9 Trