1, x_2, \ldots, x_m)$ wherein the nodes of each pair with adjacent subscripts (x_{i-1}, x_i) are adjacent nodes. For example, (**a**, **b**, **c**, **d**) is a path in the tree shown above, but (**a**, **d**, **b**, **c**) is not. The *length* of a path is the number m of its adjacent pairs.

It follows from the definition that trees are *acyclic*, that is, no path can contain the same node more than once.

A *root path* for a node x_0 in a tree is a path $(x_0, x_1, x_2, \ldots, x_m)$ where x_m is the root of the tree. A root path for a leaf node is called a *leaf-to-root path*.

Theorem 10.1 Every node in a tree has a unique root path.
For a proof, see Problem 10.1 on page 194.

The *depth* of a node in a tree is the length of its root path. Of course, the depth of the root in any tree is 0. We also refer to the *depth* of a subtree in a tree, meaning the depth of its root.

A *level* in a tree is the set of all nodes at a given depth.

The *height* of a tree is the greatest depth among all of its nodes. By definition, the height of a singleton is 0, and the height of the empty tree is –1. For example, the tree A, shown in Figure 10.2, has height 2. Its subtree C has height 1, and its two subtrees B and D each have height 0.

A node y is said to be an *ancestor* of another node x if it is on x's root path. Note that the root of a tree is an ancestor of every other node in the tree.

A node x is said to be a *descendant* of another node y if y is an ancestor of x. For each node y in a tree, the set consisting of y and all its descendants form the *subtree* rooted at y. If S is a subtree of T, then we say that T is a *supertree* of S.

The *path length* of a tree is the sum of the lengths of all paths from its root. This is the same as the weighted sum, adding each level times the number of nodes on that level. The path length of the tree shown here is $1 \cdot 3 + 2 \cdot 4 + 3 \cdot 8 = 35$.

EXAMPLE 10.2 Properties of a Tree

The root of the tree shown in Figure 10.3 is node **a**. The six nodes **a**, **b**, **c**, **e**, **f**, and **h** are all internal nodes. The other nine nodes are leaves. The path (**l**, **h**, **c**, **a**) is a leaf-to-root path. Its length is 3. Node **b** has depth 1, and node **m** has depth 3. Level 2 consists of nodes **e**, **f**, **g**, and **h**. The height of the tree is 3. Nodes **a**, **c**, and **h** are all ancestors of node **l**. Node **k** is a descendant of node **c** but not of node **b**. The subtree rooted at **b** consists of nodes **b**, **e**, **i**, and **j**.

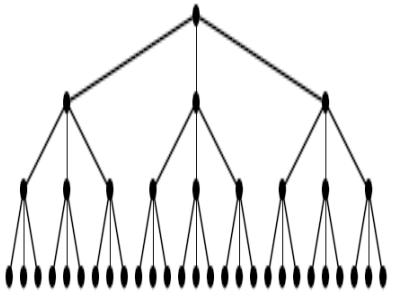

Figure 10.3 A leaf-to-root path

The *degree* of a node is the number of its children. In Example 10.2, **b** has degree 1, **d** has degree 0, and **h** has degree 5.

The *order* of a tree is the maximum degree among all of its nodes.

A tree is said to be *full* if all of its internal nodes have the same degree and all of its leaves are at the same level. The tree shown in Figure 10.4 is a full tree of degree 3. Note that it has a total of 40 nodes.

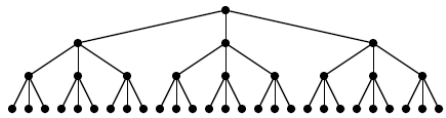

Figure 10.4 A full tree

Theorem 10.2 The full tree of order *d* and height *h* has $\frac{d^{h+1}-1}{d-1}$ nodes.
For a proof, see Problem 10.1 on page 194.

Corollary 10.1 The height of a full tree of order *d* and size *n* is $h = \log_d(nd - n + 1) - 1$.

Corollary 10.2 The number of nodes in any tree of height *h* is at most $\frac{d^{h+1}-1}{d-1}$ where *d* is the maximum degree among its nodes.

DECISION TREES

A *decision tree* is a tree diagram that summarizes all the different possible stages of a process that solves a problem by means of a sequence of decisions. Each internal node is labeled with a question, each arc is labeled with an answer to its question, and each leaf node is labeled with the solution to the problem.

EXAMPLE 10.3 Finding the Counterfeit Coin

Five coins that appear identical are to be tested to determine which one of them is counterfeit. The only feature that distinguishes the counterfeit coin is that it weighs less than the legitimate coins. The only available test is to weigh one subset of the coins against another. How should the subsets be chosen to find the counterfeit?

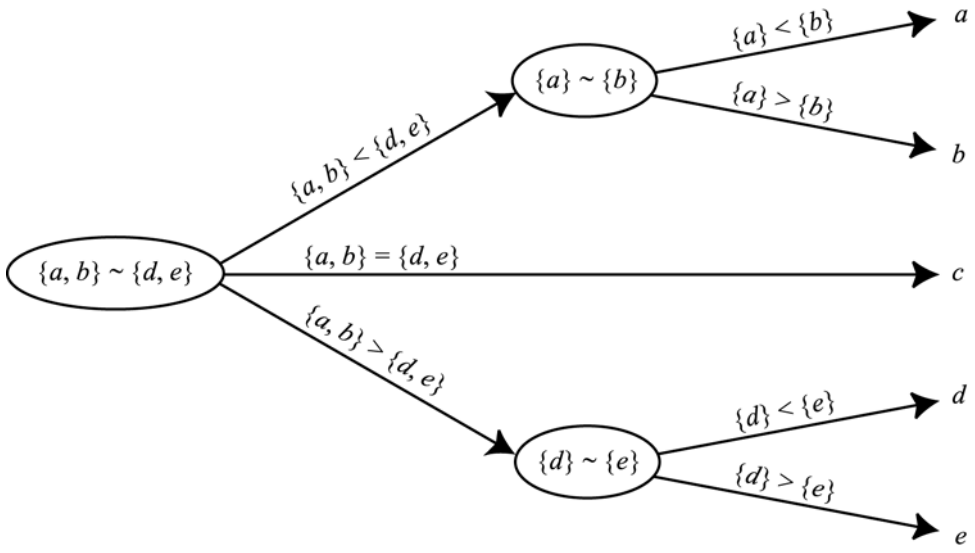

Figure 10.5 A decision tree

In the decision tree shown in Figure 10.5, the symbol $\sim$ means to compare the weights of the two operands. So, for example, $\{a, b\} \sim \{d, e\}$ means to weight coins a and b against coins d and e.

TRANSITION DIAGRAMS

A *transition diagram* is a tree or graph (see Chapter 15) whose internal nodes represent different states or situations that may occur during a multistage process. As in a decision tree, each leaf represents a different outcome from the process. Each branch is labeled with the conditional probability that the resulting child event will occur, given that the parent event has occurred.

EXAMPLE 10.4 The Game of Craps

The game of *craps* is a dice game played by two players, *X* and *Y*. First *X* tosses the pair of dice. If the sum of the dice is 7 or 11, *X* wins the game. If the sum is 2, 3, or 12, *Y* wins. Otherwise, the sum is designated as the "point," to be matched by another toss. So if neither player has won on the first toss, then the dice are tossed repeatedly until either the point comes up or a 7 comes up. If a 7 comes up first, *Y* wins. Otherwise, *X* wins when the point comes up.

The transition diagram shown in Figure 10.6 models the game of craps.

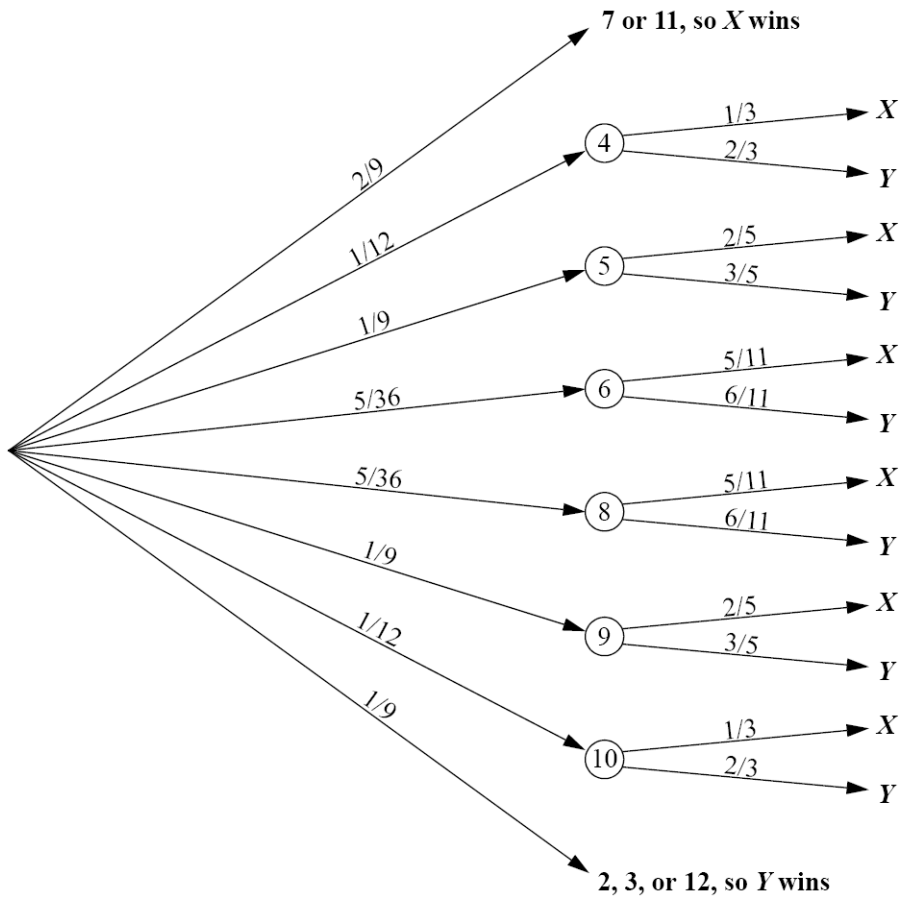

Figure 10.6 A decision tree for the game of craps

When a pair of dice is tossed, there are 36 possible outcomes (6 outcomes on the first die, and 6 outcomes on the second for each outcome on the first die). Of those 36 outcomes, 1 will produce a sum of 2 (1 + 1), 2 will produce a sum of 3 (1 + 2 or 2 + 1), and 1 will produce a sum of 12 (6 + 6). So there are a total of 4 chances out of 36 of the event "2, 3, or 12" happening. That's a probability of 4/36 = 1/9. Similarly, there are 6 ways that a sum of 7 will occur and 2 ways that a sum of 11 will occur, so the probability of the event "7 or 11" is 8/36 = 2/9. The other probabilities on the first level of the tree are computed similarly.

To see how the probabilities are computed for the second level of the tree, consider the case where the point is 4. If the next toss comes up 4, X wins. If it comes up 7, Y wins. Otherwise, that step is repeated. The transition diagram shown in Figure 10.7 summarizes those three possibilities. The probabilities 1/12, 1/6, and 3/4 are computed as shown in the transition diagram in Figure 10.7:

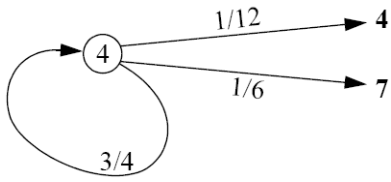

Figure 10.7 The game of craps

$P(4) = 3/36 = 1/12$

$P(7) = 6/36 = 1/3$

$P(2,3,5,6,8,9,10,11, \text{ or } 12) = 27/36 = 3/4$

So once the point 4 has been established on the first toss, X has a probability of 1/12 of winning on the second toss and a probability of 3/4 of getting to the third toss. So once the point 4 has been established on the first toss, X has a probability of (3/4)(1/12) of winning on the third toss and a probability of (3/4)(3/4) of getting to the fourth toss. Similarly, once the point 4 has been established on the first toss, X has a probability of (3/4)(1/12) + (3/4)(3/4)(1/12) of winning on the fourth toss, and so on. Summing these partial probabilities, we find that once the point 4 has been established on the first toss, the probability that X wins on *any* toss thereafter is

$$P_4 = \frac{1}{12} + \left(\frac{3}{4}\right)\frac{1}{12} + \left(\frac{3}{4}\right)^2\frac{1}{12} + \left(\frac{3}{4}\right)^3\frac{1}{12} + \left(\frac{3}{4}\right)^4\frac{1}{12} + \left(\frac{3}{4}\right)^5\frac{1}{12} + \cdots$$

$$= \frac{\frac{1}{12}}{1 - \frac{3}{4}}$$

$$= \frac{1/12}{1/4}$$

$$= \frac{1}{3}$$

This calculation applies the formula for geometric series. (See page 323.)

If the probability is 1/3 that X wins once the point 4 has been established on the first toss, the probability that Y wins at that point must be 2/3. The other probabilities at the second level are computed similarly.

Now we can calculate the probability that X wins the game from the main transition diagram:

$$P = \frac{2}{9} + \frac{1}{12}(P_4) + \frac{1}{9}(P_5) + \frac{5}{36}(P_6) + \frac{5}{36}(P_8) + \frac{1}{9}(P_9) + \frac{1}{12}(P_{10})$$

$$= \frac{2}{9} + \frac{1}{12}\left(\frac{1}{3}\right) + \frac{1}{9}\left(\frac{2}{5}\right) + \frac{5}{36}\left(\frac{5}{11}\right) + \frac{5}{36}\left(\frac{5}{11}\right) + \frac{1}{9}\left(\frac{2}{5}\right) + \frac{1}{12}\left(\frac{1}{3}\right)$$

$$= \frac{244}{495}$$

$$= 0.4929$$

So the probability that X wins is 49.29 percent, and the probability that Y wins is 50.71 percent.

A *stochastic process* is a process that can be analyzed by a transition diagram, that is, it can be decomposed into sequences of events whose conditional probabilities can be computed. The game of craps is actually an infinite stochastic process since there is no limit to the number of events that could occur. As with the analysis in Example 10.4, most infinite stochastic processes can be reformulated into an equivalent finite stochastic process that is amenable to (finite) computers.

Note that, unlike other tree models, decision trees and transition trees are usually drawn from left to right to suggest the time-dependent movement from one node to the next.

ORDERED TREES

Here is the recursive definition of an ordered tree:

> **An *ordered tree* is either the empty set or a pair $T = (r, S)$, where r is a node and S is a sequence of disjoint ordered trees, none of which contains r.**

The node r is called the *root* of the tree T, and the elements of the sequence S are its *subtrees*. The sequence S of course may be empty, in which case T is a singleton. The restriction that none of the subtrees contains the root applies recursively: x cannot be in any subtree, or in any subtree of any subtree, and so on.

Note that this definition is the same as that for unordered trees except for the facts that the subtrees are in a sequence instead of a set and an ordered tree may be empty. Consequently, if two unordered trees have the same subsets, then they are equal; but as ordered trees, they won't be equal unless their equal subtrees are in the same order. Also subtrees of an ordered set may be empty.

EXAMPLE 10.5 Unequal Ordered Trees

The two trees shown in Figure 10.8 are not equal as ordered trees.

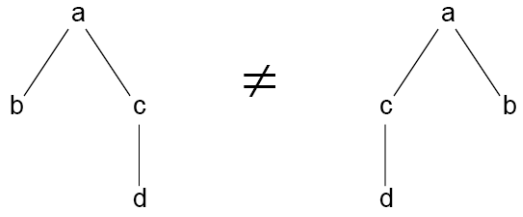

Figure 10.8 Unequal ordered trees

The ordered tree on the left has root node **a** and subtree sequence ((**b**, ∅), (**c**, (**d**, ∅))). The ordered tree on the right has root node **a** and subtree sequence ((**c**, (**d**, ∅)), (**b**, ∅)). These two subtree sequences have the same elements, but not in the same order. Thus the two ordered trees are not the same.

Strict adherence to the definition reveals a subtlety often missed, as illustrated by the next example.

EXAMPLE 10.6 Unequal Ordered Trees

The two trees T_1 = (**a**, (B, C)) and T_2 = (**a**, (B, ∅, C)) are not the same ordered trees, even though they would probably both be drawn the same, as shown in Figure 10.9.

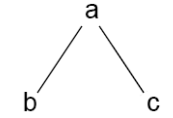

Figure 10.9 A tree

All the terminology for unordered trees applies the same way to ordered trees. In addition, we can also refer to the *first child* and the *last child* of a node in an ordered tree. It is sometimes useful to think analogously of a human genealogical tree, where the children are ordered by age: oldest first and youngest last.

TRAVERSAL ALGORITHMS

A *traversal algorithm* is a method for processing a data structure that applies a given operation to each element of the structure. For example, if the operation is to print the contents of the element, then the traversal would print every element in the structure. The process of applying the operation to an element is called *visiting* the element. So executing the traversal algorithm causes each element in the structure to be visited. The order in which the elements are visited depends upon which traversal algorithm is used. There are three common algorithms for traversing a general tree.

The *level order traversal* algorithm visits the root, then visits each element on the first level, then visits each element on the second level, and so forth, each time visiting all the elements on one level before going down to the next level. If the tree is drawn in the usual manner with its root at the top and leaves near the bottom, then the level order pattern is the same left-to-right top-to-bottom pattern that you follow to read English text.

EXAMPLE 10.7 The Level Order Traversal

The level order traversal of the tree shown in Figure 10.10 would visit the nodes in the following order: **a**, **b**, **c**, **d**, **e**, **f**, **g**, **h**, **i**, **j**, **k**, **l**, **m**.

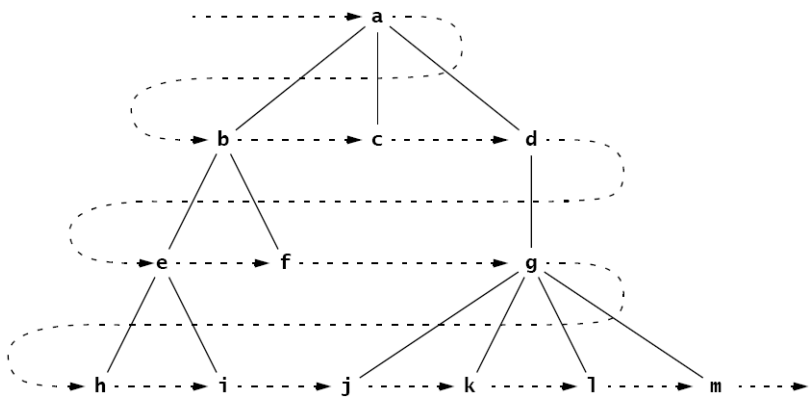

Figure 10.10 A level order traversal

Algorithm 10.1 The Level Order Traversal of an Ordered Tree

To traverse a nonempty ordered tree:

1. Initialize a queue.
2. Enqueue the root.
3. Repeat steps 4–7 until the queue is empty.
4. Dequeue node x from the queue.
5. Visit x.
6. Enqueue all the children of x in order.

The *preorder traversal* algorithm visits the root first and then does a preorder traversal recursively to each subtree in order.

EXAMPLE 10.8 The Preorder Traversal

The preorder traversal of the tree shown in Figure 10.11 would visit the nodes in this order: **a**, **b**, **e**, **h**, **i**, **f**, **c**, **d**, **g**, **j**, **k**, **l**, **m**.

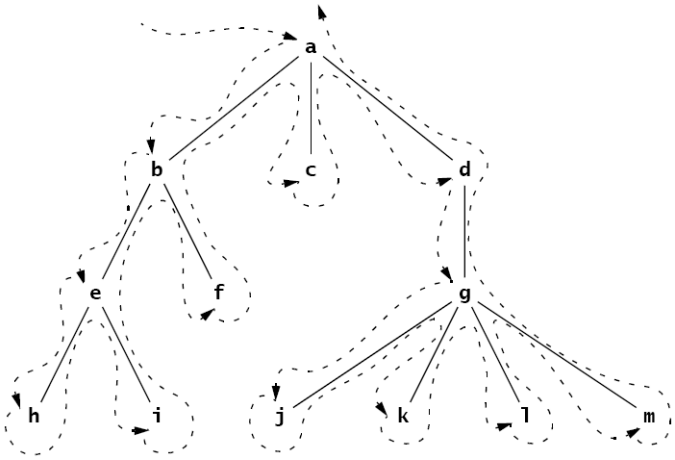

Figure 10.11 A preorder traversal

Note that the preorder traversal of a tree can be obtained by circumnavigating the tree, beginning at the root and visiting each node the first time it is encountered on the left.

Algorithm 10.2 The Preorder Traversal of an Ordered Tree
To traverse a nonempty ordered tree:

1. Visit the root.
2. Do a recursive preorder traversal of each subtree in order.

The *postorder traversal* algorithm does a postorder traversal recursively to each subtree before visiting the root.

EXAMPLE 10.9 The Postorder Traversal

The postorder traversal of the tree shown in Figure 10.12 would visit the nodes in the following order: **h**, **i**, **e**, **f**, **b**, **c**, **j**, **k**, **l**, **m**, **g**, **d**, **a**.

Algorithm 10.3 The Postorder Traversal of an Ordered Tree
To traverse a nonempty ordered tree:

1. Do a recursive preorder traversal of each subtree in order.
2. Visit the root.

Note that the level order and the preorder traversals always visit the root of each subtree first before visiting its other nodes. The postorder traversal always visits the root of each subtree last after visiting all of

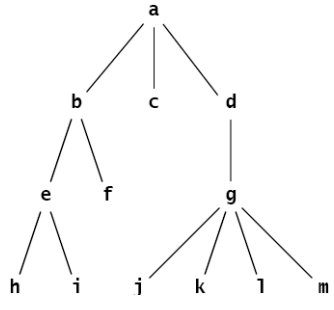

Figure 10.12 A tree

its other nodes. Also, the preorder traversal always visits the right-most node last, while the postorder traversal always visits the left-most node first.

The preorder and postorder traversals are recursive. They also can be implemented iteratively using a stack. The level order traversal is implemented iteratively using a queue.

Review Questions

10.1 All the classes in Java form a single tree, called the *Java inheritance tree.*

- **a.** What is the size of the Java inheritance tree in Java 1.3?
- **b.** What is the root of the tree?
- **c.** What kind of node is a `final` class in the Java inheritance tree?

10.2 True or false:

- **a.** The depth of a node in a tree is equal to the number of its ancestors.
- **b.** The size of a subtree is equal to the number of descendants of the root of the subtree.
- **c.** If x is a descendant of y, then the depth of x is greater than the depth of y.
- **d.** If the depth of x is greater than the depth of y, then x is a descendant of y.
- **e.** A tree is a singleton if and only if its root is a leaf.
- **f.** Every leaf of a subtree is also a leaf of its supertree.
- **g.** The root of a subtree is also the root of its supertree.
- **h.** The number of ancestors of a node equals its depth.
- **i.** If R is a subtree of S and S is a subtree of T, then R is a subtree of T.
- **j.** A node is a leaf if and only if it has degree 0.
- **k.** In any tree, the number of internal nodes must be less than the number of leaf nodes.
- **l.** A tree is full if and only if all of its leaves are at the same level.
- **m.** Every subtree of a full binary tree is full.
- **n.** Every subtree of a complete binary tree is complete.

10.3 For the tree shown in Figure 10.13, find:

- **a.** all ancestors of node F
- **b.** all descendants of node F
- **c.** all nodes in the subtree rooted at F
- **d.** all leaf nodes

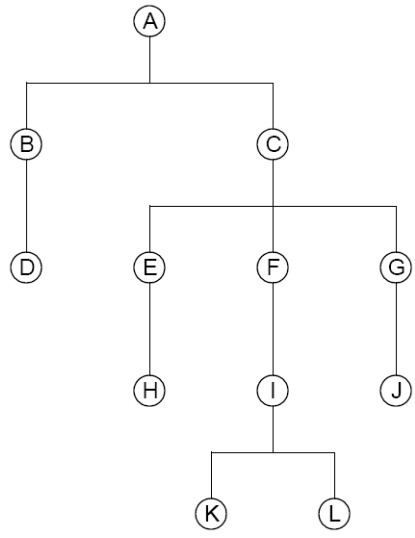

Figure 10.13 A tree

10.4 For each of the five trees shown in Figure 10.14 on page 195, list the leaf nodes, the children of node C, the depth of node F, all the nodes at level 3, the height, and the order of the tree.

10.5 How many nodes are in the full tree of:

- **a.** order 3 and height 4?
- **b.** order 4 and height 3?
- **c.** order 10 and height 4?
- **d.** order 4 and height 10?

10.6 Give the order of visitation of the tree shown in Example 10.2 on page 187 using the:

- **a.** level order traversal
- **b.** preorder traversal
- **c.** postorder traversal

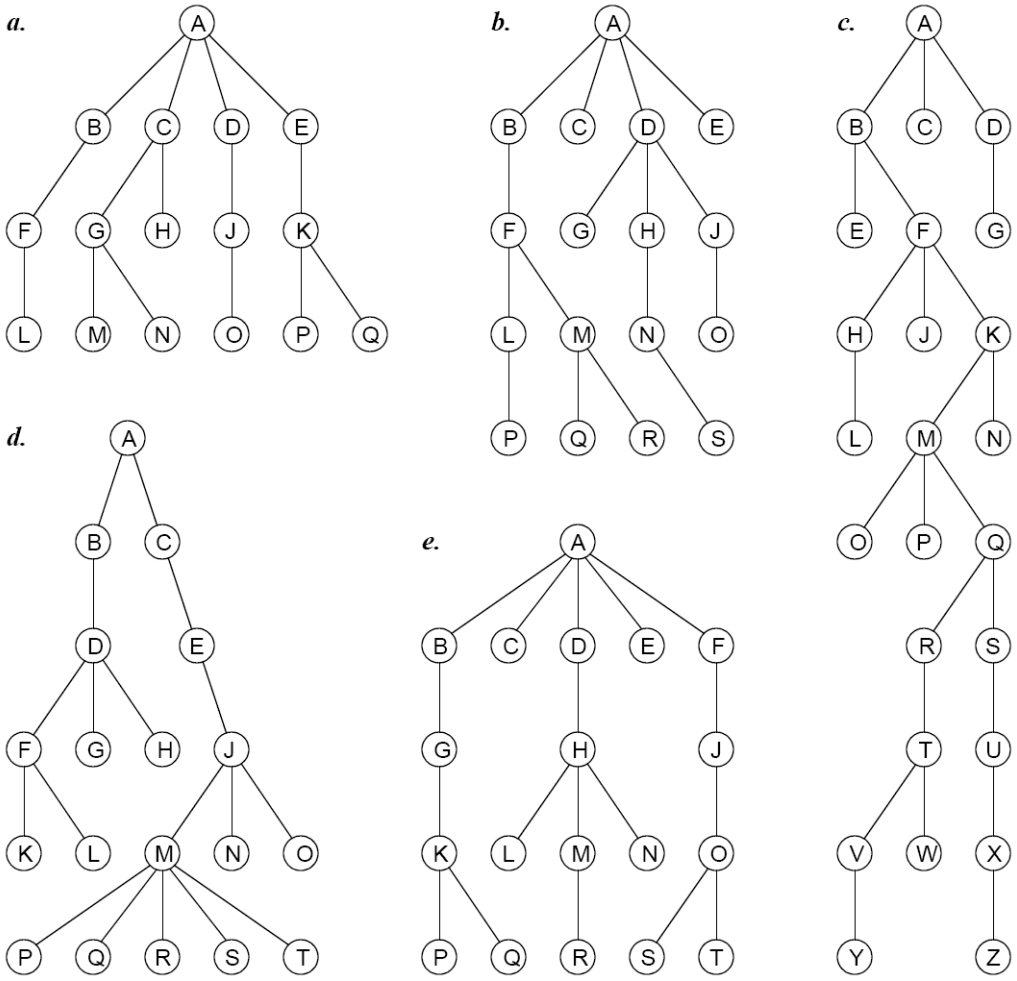

Figure 10.14 A tree

10.7 Which traversals always visit:
- **a.** the root first?
- **b.** the left-most node first?
- **c.** the root last?
- **d.** the right-most node last?

10.8 The level order traversal follows the pattern as reading a page of English text: left-to-right, row-by-row. Which traversal algorithm follows the pattern of reading vertical columns from left to right?

10.9 Which traversal algorithm is used in the call tree for the solution to Problem 9.32 on page 184?

Problems

10.1 Prove Theorem 10.1 on page 187.

10.2 Prove Theorem 10.2 on page 188.

10.3 Prove Corollary 10.1 on page 188.

10.4 Prove Corollary 10.2 on page 188.

10.5 Derive the formula for the path length of a full tree of order d and height h.

10.6 The *St. Petersburg Paradox* is a betting strategy that seems to guarantee a win. It can be applied to any binomial game in which a win or lose are equally likely on each trial and in which the amount bet on each trial may vary. For example, in a coin-flipping game, bettors may bet any number of dollars on each flip, and they will win what they bet if a head comes up, and they will lose what they bet if a tail comes up. The St. Petersburg strategy is to continue playing until a head comes up, and to double your bet each time it doesn't. For example, the sequence of tosses is {T, T, T, H}, then the bettor will have bet \$1 and lost, then \$2 and lost, then \$4 and lost, then \$8 and won, ending up with a net win of –\$1 + –\$2 + –\$4 + \$8 = \$1. Since a head has to come up eventually, the bettor is guaranteed to win \$1, no matter how many coin flips it takes. Draw the transition diagram for this strategy showing the bettor's winnings at each stage of play. Then explain the flaw in this strategy.

10.7 Some people play the game of craps allowing 3 to be a possible point. In this version, player Y wins on the first toss only if it comes up 2 or 12. Use a transition diagram to analyze this version of the game and compute the probability that X wins.

10.8 Seven coins that appear identical are to be tested to determine which one of them is counterfeit. The only feature that distinguishes the counterfeit coin is that it weighs less than the legitimate coins. The only available test is to weigh one subset of the coins against another. How should the subsets be chosen to find the counterfeit? (See Example 10.3 on page 188.)

Answers to Review Questions

10.1 In the *Java inheritance tree:*

- **a.** The size of the tree in Java 1.3 is 1730.
- **b.** The `Object` class is at the root of the tree.
- **c.** A `final` class is a leaf node in the Java inheritance tree.

10.2

- **a.** True.
- **b.** False: It's one more because the root of the subtree is in the subtree but is not a descendant of itself.
- **c.** True
- **d.** False
- **e.** True
- **f.** True
- **g.** False
- **h.** True
- **i.** True
- **j.** True
- **k.** False
- **l.** False
- **m.** True
- **n.** True

10.3

- **a.** The leaf nodes are L, M, N, H, O, P, Q; the children of node C are G and H; node F has depth 2; the nodes at 3 three are L, M, N, O, P, and Q; the height of the tree is 3; the order of the tree is 4.
- **b.** The leaf nodes are C, E, G, O, P, Q, R, and S; node C has no children; node F has depth 2; the nodes at level 3 are L, M, N, and O; the height of the tree is 4; the order of the tree is 4.

c. The leaf nodes are C, E, G, J, L, N, O, P, W, Y, and Z; node C has no children; node F has depth 2; the nodes at level 3 are H, J, and K; the height of the tree is 9; the order of the tree is 3.

d. The leaf nodes are G, H, K, L, N, O, P, Q, R, S, and T; the only child node C has is node E; node F has depth 3; the nodes at level 3 are F, G, H, and J; the height of the tree is 5; the order is 5.

e. The leaf nodes are D, E, L, N, P, Q, R, S, and T; node C has no children; node F has depth 1; the nodes at level 3 are K, L, M, N, and O; the height of the tree is 4; the order of the tree is 5.

10.4 **a.** The ancestors of F are C and A

b. The descendants of F are I, K, and L.

c. The nodes in the subtree rooted at F are F, I, K, and L.

d. The leaf nodes are D, H, J, K, and L.

10.5 **a.** $(3^5 - 1)/2 = 121$ nodes

b. $(4^4 - 1)/3 = 85$ nodes

c. $(10^5 - 1)/9 = 11{,}111$ nodes

d. $(4^{11} - 1)/3 = 1{,}398{,}101$ nodes

10.6 **a.** Level order: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o.

b. Preorder: a, b, e, i, j, c, f, k, g, h, l, m, n, o, d.

c. Postorder: i, j, e, b, k, f, g, l, m, n, o, h, c, d, a.

10.7 **a.** The level order and the preorder traversals always visit the root first.

b. The postorder traversal always visits the left-most node first.

c. The postorder traversal always visits the root last.

d. The preorder traversal always visits the right-most node last.

10.8 The preorder traversal follows the pattern of reading by column from left to right.

10.9 The preorder traversal is used in Problem 9.32 on page 184.

Solutions to Problems

10.1 Proof of Theorem 10.1 on page 187:

If there were no path from a given node x to the root of the tree, then the definition of tree would be violated, because to be an element of the tree, x must be the root of some subtree. If there were more than one path from x back to the root, then x would be an element of more than one distinct subtree. That also violates the definition of tree, which requires subtrees to be disjoint.

10.2 Proof of Theorem 10.2 on page 188:

If the tree is empty, then its height is $h = -1$ and the number of nodes $n = 0$. In that case, the formula is correct: $n = (d^{(h)+1}-1)/(d-1) = (d^{(-1)+1}-1)/(d-1) = (d^0-1)/(d-1) = (1-1)/(d-1) = 0$.

If the tree is a singleton, then its height is $h = 0$ and the number of nodes $n = 1$. In that case, the formula is again correct: $n = (d^{(h)+1}-1)/(d-1) = (d^{(0)+1}-1)/(d-1) = (d-1)/(d-1) = 1$.

Now assume that the formula is correct for any full tree of height $h-1$, where $h \geq 0$. Let T be the full tree of height h. Then by definition, T consists of a root node and a set of d subtrees. And since T is full, each of its d subtrees has height $h-1$. Therefore, by the inductive hypothesis, the number of nodes in each subtree is $n_S = (d^{(h-1)+1}-1)/(d-1) = (d^h-1)/(d-1)$. Thus, the total number of nodes in T is

$$\begin{aligned} n &= 1 + (d)(n_S) \\ &= 1 + d\left(\frac{d^h - 1}{d - 1}\right) \\ &= \frac{d-1}{d-1} + \frac{d^{h+1} - d}{d-1} \\ &= \frac{d^{h+1} - 1}{d-1} \end{aligned}$$

Thus, by the Principle of Mathematical Induction (see page 321), the formula must be correct for all full trees of any height.

10.3 Proof of Corollary 10.1 on page 188:

This proof is purely algebraic:

$$\begin{aligned} n &= \frac{d^{h+1}-1}{d-1} \\ n(d-1) &= d^{h+1}-1 \\ d^{h+1} &= n(d-1)+1 \\ &= nd-n+1 \\ h+1 &= \log_d(nd-n+1) \\ h &= \log_d(nd-n+1)-1 \end{aligned}$$

10.4 Proof of Corollary 10.2 on page 188:

Let T be a tree of any order d and any height h. Then T can be embedded into the full tree of the same degree and height. That full tree has exactly $\frac{d^{h+1}-1}{d-1}$ nodes, so its subtree T has at most that many nodes.

10.5 The path length of a full tree of order d and height h is $\frac{d}{(d-1)^2}[hd^{h+1}-(h+1)d+1]$. For example, the path length of the full tree on Figure 10.4 on page 188 is 102.

10.6 The tree diagram analysis of the St. Petersburg Paradox is shown in Figure 10.15. The flaw in this strategy is that there is a distinct possibility (i.e., a positive probability) that enough tails could come up in a row to make the required bet exceed the bettor's stake. After n successive tails, the bettor must bet $\$2^n$. For example, if 20 tails come up in a row, the next bet will have to be more than a million dollars!

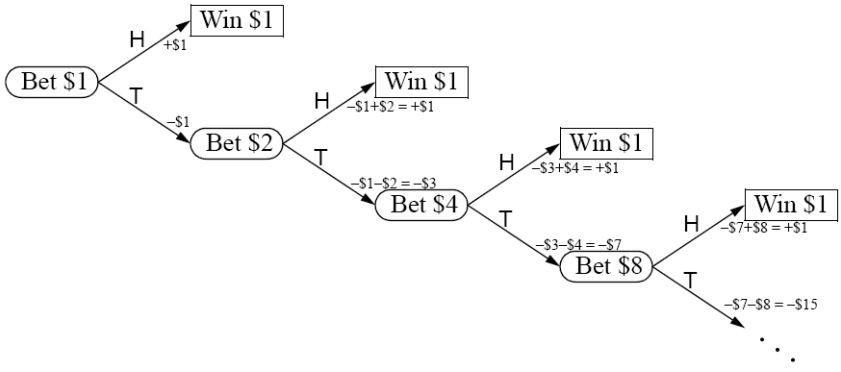

Figure 10.15 Analysis of the St. Petersburg Paradox

10.7 The decision tree for the version of craps where 3 can be a point is shown in Figure 10.16. The probability that X wins this version is 0.5068 or 50.68 percent.

10.8 The decision tree in Figure 10.17 shows all possible outcomes from the algorithm that solves the 7-coin problem.

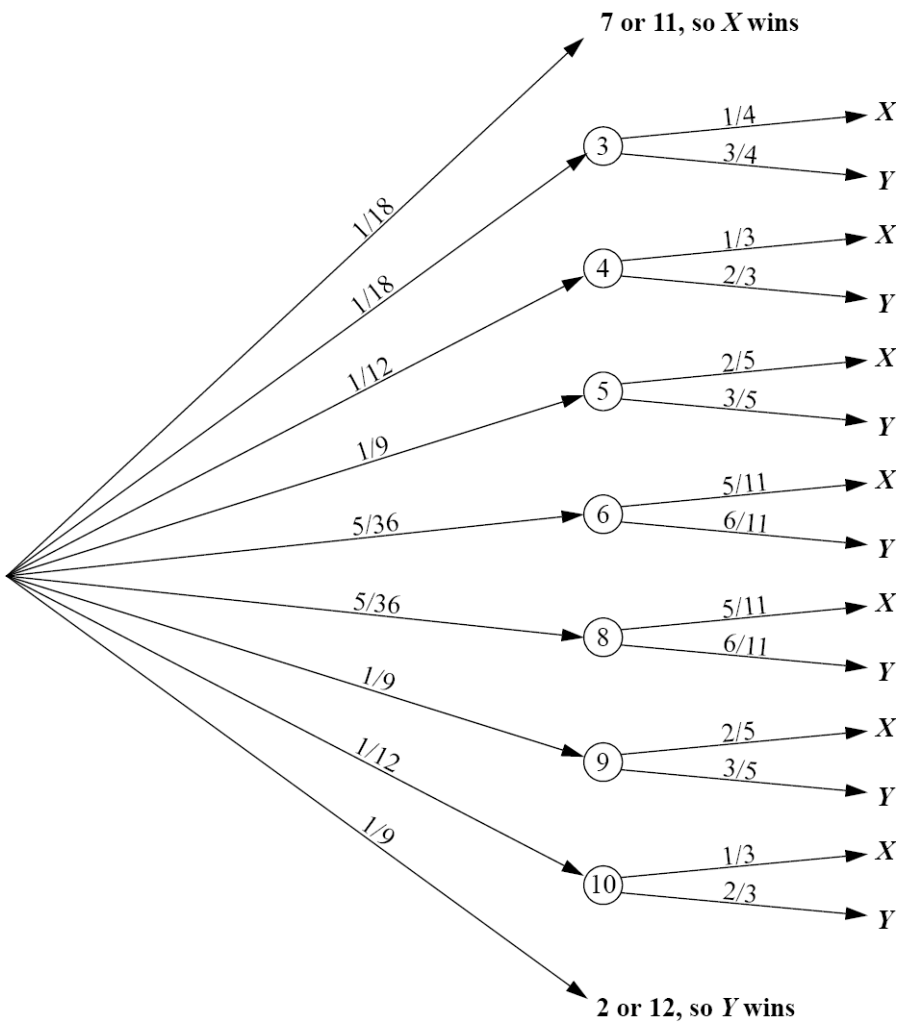

Figure 10.16 Decision tree for a craps game

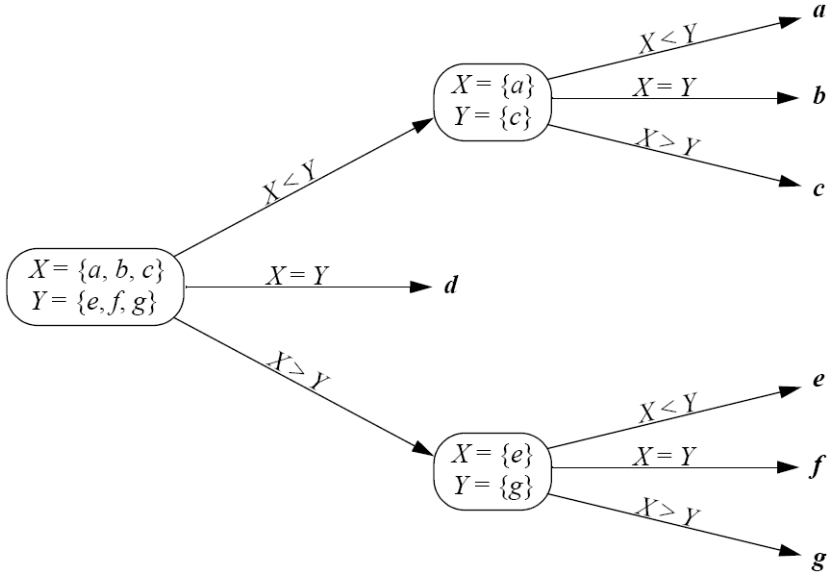

Figure 10.17 Decision tree for the 7-coin problem

CHAPTER 11

Binary Trees

DEFINITIONS

Here is the recursive definition of a binary tree:

> **A *binary tree* is either the empty set or a triple $T = (x, L, R)$, where x is a node and L and R are disjoint binary trees, neither of which contains x.**

The node x is called the *root* of the tree T, and the subtrees L and R are called the *left subtree* and the *right subtree* of T rooted at x.

Comparing this definition with the one on page 186, it is easy to see that a binary tree is just an ordered tree of order 2. But be aware that an empty left subtree is different from an empty right subtree. (See Example 10.5 on page 191.) Consequently, the two binary trees shown Figure 11.1 are not the same.

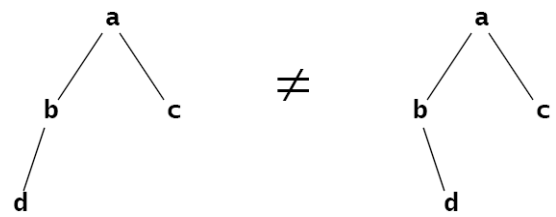

Figure 11.1 Unequal binary trees

Here is an equivalent, nonrecursive definition for binary trees:

> **A *binary tree* is an ordered tree in which every internal node has degree 2.**

In this simpler definition, the leaf nodes are regarded as dummy nodes whose only purpose is to define the structure of the tree. In applications, the internal nodes would hold data, while the leaf nodes would be either identical empty nodes, a single empty node, or just the `null` reference. This may seem inefficient and more complex, but it is usually easier to implement. In Figure 11.2, the dummy leaf nodes in the tree on the right are shown as asterisks.

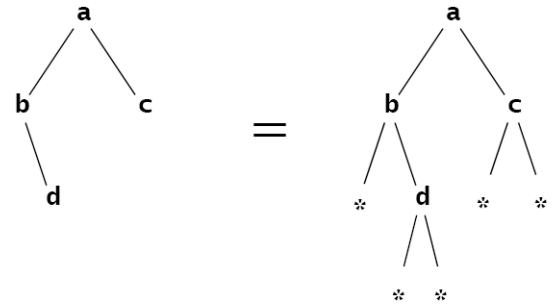

Figure 11.2 Equal binary trees

Except where noted, in this book we adhere to the first definition for binary trees. So some internal nodes may have only one child, either a left child or a right child.

The definitions of the terms *size*, *path*, *length* of a path, *depth* of a node, *level*, *height*, *interior* node, *ancestor*, *descendant*, *subtree*, and *supertree* are the same for binary trees as for general trees. (See page 186.)

EXAMPLE 11.1 Characteristics of a Binary Tree

Figure 11.3 shows a binary tree of size 10 and height 3. Node **a** is its root. The path from node **h** to node **b** has length 2. Node **b** is at level 1, and node **h** is at level 3. **b** is an ancestor of **h**, and **h** is a descendant of **b**. The part in the shaded region is a subtree of size 6 and height 2. Its root is node **b**.

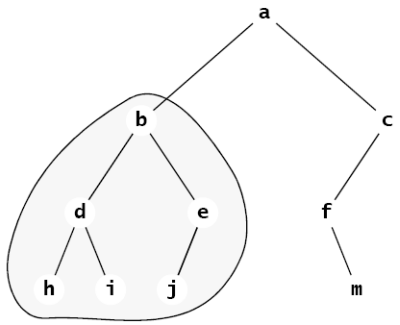

Figure 11.3 A binary tree

COUNTING BINARY TREES

EXAMPLE 11.2 All the Binary Trees of Size 3

There are five different binary trees of size $n = 3$, as shown in Figure 11.4.

Figure 11.4 The five different binary trees of size 3

Four have height 2, and the other one has height 1.

EXAMPLE 11.3 All the Binary Trees of Size 4

There are 14 different binary trees of size $n = 4$, as shown in Figure 11.5.

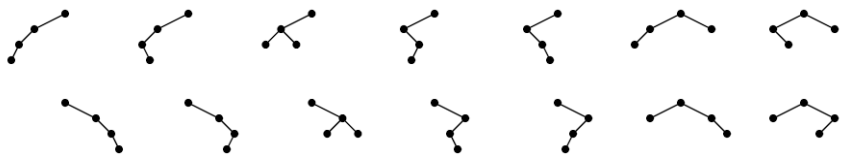

Figure 11.5 The 14 different binary trees of size 4

Ten have height 3, and the other four have height 2.

EXAMPLE 11.4 All the Binary Trees of Size 5

To find all the binary trees of size 5, apply the recursive definition for binary trees. If `t` is a binary tree of size 5, then it must consist of a root node together with two subtrees the sum of whose sizes equals 4. There are four possibilities: The left subtree contains either 4, 3, 2, 1, or 0 nodes.

First count all the binary trees of size 5 whose left subtree has size 4. From Example 11.3, we see that there are 14 different possibilities for that left subtree. But for each of those 14 choices, there are no other options because the right subtree must be empty. Therefore, there are 14 different binary trees of size 5 whose left subtree has size 4.

Next, count all the binary trees of size 5 whose left subtree has size 3. From Example 11.2, we see that there are five different possibilities for that left subtree. But for each of those five choices, there are no

other options because the right subtree must be a singleton. Therefore, there are five different binary trees of size 5 whose left subtree has size 3.

Next, count all the binary trees of size 5 whose left subtree has size 2. There are only two different possibilities for that left subtree. But for each of those two choices, we have the same two different possibilities for the right subtree because it also must have size 2. Therefore, there are $2 \times 2 = 4$ different binary trees of size 5 whose left subtree has size 2.

By similar reasoning, we find that there are five different binary trees of size 5 whose left subtree has size 1, and there are 14 different binary trees of size 5 whose left subtree has size 0. Therefore, the total number of different binary trees of size 5 is $14 + 5 + 4 + 5 + 14 = 42$.

FULL BINARY TREES

A binary tree is said to be *full* if all its leaves are at the same level and every interior node has two children.

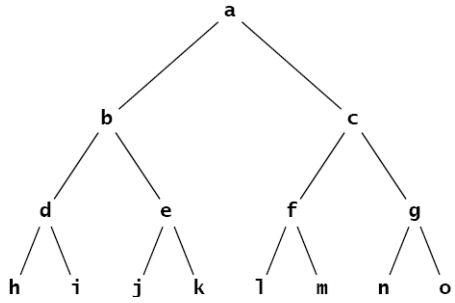

Figure 11.6 A full binary tree of height 3

EXAMPLE 11.5 The Full Binary Tree of Height 3

The tree shown in Figure 11.6 is the full binary tree of height 3. Note that it has 15 nodes: 7 interior nodes and 8 leaves.

Theorem 11.1 The full binary tree of height h has $l = 2^h$ leaves and $m = 2^h - 1$ internal nodes.

Proof: The full binary tree of height $h = 0$ is a single leaf node, so it has $n = 1$ node, which is a leaf. Therefore, since $2^h - 1 = 2^0 - 1 = 1 - 1 = 0$, and $2^h = 2^0 = 1$, the formulas are correct for the case where $h = 0$. More generally, let $h > 0$ and assume (the inductive hypothesis) that the formulas are true for all full binary trees of height less than h. Then consider a full binary tree of height h. Each of its subtrees has height $h - 1$, so we apply the formulas to them: $l_L = l_R = 2^{h-1}$ and $m_L = m_R = 2^{h-1} - 1$. (These are the number of leaves in the left subtree, the number of leaves in the right subtree, the number of internal nodes in the left subtree, and the number of internal nodes in the right subtree, respectively.) Then

$$l = l_L + l_R = 2^{h-1} + 2^{h-1} = 2 \cdot 2^{h-1} = 2^h$$

and

$$m = m_L + m_R + 1 = (2^{h-1} - 1) + (2^{h-1} - 1) + 1 = 2 \cdot 2^{h-1} - 1 = 2^I - 1$$

Therefore, by the (Second) Principle of Mathematical Induction, the formulas must be true for full binary trees of any height $h \geq 0$.

By simply adding the formulas for m and l, we obtain the first corollary.

Corollary 11.1 The full binary tree of height h has a total of $n = 2^{h+1} - 1$ nodes.

By solving the formula $n = 2^{h+1} - 1$ for h, we obtain this corollary:

Corollary 11.2 The full binary tree with n nodes has height $h = \lg(n+1) - 1$.

Note that the formula in Corollary 11.2 is correct even in the special case where $n = 0$: The *empty binary tree* has height $h = \lg(n+1) - 1 = \lg(0+1) - 1 = \lg(1) - 1 = 0 - 1 = -1$.

The next corollary applies Corollary 11.1 together with the fact that the full binary tree of height h has more nodes than any other binary tree of height h.

Corollary 11.3 In any binary tree of height h,

$$h + 1 \le n \le 2^{h+1} - 1 \text{ and } \lfloor \lg n \rfloor \le h \le n-1$$

where n is the number of its nodes.

IDENTITY, EQUALITY, AND ISOMORPHISM

In a computer, two objects are *identically equal* if they occupy the same space in memory, so they have the same address. In other words, there really only one object, but with two different names. That meaning of equality is reflected in Java by the equality operator. If `x` and `y` are references to objects, then the condition `(x == y)` will be true only if `x` and `y` both refer to the same object.

But the normal concept of equality in mathematics is that the two things have the same value. This distinction is handled in Java by the `equals()` method, defined in the `Object` class (see Chapter 4) and thus inherited by every class. As defined there, it has the same effect as the equals operator: `x.equals(y)` means `x == y`. But that `equals()` method is intended to be overridden in subclasses so that it will return `true` not only when the two objects are identically equal, but also when they are separate objects that are "the same" in whatever sense the class designer intends. For example, `x.equals(y)` could be defined to be `true` for distinct instances `x` and `y` of `Point` class if they have the same coordinates.

EXAMPLE 11.6 Testing Equality of Strings

```
1   public class TestStringEquality {
2     static public void main(String[] args) {
3       String x = new String("ABCDE");
4       String y = new String("ABCDE");
5       System.out.println("x = " + x);
6       System.out.println("y = " + y);
7       System.out.println("(x == y) = " + (x == y));
8       System.out.println("x.equals(y) = " + x.equals(y));
9     }
10  }
```

The output is:

```
x = ABCDE
y = ABCDE
(x == y) = false
x.equals(y) = true
```

Here, the two objects `x` and `y` (or, more precisely, the two objects that are referenced by the reference variables `x` and `y`) are different objects, occupying different memory locations, so they are not identically equal: `(x == y)` evaluates to `false` at line 7. But they do both have the same contents, so they are mathematically equal, and `x.equals(y)` evaluates to `true` at line 8.

The distinction between identical equality and mathematical equality exists in Java only for reference variables (i.e., only for objects). For all variables of primitive types, the equality operator tests for mathematical equality.

Data structures have both content and structure. So it is possible for two data structures to have equal contents (i.e., have the same contents) but be organized differently. For example, two arrays could both contain the three numbers 22, 44, and 88, but in different orders.

EXAMPLE 11.7 Testing Equality of Arrays

```
1     public class TestArraysEquality {
2       public static void main(String[] args) {
3         int[] x = { 22, 44, 88 };
4         int[] y = { 88, 44, 22 };
5         ch02.ex02.DuplicatingArrays.print(x);
6         ch02.ex02.DuplicatingArrays.print(y);
7         System.out.println("Arrays.equals(x, y) = " + Arrays.equals(x, y));
8         Arrays.sort(x);
9         Arrays.sort(y);
10        ch02.ex02.DuplicatingArrays.print(x);
11        ch02.ex02.DuplicatingArrays.print(y);
12        System.out.println("Arrays.equals(x, y) = " + Arrays.equals(x, y));
13      }
14    }
```

The output it:

```
{22, 44, 88}
{88, 44, 22}
Arrays.equals(x, y) = false
{22, 44, 88}
{22, 44, 88}
Arrays.equals(x, y) = true
```

This shows that the `java.util.Arrays.equal()` method requires not only the same contents for arrays to be equal, but also in the same order, as would be expected.

Equality is a weaker relation than identity: Identical objects are always equal, but equal objects may not be identical; they could be distinct. Equality of data structures means the same structure and the same contents in the same order.

A weaker kind of reflexive relation is isomorphism. Two data structures are isomorphic if they have the same structure. This concept is used when the "data" part of the data structure is irrelevant.

Two arrays are isomorphic if they have the same length.

Two trees are isomorphic if one tree can be rearranged to match the other. More formally, T_1 is *isomorphic* to T_2 (sometimes written $T_1 \cong T_2$) if there is a one-to-one mapping (an *isomorphism*) between them that preserves parent-child relationship between all nodes.

EXAMPLE 11.8 Isomorphic Trees

As unordered trees, Tree 1 and Tree 2 in Figure 11.7 are isomorphic, but not equal.

However, Tree 3 is not isomorphic to either of the other two trees because it has only three leaves; the other two trees each have four leaves:Tthat's a different structure. That distinction leads fairly easily to a formal deduction that there is no isomorphism between Tree 1 and Tree 3.

As ordered trees, Tree 1 is not isomorphic to Tree 2 because their roots' left-most subtrees have different sizes. The left-most subtree in Tree 1 has three nodes, while that of Tree 2 has only two nodes. That distinction also leads fairly easily to a formal deduction that no isomorphism between Tree 1 and Tree 2 can exist.

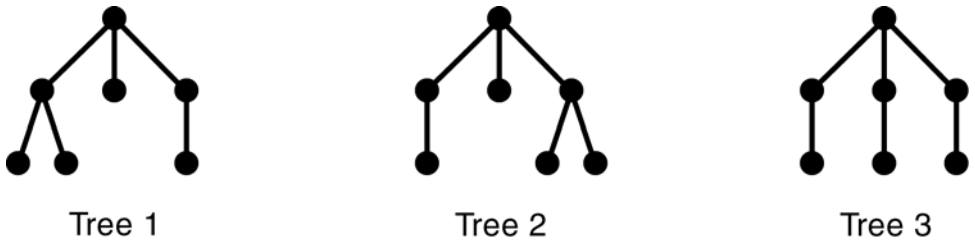

Figure 11.7 Isomorphic and nonisomorphic trees

Binary trees are ordered trees. The order of the two children at each node is part of the structure of the binary tree.

Binary trees are ordered trees. So any isomorphism between binary trees must preserve the order of each node's children.

EXAMPLE 11.9 Nonisomorphic Binary Trees

Figure 11.8 Nonisomorphic binary trees

In Figure 11.8, Binary Tree 1 is not isomorphic to Binary Tree 2, for the same reason that the ordered trees in Example 11.8 are not isomorphic: The subtrees don't all match, as ordered trees. In Tree 1, the root's right child has a left child; but in Tree 1, the root's right child has no (nonempty) left child.

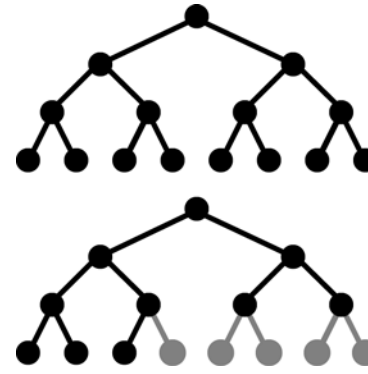

Figure 11.9 Complete binary trees

COMPLETE BINARY TREES

A *complete binary tree* is either a full binary tree or one that is full except for a segment of missing leaves on the right side of the bottom level.

EXAMPLE 11.10 A Complete Binary Tree of Height 3

The tree shown in Figure 11.9 is complete. It is shown together with the full binary tree from which it was obtained by adding five leaves on the right at level 3.

Theorem 11.2 In a complete binary tree of height h,

$$h + 1 \leq n \leq 2^{h+1} - 1 \text{ and } h = \lfloor \lg n \rfloor$$

where n is the number of its nodes.

EXAMPLE 11.11 More Complete Binary Trees

Figure 11.10 shows three more examples of complete binary trees.

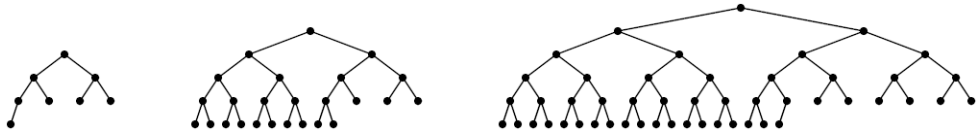

Figure 11.10 Complete binary trees

Complete binary trees are important because they have a simple and natural implementation using ordinary arrays. The *natural mapping* is actually defined for any binary tree: Assign the number 1 to the root; for any node, if i is its number, then assign $2i$ to its left child and $2i+1$ to its right child (if they exist). This assigns a unique positive integer to each node. Then simply store the element at node `i` in `a[i]`, where `a[]` is an array.

Complete binary trees are important because of the simple way in which they can be stored in an array. This is achieved by assigning index numbers to the tree nodes by level, as shown in Figure 11.11. The beauty in this natural mapping is the simple way that it allows the array indexes of the children and parent of a node to be computed.

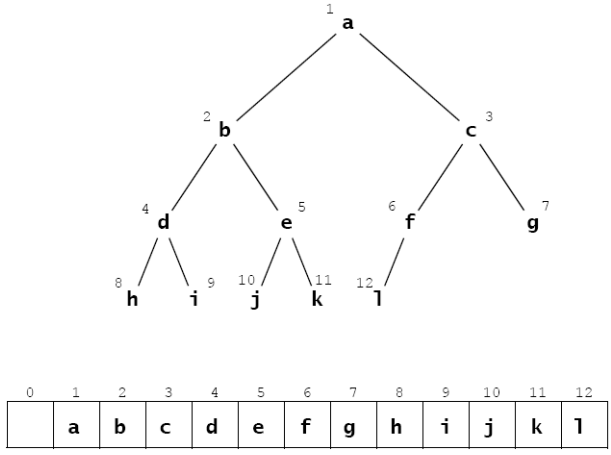

Figure 11.11 The natural mapping of a complete binary tree

Algorithm 11.1 The Natural Mapping of a Complete Binary Tree into an Array

To navigate about a complete binary tree stored by its natural mapping in an array:

1. The parent of the node stored at location i is stored at location $i/2$.
2. The left child of the node stored at location i is stored at location $2i$.
3. The right child of the node stored at location i is stored at location $2i + 1$.

For example, node **e** is stored at index $i = 5$ in the array; its parent node **b** is stored at index $i/2 = 5/2 = 2$, its left child node **j** is stored at location $2i = 2 \cdot 5 = 10$, and its right child node **k** is stored at index $2i + 1 = 2 \cdot 5 + 1 = 11$.

The use of the adjective "complete" should now be clear: The defining property for complete binary trees is precisely the condition that guarantees that the natural mapping will store the tree nodes "completely" in an array with no gaps.

EXAMPLE 11.12 An Incomplete Binary Tree

Figure 11.12 shows the incomplete binary tree from Example 11.1 on page 201. The natural mapping of its nodes into an array leaves some gaps, as shown in Figure 11.13.

Note: Some authors use the term "almost complete binary tree" for a complete binary tree and the term "complete binary tree" for a full binary tree.

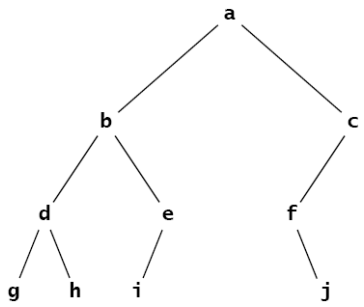

Figure 11.12 A binary tree

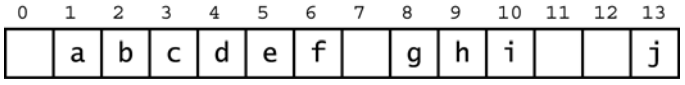

Figure 11.13 The natural mapping of an incomplete binary tree

BINARY TREE TRAVERSAL ALGORITHMS

The three traversal algorithms that are used for general trees (see Chapter 10) apply to binary trees as well: the preorder traversal, the postorder traversal, and the level order traversal. In addition, binary trees support a fourth traversal algorithm: the inorder traversal. These four traversal algorithms are given next.

Algorithm 11.2 The Level Order Traversal of a Binary Tree

To traverse a nonempty binary tree:

1. Initialize a queue.
2. Enqueue the root.
3. Repeat steps 4–7 until the queue is empty.
4. Dequeue a node x from the queue.
5. Visit x.
6. Enqueue the left child of x if it exists.
7. Enqueue the right child of x if it exists.

EXAMPLE 11.13 The Level Order Traversal of a Binary Tree

Figure 11.14 on page 207 shows how the level order traversal looks on the full binary tree of height 3.

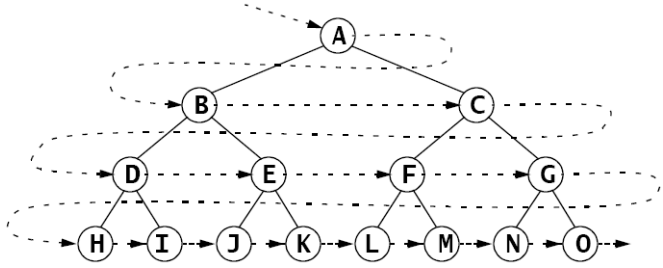

Figure 11.14 The level order traversal of a binary tree

The nodes are visited in the order **A**, **B**, **C**, **D**, **E**, **F**, **G**, **H**, **I**, **J**, **K**, **L**, **M**, **N**, **O**.

Algorithm 11.3 The Preorder Traversal of a Binary Tree

To traverse a nonempty binary tree:

1. Visit the root.
2. If the left subtree is nonempty, do a preorder traversal on it.
3. If the right subtree is nonempty, do a preorder traversal on it.

EXAMPLE 11.14 The Preorder Traversal of a Binary Tree

Figure 11.15 on page 208 shows the preorder traversal on the full binary tree of height 3.

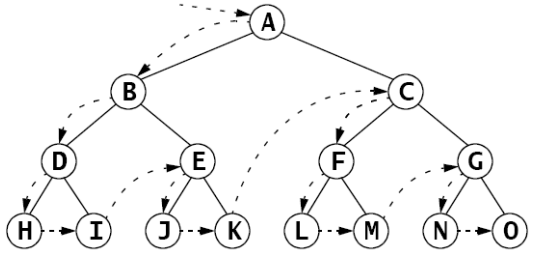

Figure 11.15 The preorder traversal of a binary tree

The nodes are visited in the order **A**, **B**, **D**, **H**, **I**, **E**, **J**, **K**, **C**, **F**, **L**, **M**, **G**, **N**, **O**.

Figure 11.16 shows how the preorder traversal of a binary tree can be obtained by circumnavigating the tree, beginning at the root and visiting each node the first time it is encountered on the left:

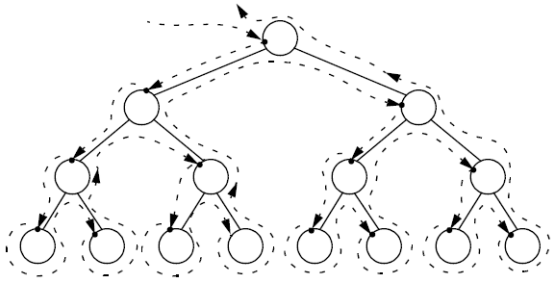

Figure 11.16 The preorder traversal of a binary tree

Algorithm 11.4 The Postorder Traversal of a Binary Tree

To traverse a nonempty binary tree:

1. If the left subtree is nonempty, do a postorder traversal on it.
2. If the right subtree is nonempty, do a postorder traversal on it.
3. Visit the root.

EXAMPLE 11.15 The Postorder Traversal of a Binary Tree

Figure 11.17 shows the postorder traversal looks on the full binary tree of height 3.

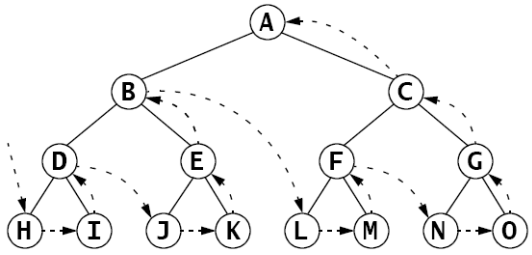

Figure 11.17 The postorder traversal of a binary tree

The nodes are visited in the order **H**, **I**, **D**, **J**, **K**, **E**, **B**, **L**, **M**, **F**, **N**, **O**, **G**, **C**, **A**.

The preorder traversal visits the root first and the postorder traversal visits the root last. This suggests a third alternative for binary trees: Visit the root in between the traversals of the two subtrees. That is called the *inorder traversal*.

Algorithm 11.5 The Inorder Traversal of a Binary Tree

To traverse a nonempty binary tree:

1. If the left subtree is nonempty, do a preorder traversal on it.
2. Visit the root.
3. If the right subtree is nonempty, do a preorder traversal on it.

EXAMPLE 11.16 The Inorder Traversal of a Binary Tree

Figure 11.18 shows how the inorder traversal looks on the full binary tree of height 3.

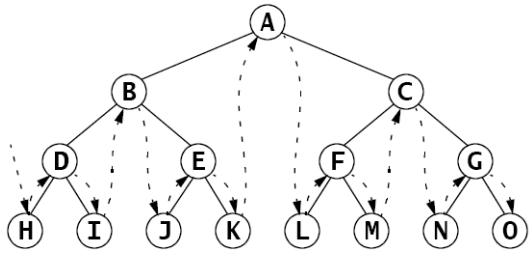

Figure 11.18 The inorder traversal of a binary tree

The nodes are visited in the order **H**, **D**, **I**, **B**, **J**, **E**, **K**, **A**, **L**, **F**, **M**, **C**, **N**, **G**, **O**.

EXPRESSION TREES

An *arithmetic expression* such as `(5 - x)*y + 6/(x + z)` is a combination of *arithmetic operators* (`+`, `-`, `*`, `/`, etc.), *operands* (`5`, `x`, `y`, `6`, `z`, etc.), and parentheses to override the precedence of operations. Each expression can be represented by a unique binary tree whose structure is determined by the precedence of operations in the expression. Such a tree is called an *expression tree*.

EXAMPLE 11.17 An Expression Tree

Figure 11.19 shows the expression tree for the expression `(5 – x)*y + 6/(x + z)`.

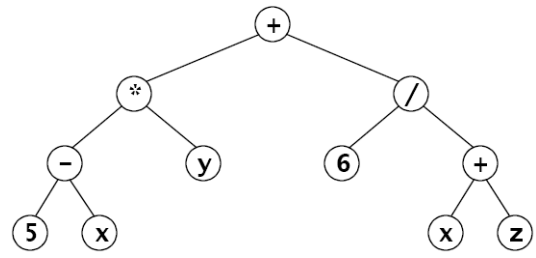

Figure 11.19 An expression tree

Here is a recursive algorithm for building an expression tree:

Algorithm 11.6 Build an Expression Tree
The expression tree for a given expression can be built recursively from the following rules:

1. The expression tree for a single operand is a single root node that contains it.
2. If E_1 and E_2 are expressions represented by expression trees T_1 and T_2, and if *op* is an operator, then the expression tree for the expression E_1 *op* E_2 is the tree with root node containing *op* and subtrees T_1 and T_2.

An expression has three representations, depending upon which traversal algorithm is used to traverse its tree. The preorder traversal produces the *prefix representation*, the inorder traversal produces the *infix representation*, and the postorder traversal produces the *postfix representation* of the expression. The postfix representation is also called *reverse Polish notation* or *RPN*. These are outlined on page 109.

EXAMPLE 11.18 The Three Representations of an Expression

The three representations for the expression in Example 11.17 are:

Prefix:	`+*-5xy/6+xz`
Infix:	`5-x*y+6/x+z`
Postfix (RPN):	`5x-y*6xz+/+`

Ordinary function syntax uses the prefix representation. The expression in Example 11.17 could be evaluated as

```
sum(product(difference(5, x), y), quotient(6, sum(x, z)))
```

Some scientific calculators use RPN, requiring both operands to be entered before the operator.

The next algorithm can be applied to a postfix expression to obtain its value.

Algorithm 11.7 Evaluating an Expression from Its Postfix Representation
To evaluate an expression represented in postfix, scan the representation from left to right:

1. Create a stack for operands.
2. Repeat steps 3–9 until the end of representation is reached.
3. Read the next token *t* from the representation.
4. If it is an operand, push its value onto the stack.
5. Otherwise, do steps 6–9:
6. Pop *a* from the stack.
7. Pop *b* from the stack.
8. Evaluate *c* = *a t b*.
9. Push *c* onto the stack.
10. Return the top element on the stack.

EXAMPLE 11.19 Evaluating an Expression from Its Postfix Representation

Figure 11.20 shows the evaluation of the expression in Example 11.18 using 2 for *x*, 3 for *y*, and 1 for *z*:

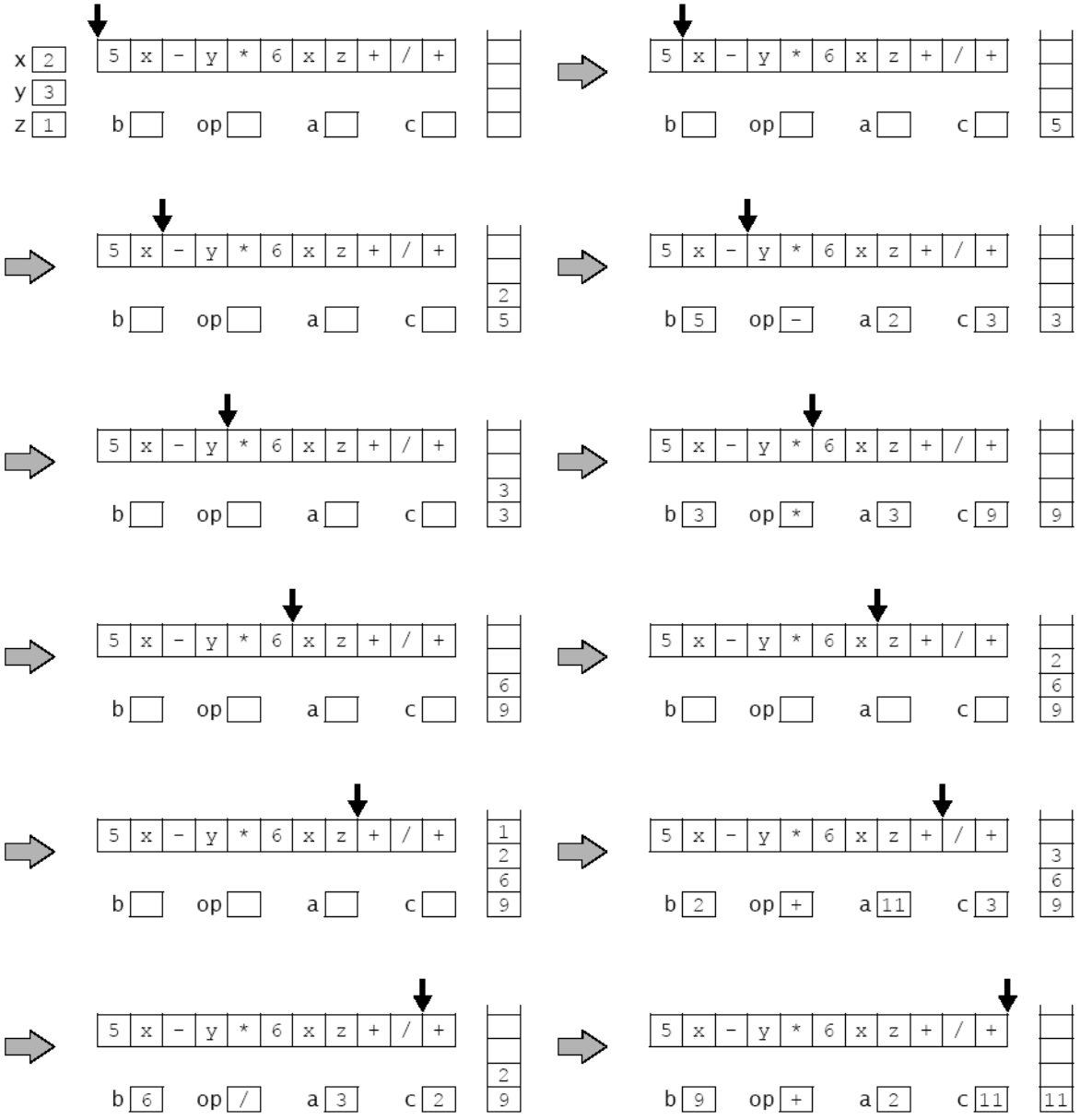

Figure 11.20 Evaluating a postfix expression

A `BinaryTree` CLASS

Here is a class for binary trees that directly implements the recursive definition. (See page 200.) By extending the `AbstractCollection` class, it remains consistent with the Java Collections Framework. (See Chapter 4.)

EXAMPLE 11.20 A `BinaryTree` Class

```
public class BinaryTree<E> extends AbstractCollection {
  protected E root;
  protected BinaryTree<E> left, right, parent;
  protected int size;

  public BinaryTree() {
  }

  public BinaryTree(E root) {
    this.root = root;
    size = 1;
  }

  public BinaryTree(E root, BinaryTree<E> left, BinaryTree<E> right) {
    this(root);
    if (left != null) {
      this.left = left;
      left.parent = this;
      size += left.size();
    }
    if (right != null) {
      this.right = right;
      right.parent = this;
      size += right.size();
    }
  }

  public boolean equals(Object object) {
    if (object == this) {
      return true;
    } else if (!(object instanceof BinaryTree)) {
      return false;
    }
    BinaryTree that = (BinaryTree)object;
    return that.root.equals(this.root)
        && that.left.equals(this.left)
        && that.right.equals(this.right)
        && that.parent.equals(this.parent)
        && that.size == this.size;
  }

  public int hashCode() {
    return root.hashCode() + left.hashCode() + right.hashCode() + size;
  }

  public int size() {
    return size;
  }

  public Iterator iterator() {
    return new java.util.Iterator() { // anonymous inner class
      private boolean rootDone;
      private Iterator lIt, rIt;  // child iterators
```

```
54          public boolean hasNext() {
55            return !rootDone || lIt != null && lIt.hasNext()
56                             || rIt != null && rIt.hasNext();
57          }
58          public Object next() {
59            if (rootDone) {
60              if (lIt != null && lIt.hasNext()) {
61                return lIt.next();
62              }
63              if (rIt != null && rIt.hasNext()) {
64                return rIt.next();
65              }
66              return null;
67            }
68            if (left != null) {
69              lIt = left.iterator();
70            }
71            if (right != null) {
72              rIt = right.iterator();
73            }
74            rootDone = true;
75            return root;
76          }
77          public void remove() {
78            throw new UnsupportedOperationException();
79          }
80        };
81      }
82    }
```

The `java.util.AbstractCollection` class requires the four methods that are defined here: `equals()`, `hashCode()`, `iterator()`, and `size()`.[1]

The `iterator()` method overrides the empty version that is defined in the `AbstractCollection` class. Its job is to build an iterator object that can traverse its `BinaryTree` object. To do that, it creates its own anonymous inner `Iterator` class using the Java `return new` construct at line 47. The body of this anonymous class is defined between the braces that immediately follow the invocation of the constructor `Iterator()`. Note that this block must be followed by a semicolon because it is actually the end of the `return` statement. The complete construct looks like a method definition, but it is not. It really is a complete class definition embedded within a `return` statement.

To return an `Iterator` object, this anonymous class must implement the `Iterator` interface. (See page 77.) This requires definitions for the three methods

```
public boolean hasNext()  ...
public Object next()  ...
public void remove()  ...
```

This implementation is recursive. The `hasNext()` method invokes the `hasNext()` methods of iterators on the two subtrees, and the `next()` method invokes the `next()` methods of those two iterators, named `lIt` and `rIt`. The other local variable is a flag named `rootDone` that keeps track of whether the root object has been visited yet by the iterator.

The `hasNext()` method returns `true` unless all three parts of the tree have been visited: the root, the left subtree, and the right subtree. It does that by using the `lIt` and `rIt` iterators recursively.

---

1. Actually, the `equals()` and `hashCode()` methods are defined in the `Object` class and do not have to be overridden.

The `next()` method also uses the `lIt` and `rIt` iterators recursively. If the `root` has already been visited, then the iterator visits the next node in the left subtree if there are any, and otherwise visits the next node in the right subtree if there are any. If the root has not yet been visited, then this must be the first call to the iterator on that particular subtree, so it initializes the `lIt` and `rIt` iterators, sets the `rootDone` flag, and returns the `root`.

The `remove()` method is not implemented because there is no simple way to remove an internal node from a binary tree.

EXAMPLE 11.21 Testing the `BinaryTree` Class

```
1    public class TestBinaryTree {
2      static public void main(String[] args) {
3        BinaryTree<String> e = new BinaryTree<String>("E");
4        BinaryTree<String> g = new BinaryTree<String>("G");
5        BinaryTree<String> h = new BinaryTree<String>("H");
6        BinaryTree<String> i = new BinaryTree<String>("I");
7        BinaryTree<String> d = new BinaryTree<String>("D", null, g);
8        BinaryTree<String> f = new BinaryTree<String>("F", h, i);
9        BinaryTree<String> b = new BinaryTree<String>("B", d, e);
10       BinaryTree<String> c = new BinaryTree<String>("C", f, null);
11       BinaryTree<String> tree = new BinaryTree<String>("A", b, c);
12       System.out.printf("tree: %s", tree);
13     }
14   }
```

The output is:

```
tree: [A, B, D, G, E, C, F, H, I]
```

The program creates the binary tree shown in Figure 11.21 and then indirectly invokes its `toString()` method that it inherits from the `AbstractCollections` class.

Figure 11.21 shows two views of the same tree. The larger view shows all the details, representing each object reference with an arrow.

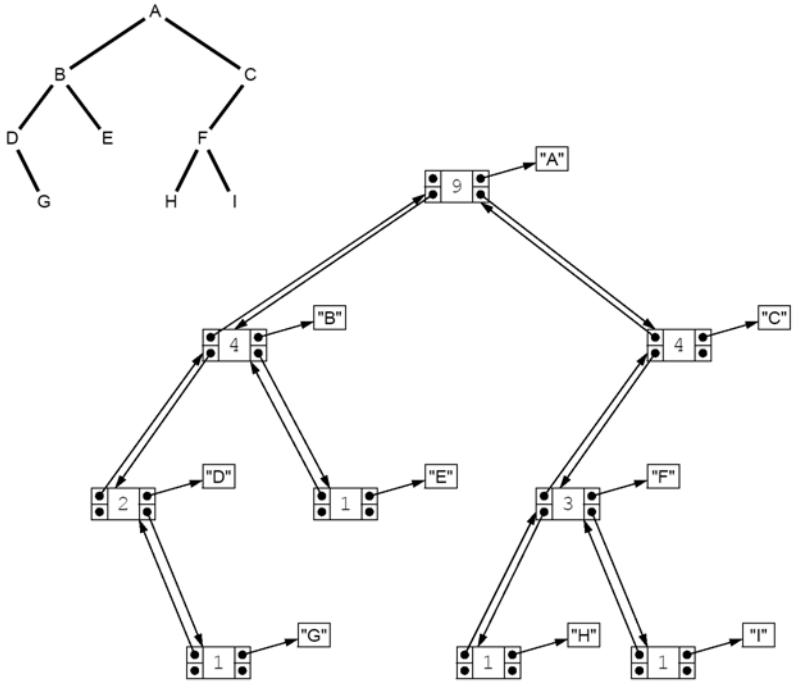

Figure 11.21 The binary tree constructed in Example 11.21

By extending the `AbstractCollection` class, the `BinaryTree` class automatically inherits these methods that are defined by using the `iterator()` and `size()` methods:

```
public boolean  isEmpty()
public boolean  contains(Object object)
public Object[] toArray()
public Object[] toArray(Object[] objects)
public String   toString()
public boolean  add(Object object)
public boolean  addAll(Collection collection)
public void     clear()
public boolean  containsAll(Collection collection)
public boolean  remove(Object object)
public boolean  removeAll(Collection collection)
public boolean  retainAll(Collection collection)
```

However, the mutating methods will throw an `UnsupportedOperationException` because they invoke other methods that are not implemented, namely the `add()` and the `Iterator.remove()` methods.

EXAMPLE 11.22 Testing the `contains()` Method on a Binary Tree

This example builds the same tree as the one in Example 11.21 and then tests the `contains()` method on it and its subtrees:

```
1   public class TestContains {
2     static public void main(String[] args) {
3       BinaryTree<String> e = new BinaryTree<String>("E");
4       BinaryTree<String> g = new BinaryTree<String>("G");
5       BinaryTree<String> h = new BinaryTree<String>("H");
6       BinaryTree<String> i = new BinaryTree<String>("I");
7       BinaryTree<String> d = new BinaryTree<String>("D", null, g);
8       BinaryTree<String> f = new BinaryTree<String>("F", h, i);
9       BinaryTree<String> b = new BinaryTree<String>("B", d, e);
10      BinaryTree<String> c = new BinaryTree<String>("C", f, null);
11      BinaryTree<String> a = new BinaryTree<String>("A", b, c);
12      System.out.printf("a: %s%n", a);
13      System.out.println("a.contains(\"H\") = " + a.contains("H"));
14      System.out.printf("b: %s%n", b);
15      System.out.println("b.contains(\"H\") = " + b.contains("H"));
16      System.out.printf("c: %s%n", c);
17      System.out.println("c.contains(\"H\") = " + c.contains("H"));
18    }
19  }
```

The output is:

```
a: [A, B, D, G, E, C, F, H, I]
a.contains("H") = true
b: [B, D, G, E]
b.contains("H") = false
c: [C, F, H, I]
c.contains("H") = true
```

The subtrees `b` and `c` are shown in Figure 11.22. The tree `a` contains the element `H`. The subtree `b` does not contain the element `H`. The subtree `c` does contain the element `H`.

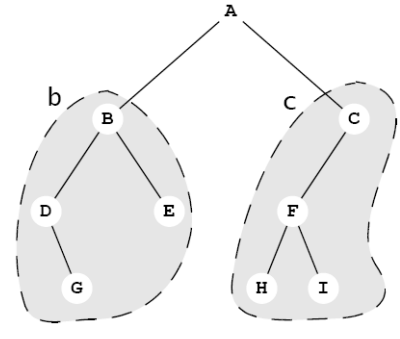

Figure 11.22

IMPLEMENTATIONS OF THE TRAVERSAL ALGORITHMS

The iterator that is returned by the `iterator()` method follows the preorder traversal algorithm (Algorithm 11.3 on page 208) to traverse the binary tree. The following modification of the `BinaryTree` class implements all four of the binary tree traversal algorithms.

EXAMPLE 11.23 Implementing the Four Traversal Algorithms

```
1   public class BinaryTree<E> extends AbstractCollection {
2     //  insert lines 2-49 from Example 11.20 on page 212
50    public Iterator iterator() {
51      return new PreOrder();
52    }
53
54    abstract public class BinaryTreeIterator implements Iterator {
55      protected boolean rootDone;
56      protected Iterator lIt, rIt;  // child iterators
57      public boolean hasNext() {
58        return !rootDone || lIt != null && lIt.hasNext()
59                         || rIt != null && rIt.hasNext();
60      }
61      abstract public Object next();
62      public void remove() {
63        throw new UnsupportedOperationException();
64      }
65    }
66
67    public class PreOrder extends BinaryTreeIterator {
68      public PreOrder() {
69        if (left != null) {
70          lIt = left.new PreOrder();
71        }
72        if (right != null) {
73          rIt = right.new PreOrder();
74        }
75      }
76      public Object next() {
77        if (!rootDone) {
78          rootDone = true;
79          return root;
80        }
81        if (lIt != null && lIt.hasNext()) {
82          return lIt.next();
83        }
84        if (rIt != null && rIt.hasNext()) {
85          return rIt.next();
86        }
87        return null;
88      }
89    }
90
91    public class InOrder extends BinaryTreeIterator {
92      public InOrder() {
93        if (left != null) {
94          lIt = left.new InOrder();
95        }
```

```
96          if (right != null) {
97            rIt = right.new InOrder();
98          }
99        }
100       public Object next() {
101         if (lIt != null && lIt.hasNext()) {
102           return lIt.next();
103         }
104         if (!rootDone) {
105           rootDone = true;
106           return root;
107         }
108         if (rIt != null && rIt.hasNext()) {
109           return rIt.next();
110         }
111         return null;
112       }
113     }
114
115     public class PostOrder extends BinaryTreeIterator {
116       public PostOrder() {
117         if (left != null) {
118           lIt = left.new PostOrder();
119         }
120         if (right != null) {
121           rIt = right.new PostOrder();
122         }
123       }
124       public Object next() {
125         if (lIt != null && lIt.hasNext()) {
126           return lIt.next();
127         }
128         if (rIt != null && rIt.hasNext()) {
129           return rIt.next();
130         }
131         if (!rootDone) {
132           rootDone = true;
133           return root;
134         }
135         return null;
136       }
137     }
138
139     public class LevelOrder extends BinaryTreeIterator {
140       Queue<BinaryTree<E>> queue = new ArrayDeque<BinaryTree<E>>();
141       public boolean hasNext() {
142         return (!rootDone || !queue.isEmpty());
143       }
144       public Object next() {
145         if (!rootDone) {
146           if (left != null) {
147             queue.add(left);
148           }
149           if (right != null) {
150             queue.add(right);
151           }
```

```
152             rootDone = true;
153             return root;
154           }
155           if (!queue.isEmpty()) {
156             BinaryTree<E> tree = queue.remove();
157             if (tree.left != null) {
158               queue.add(tree.left);
159             }
160             if (tree.right != null) {
161               queue.add(tree.right);
162             }
163             return tree.root;
164           }
165           return null;
166         }
167       }
168     }
```

At line 64 we define an abstract inner class named `BinaryTreeIterator`. This serves as a base class for all four of the concrete iterator classes. It declares the same three fields (`rootDone`, `rIt`, and `lIt`) as the anonymous iterator class defined previously.

The `hasNext()` and `remove()` methods are implemented (at lines 57 and 62) the same way the abstract `Iterator` class was done in the anonymous iterator class. But the `next()` method is declared `abstract` because each of the four traversal algorithms has a different implementation of it.

The `PreOrder` class defines the `lIt` and `rIt` iterators to be `PreOrder` iterators in its constructor to ensure that the recursive traversal follows the preorder traversal algorithm. That algorithm (Algorithm 11.3 on page 208) says to visit the root first, and then apply the same algorithm recursively to the left subtree and then to the right subtree. The three `if` statements do that at lies 77–86. The only differences between the `PreOrder`, `InOrder`, and `PostOrder` classes are their definitions of the recursive `rIt` and `lIt` iterators in the constructors and the order of those three `if` statements in the `next()` methods. For the `InOrder` class, the order visits the root between the two recursive traversals. For the `PostOrder` class, the order visits the root after the two recursive traversals. ("Pre" means before, "in" means between, and "post" means after.)

The `LevelOrder` traversal class is significantly different from the other three. Instead of being recursive, it uses a queue. (See Algorithm 11.5 on page 209.)

EXAMPLE 11.24 Testing the Traversal Algorithms

```
1   public class TestIterators {
2     public static void main(String[] args) {
3       BinaryTree<String> e = new BinaryTree<String>("E");
4       BinaryTree<String> g = new BinaryTree<String>("G");
5       BinaryTree<String> h = new BinaryTree<String>("H");
6       BinaryTree<String> i = new BinaryTree<String>("I");
7       BinaryTree<String> d = new BinaryTree<String>("D",null,g);
8       BinaryTree<String> f = new BinaryTree<String>("F",h,i);
9       BinaryTree<String> b = new BinaryTree<String>("B",d,e);
10      BinaryTree<String> c = new BinaryTree<String>("C",f,null);
11      BinaryTree<String> tree = new BinaryTree<String>("A",b,c);
12      System.out.println("tree = " + tree);
13      java.util.Iterator it;
14      System.out.print("PreOrder Traversal:   ");
15      for (it = tree.new PreOrder(); it.hasNext(); ) {
16        System.out.print(it.next() + " ");
17      }
18      System.out.print("\nInOrder Traversal:    ");
```

```
        for (it = tree.new InOrder(); it.hasNext(); ) {
          System.out.print(it.next() + " ");
        }
        System.out.print("\nPostOrder Traversal:  ");
        for (it = tree.new PostOrder(); it.hasNext(); ) {
          System.out.print(it.next() + " ");
        }
        System.out.print("\nLevelOrder Traversal: ");
        for (it = tree.new LevelOrder(); it.hasNext(); ) {
          System.out.print(it.next() + " ");
        }
        System.out.println();
      }
    }
```

The output is:

```
tree = [A, B, D, G, E, C, F, H, I]
PreOrder Traversal:   A B D G E C F H I
InOrder Traversal:    D G B E A H F I C
PostOrder Traversal:  G D E B H I F C A
LevelOrder Traversal: A B C D E F G H I
```

Each of the four iterators traverses the tree according to the algorithm that it implements.

FORESTS

A *forest* is a sequence of disjoint ordered trees.

EXAMPLE 11.25 A Forest

Figure 11.23 shows a forest that consists of three trees.

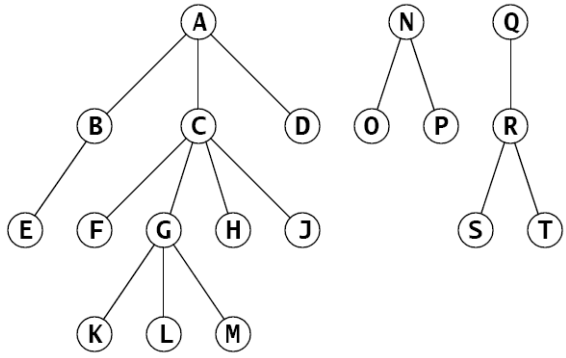

Figure 11.23 A forest

The following algorithm shows how a forest can be represented by a single binary tree.

Algorithm 11.8 The Natural Mapping of a Forest into a Binary Tree

1. Map the root of the first tree into the root of the binary tree.
2. If node X maps into X' and node Y is the first child of X, then map Y into the left child of X'.
3. If node X maps into X' and node Z is the sibling of X, then map Z into the right child of X'. The roots of the trees themselves are considered siblings.

EXAMPLE 11.26 Mapping a Forest into a Binary Tree

Figure 11.24 is the mapping of the forest shown in Example 11.25. For example, in the original forest, C has oldest child F and next sibling D. In the corresponding binary tree, C has left child F and right child D.

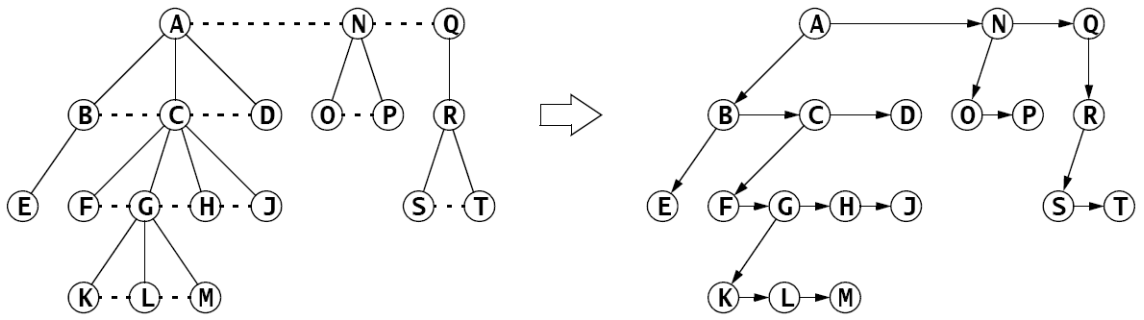

Figure 11.24 The natural mapping of a forest into a binary tree

Review Questions

11.1 How many leaf nodes does the full binary tree of height $h = 3$ have?

11.2 How many internal nodes does the full binary tree of height $h = 3$ have?

11.3 How many nodes does the full binary tree of height $h = 3$ have?

11.4 How many leaf nodes does a full binary tree of height $h = 9$ have?

11.5 How many internal nodes does a full binary tree of height $h = 9$ have?

11.6 How many nodes does a full binary tree of height $h = 9$ have?

11.7 What is the range of possible heights of a binary tree with $n = 100$ nodes?

11.8 Why is there no inorder traversal for general trees?

11.9 True or false:

- **a.** If all of its leaves are at the same level, then the binary tree is full.
- **b.** If the binary tree has n nodes and height h, then $h \geq \lfloor \lg n \rfloor$.
- **c.** A binary tree cannot have more than 2^d nodes at depth d.
- **d.** If every proper subtree of a binary tree is full, then the tree itself must also be full.

Problems

11.1 For each of the binary trees in Figure 11.25 on page 221, draw the equivalent version that satisfies the second definition, namely that every internal node has two children.

11.2 Give the order of visitation of the binary tree shown in Figure 11.26 using the specified traversal algorithm:

- **a.** the level order traversal
- **b.** the preorder traversal
- **c.** the inorder traversal
- **d.** the postorder traversal

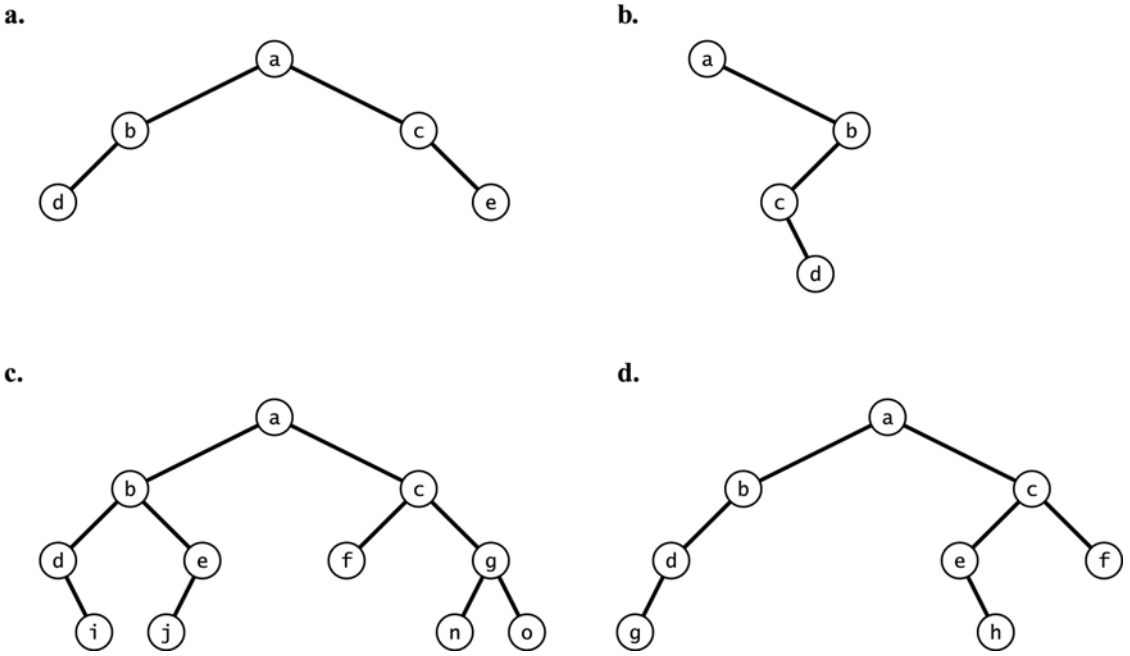

Figure 11.25 Binary trees

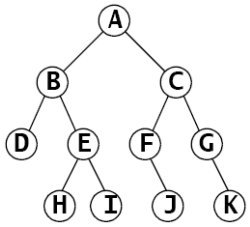

Figure 11.26 A binary tree

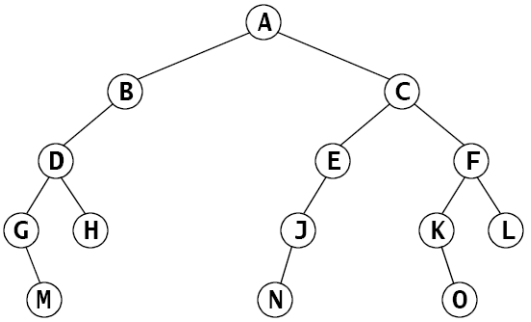

Figure 11.27 A binary tree

11.3 Give the order of visitation of the binary tree of size 10 shown in Example 11.1 on page 201 using:

- **a.** the level order traversal
- **b.** the preorder traversal
- **c.** the inorder traversal
- **d.** the postorder traversal

11.4 Give the order of visitation of the binary tree shown in Figure 11.27 using:

- **a.** the level order traversal
- **b.** the preorder traversal
- **c.** the inorder traversal
- **d.** the postorder traversal

11.5 Show the array that is obtained by using the natural mapping to store the binary tree shown in Problem 11.1.

11.6 Show the array that is obtained by using the natural mapping to store the binary tree shown in Example 11.1 on page 201.

11.7 Show the array that is obtained by using the natural mapping to store the binary tree shown in Problem 11.4.

11.8 If the nodes of a binary tree are numbered according to their natural mapping, and the visit operation prints the node's number, which traversal algorithm will print the numbers in order?

11.9 Draw the expression tree for $a*(b + c)*(d*e + f)$.

11.10 Write the prefix and the postfix representations for the expression in Problem 11.8.

11.11 Draw the expression tree for each of the prefix expressions given in Problem 5.2 on page 111.

11.12 Draw the expression tree for each of the infix expressions given in Problem 5.4 on page 111.

11.13 Draw the expression tree for each of the postfix expressions given in Problem 5.6 on page 111.

11.14 Draw the expression tree for the expression $a*(b + c)*(d*e + f)$.

11.15 What are the bounds on the number n of nodes in a binary tree of height 4?

11.16 What are the bounds on the height h of a binary tree with 7 nodes?

11.17 What form does the highest binary tree have for a given number of nodes?

11.18 What form does the lowest binary tree (i.e., the least height) have for a given number of nodes?

11.19 Verify the recursive definition of binary trees (page 200) for the binary tree shown in Figure 11.28.

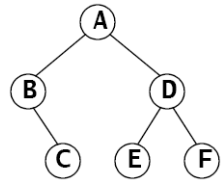

Figure 11.28 A binary tree

11.20 Draw all 42 binary trees of size $n = 5$.

11.21 How many different binary trees of size $n = 6$ are there?

11.22 Derive a recurrence relation for the number $f(n)$ of binary trees of size n.

11.23 Show that, for all $n \leq 8$, the function $f(n)$ derived in Problem 11.22 produces the same sequence as the following explicit formula

$$f(n) = \frac{\binom{2n}{n}}{n+1} = \frac{(2n)!}{n!(n+1)!}\frac{(2n)(2n-1)(2n-2)\cdots(2n+3)(2n+2)}{(n)(n-1)(n-2)(n-3)\cdots(2)(1)}$$

For example,

$$f(4) = \frac{\binom{8}{4}}{5} = \frac{8!}{4!5!} = \frac{(8)(7)(6)}{(4)(3)(2)(1)} = \frac{(8)(7)}{4} = 14$$

11.24 Prove Corollary 11.3 on page 203.

11.25 Prove Theorem 11.2 on page 205.

11.26 Draw the forest that is represented by the binary tree shown in Figure 11.29.

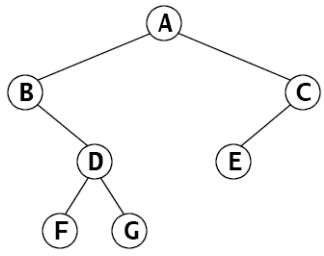

Figure 11.29 A binary tree

11.27 Derive an explicit formula for the number $f(h)$ of complete binary trees of height h.

11.28 Derive an explicit formula for the number $f(h)$ of full binary trees of height h.

11.29 Implement the each of the following methods for the `BinaryTree` class:

a.
```
public int leaves();
// returns the number of leaves in this tree
```

b.
```
public int height();
// returns the height of this tree
```

c.
```
public int level(Object object);
// returns -1 if the given object is not in this tree;
// otherwise, returns its level in this tree;
```

d.
```
public void reflect();
// swaps the children of each node in this tree
```

e.
```
public void defoliate();
// removes all the leaves from this tree
```

Answers to Review Questions

11.1 The full binary tree of height 3 has $l = 2^3 = 8$ leaves.

11.2 The full binary tree of height 3 has $m = 2^3 - 1 = 7$ internal nodes.

11.3 The full binary tree of height 3 has $n = 2^{3+1} - 1 = 2^4 - 1 = 16 - 1 = 15$ nodes.

11.4 The full binary tree of height 9 has $l = 2^9 = 512$ leaves.

11.5 The full binary tree of height 9 has $m = 2^9 - 1 = 512 - 1 = 511$ internal nodes.

11.6 The full binary tree of height 9 has $n = 2^{9+1} - 1 = 2^{10} - 1 = 1024 - 1 = 1023$ nodes.

11.7 By Corollary 11.3, in any binary tree: $\lfloor \lg n \rfloor \le h \le n-1$. Thus in a binary tree with 100 nodes $\lfloor \lg 100 \rfloor \le h \le 100-1 = 99$. Since $\lfloor \lg 100 \rfloor = \lfloor (\log 100)/(\log 2) \rfloor = \lfloor 6.6 \rfloor = 6$, it follows that the height must be between 6 and 99, inclusive: $6 \le h \le 99$.

11.8 The inorder traversal algorithm for binary trees recursively visits the root in between traversing the left and right subtrees. This presumes the existence of exactly two (possibly empty) subtrees at every (nonempty) node. In general trees, a node may have any number of subtrees, so there is no simple algorithmic way to generalize the inorder traversal.

11.9
a. True
b. True
c. True
d. False

Solutions to Problems

11.1 The equivalent trees are shown in Figure 11.30.

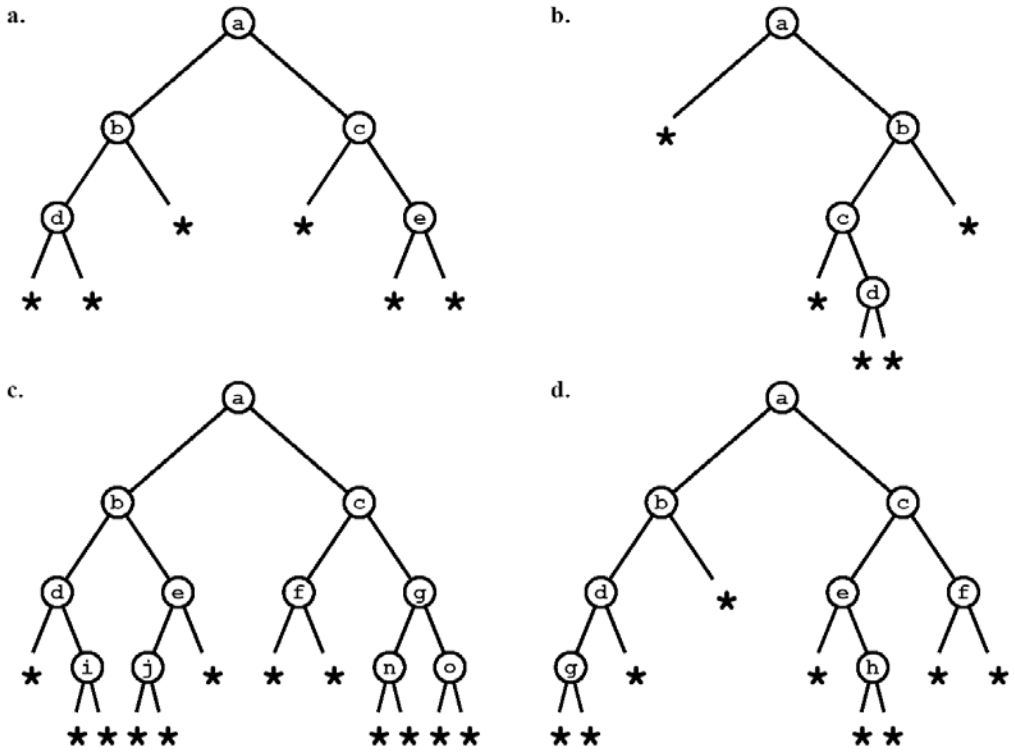

Figure 11.30 Binary trees

11.2 The order of visitation in the binary tree traversal:

- **a.** Level order: **A**, **B**, **C**, **D**, **E**, **F**, **G**, **H**, **I**, **J**, **K**
- **b.** Preorder: **A**, **B**, **D**, **E**, **H**, **I**, **C**, **F**, **J**, **G**, **K**
- **c.** Inorder: **D**, **B**, **H**, **E**, **I**, **A**, **F**, **J**, **C**, **G**, **K**
- **d.** Postorder: **D**, **H**, **I**, **E**, **B**, **J**, **F**, **K**, **G**, **C**, **A**

11.3 The order of visitation in the binary tree traversal:

- **a.** Level order traversal: **A**, **B**, **C**, **D**, **E**, **F**, **H**, **I**, **J**, **M**
- **b.** Preorder traversal: **A**, **B**, **D**, **H**, **I**, **E**, **J**, **C**, **F**, **M**
- **c.** Inorder traversal: **H**, **D**, **I**, **B**, **J**, **E**, **A**, **F**, **M**, **C**
- **d.** Postorder traversal: **H**, **I**, **D**, **J**, **E**, **B**, **M**, **F**, **C**, **A**

11.4 The order of visitation in the binary tree traversal:

- **a.** Level order traversal: **A**, **B**, **C**, **D**, **E**, **F**, **G**, **H**, **J**, **K**, **L**, **M**, **N**, **O**
- **b.** Preorder traversal: **A**, **B**, **D**, **G**, **M**, **H**, **C**, **E**, **J**, **N**, **F**, **K**, **O**, **L**
- **c.** Inorder traversal: **G**, **M**, **D**, **H**, **B**, **A**, **N**, **J**, **E**, **C**, **K**, **O**, **F**, **L**
- **d.** Postorder traversal: **M**, **G**, **H**, **D**, **B**, **N**, **J**, **E**, **O**, **K**, **L**, **F**, **C**, **A**

11.5 The natural mapping of the specified binary tree is shown in Figure 11.31.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A	B	C	D	E	F	G			H	I		J		K

Figure 11.31 An array

11.6 The natural mapping of the specified binary tree is shown in Figure 11.32.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	B	C	D	E	F		H	I	J			M

Figure 11.32 An array

11.7 The natural mapping of the specified binary tree is shown in Figure 11.33.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	A	B	C	D		E	F	G	H			J		K	L		M							N					O

Figure 11.33 An array

11.8 The level order traversal will print the numbers from the natural mapping in order.

11.9 The expression tree for $a*(b+c)*(d*e+f)$ is shown in Figure 11.34.

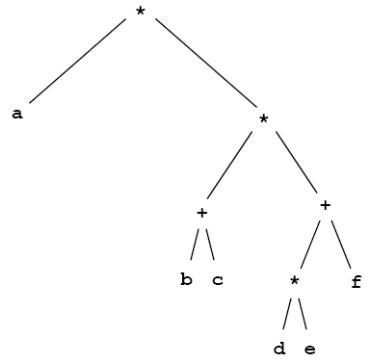

Figure 11.34 A binary tree

11.10 The prefix expression is $*a*+bc+*def$. The postfix expression is $*abc+de*f+**$.

11.11 Figure 11.35 shows the expression tree for each of the prefix expressions given in Problem 5.2 on page 111.

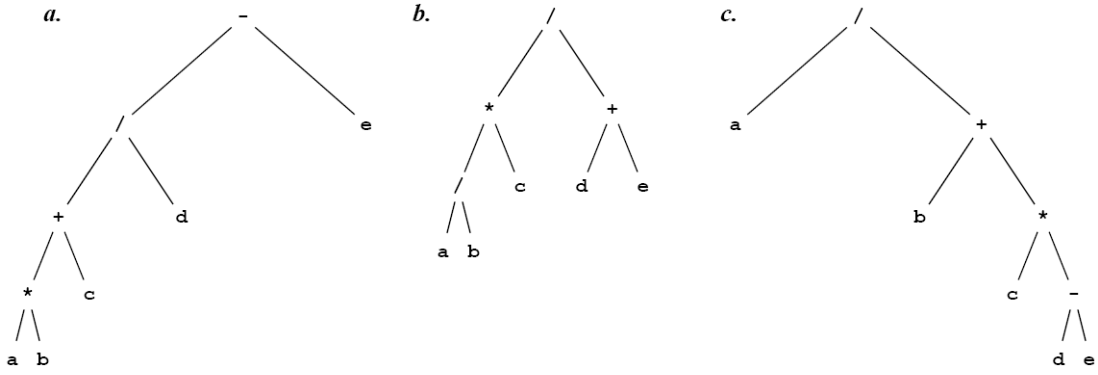

Figure 11.35 Prefix expression trees

11.12 Figure 11.36 shows the expression tree for each of the infix expressions given in Problem 5.4 on page 111.

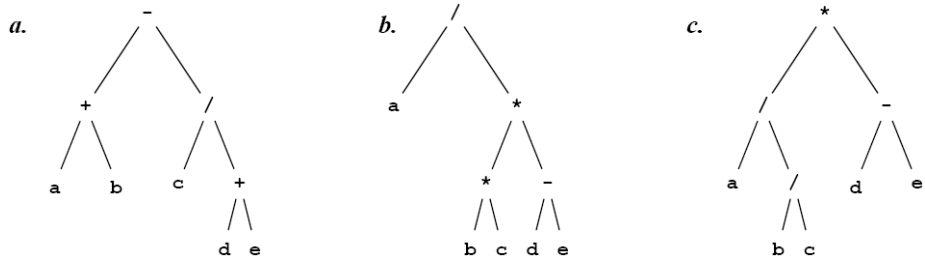

Figure 11.36 Infix expression trees

11.13 Figure 11.37 shows the expression tree for each of the postfix expressions given in Problem 5.6 on page 111.

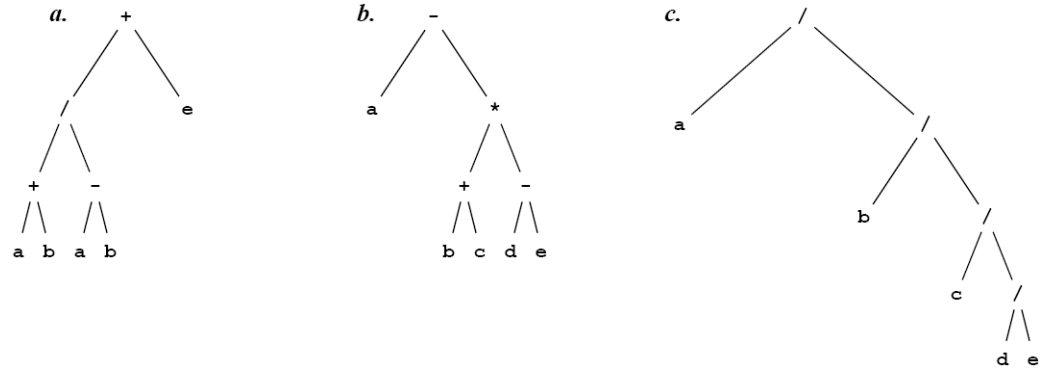

Figure 11.37 Postfix expression trees

11.14 Figure 11.38 shows the expression tree for $a*(b + c)*(d*e + f)$ is:

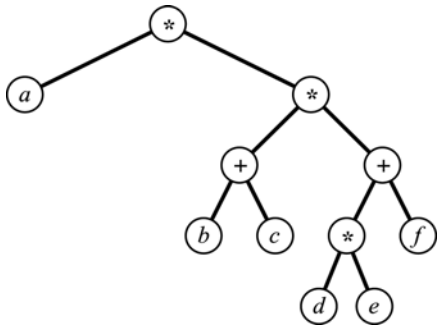

Figure 11.38 An expression tree

11.15 In a binary tree of height $h = 4$, $5 \le n \le 31$.

11.16 In a binary tree with $n = 7$ nodes, $2 \le h \le 6$.

11.17 For a given number of nodes, the highest binary tree is a linear sequence.

11.18 For a given number of nodes, the lowest binary tree is a complete binary tree.

11.19 To verify the recursive definition for the given tree, we first note that the leaves **C**, **E**, and **F** are binary trees because every singleton satisfies the recursive definition for binary trees because its left and right subtrees are both empty (and therefore binary trees). Next, it follows that the subtree rooted at **B** is a binary tree because it is a triplet (X,L,R) where $X =$ **B**, $L = \varnothing$, and $R =$ **C**. Similarly, it follows that the subtree rooted at **D** is a binary tree because it is a triplet (X,L,R) where $X =$ **D**, $L =$ **E**, and $R =$ **F**. Finally, it follows that the entire tree satisfies the recursive definition because it is a triplet (X,L,R) where $X =$ **A**, L is the binary tree rooted at **B**, and L is the binary tree rooted at **D**.

11.20 Figure 11.39 on page 227 shows all 42 different binary trees of size $n = 5$.

11.21 There are 132 different binary trees of size 6: $1\cdot42 + 1\cdot14 + 2\cdot5 + 5\cdot2 + 14\cdot1 + 42\cdot1 = 132$.

11.22 A nonempty binary tree consists of a root X, a left subtree L, and a right subtree R. Let n be the size of the binary tree, let $n_L = |L|$ = the size of L, and $n_R = |R|$ = the size of R. Then $n = 1 + n_L + n_R$. So there are only n different possible values for the pair (n_L, n_R): (0, n–1), (1, n–2), ..., (n–1,0). For example, if $n = 6$ (as in Problem 11.21), the only possibilities are (0,5), (1,4), (2,3), (3,2), (4,1), or (5,0). In the (0, n–1) case, L is empty and $|R| = n$–1; there are $f(0)\cdot f(n–1)$ different binary trees in that case. In the (1, n–2) case, L is a singleton and $|R| = n$–2; there are $f(1)\cdot f(n–2)$ different binary trees in that case. The same principle applies to each case. Therefore the total number of different binary trees of size n is

$$f(n) = 1\cdot f(n–1) + 1\cdot f(n–2) + 2\cdot f(n–3) + 5\cdot f(n–4) + 14\cdot f(n–5) + \cdots + f(i–1)\cdot f(n–i) + \cdots + f(n–1)\cdot 1$$

In closed form, the formula is

$$f(n) = \sum_{i=1}^{n} f(i-1)\cdot f(n-i)$$

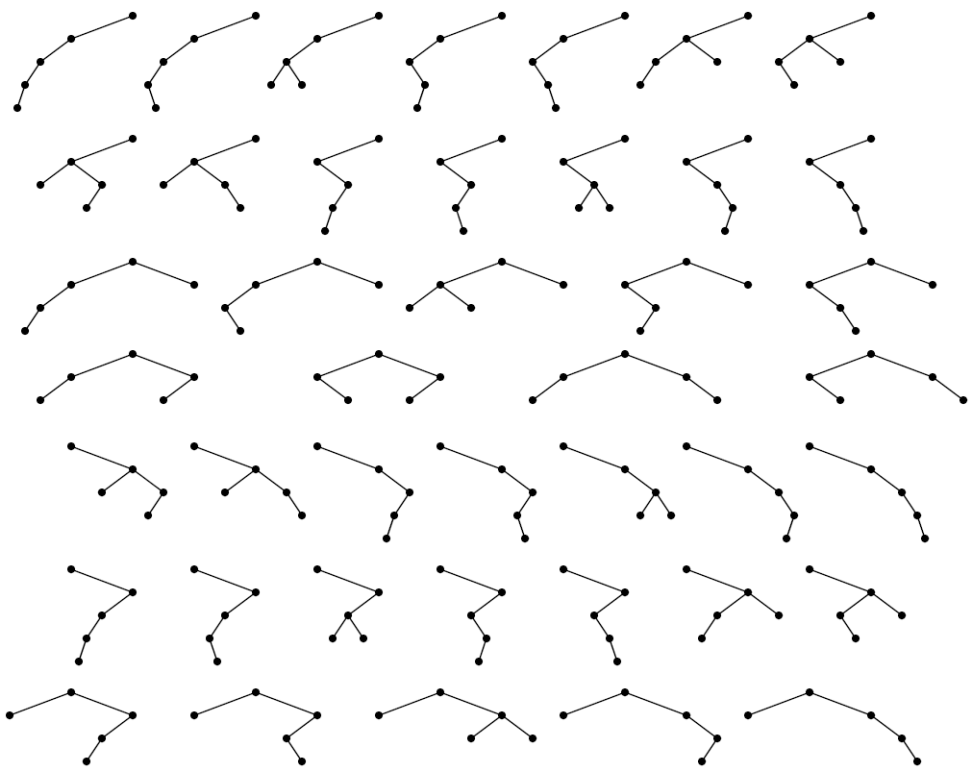

Figure 11.39 The 42 binary trees of size 5

11.23 These are called the *Catalan numbers*:

n	$\binom{2n}{n}$	$n+1$	$\frac{\binom{2n}{n}}{(n+1)}$	$\sum f(i-1)\cdot f(n-i)$
0	1	1	1	1
1	2	2	1	$1\cdot1 = 1$
2	6	3	2	$1\cdot1+1\cdot1 = 2$
3	20	4	5	$1\cdot2+1\cdot1+2\cdot1 = 5$
4	70	5	14	$1\cdot5+1\cdot2+2\cdot1+5\cdot1 = 14$
5	252	6	42	$1\cdot14+1\cdot5+2\cdot2+5\cdot1+14\cdot1 = 42$
6	924	7	132	$1\cdot42+1\cdot14+2\cdot5+5\cdot2+14\cdot1+42\cdot1 = 132$
7	3432	8	429	$1\cdot132+1\cdot42+2\cdot14+5\cdot5+14\cdot2+42\cdot1+132\cdot1 = 429$

Table 11.1 Catalan numbers

11.24 For a given height $h > 0$, the binary tree with the most nodes is the full binary tree. Corollary 11.1 on page 202 states that that number is $n = 2^{h+1} - 1$. Therefore, in any binary tree of height h, the number n of nodes must satisfy $n \le 2^{h+1} - 1$. The binary tree with the fewest nodes for a given height h is the one in which every internal node has only one child; that linear tree has $n = h + 1$ nodes because every node except the single leaf has exactly one child. Therefore, in any binary tree of height h, the number n of nodes must satisfy $n \ge h + 1$. The second pair of inequalities follows from the first by solving for h.

11.25 Let T be any binary tree of height h and size n. Let T_1 be the smallest complete binary tree that contains T. Let h_1 be the height of T_1 and let n_1 be its size. Then $h = h_1$ and $n \le n_1$. Then by Corollary 11.1 on page 202, $n \le n_1 = 2^{h+1} - 1$. The required inequalities follow from this result.

11.26 Figure 11.40 shows how the forest that produced the specified binary tree was obtained by reversing the natural map.

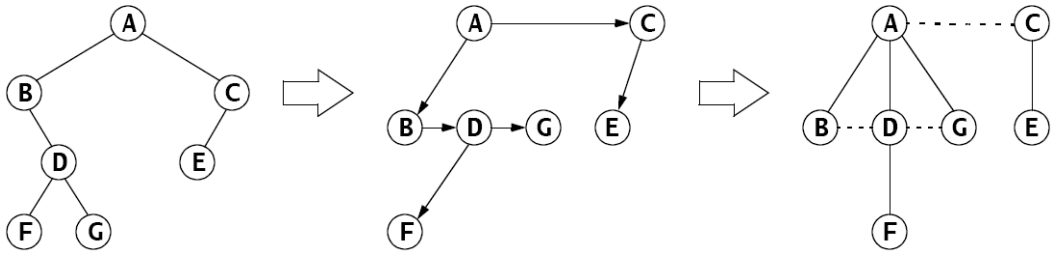

Figure 11.40 Mapping a forest into a binary tree

11.27 $f(h) = h + 1$

11.28 $f(h) = 1$

11.29 **a.**

```
public int leaves() {
  if (this == null) {
    return 0;
  }
  int leftLeaves = (left==null ? 0 : left.leaves());
  int rightLeaves = (right==null ? 0 : right.leaves());
  return leftLeaves + rightLeaves;
}
```

b.

```
public int height() {
  if (this == null) {
    return -1;
  }
  int leftHeight = (left==null ? -1 : left.height());
  int rightHeight = (right==null ? -1 : right.height());
  return 1 + (leftHeight<rightHeight ? rightHeight : leftHeight);
}
```

c.

```
public int level(Object object) {
  if (this == null) {
    return -1;
  } else if (object == root) {
    return 0;
  }
  int leftLevel = (left==null ? -1 : left.level(object));
  int rightLevel = (right==null ? -1 : right.level(object));
  if (leftLevel < 0 && rightLevel < 0) {
    return -1;
  }
  return 1 + (leftLevel<rightLevel ? rightLevel : leftLevel);
}
```

d.

```
public void reflect() {
  if (this == null) {
    return;
  }
  if (left != null) {
    left.reflect();
  }
  if (right != null) {
    right.reflect();
  }
  BinaryTree temp=left;
  left = right;
  right = temp;
}
```

e.

```
public void defoliate() {
  if (this == null) {
    return;
  } else if (left == null && right == null) {
    root = null;
    return;
  }
  if (left != null && left.left==null && left.right==null) {
    left = null;
  } else {
    left.defoliate();
  }
  if (right != null && right.left==null && right.right==null)
    right = null;
  } else {
    right.defoliate();
  }
}
```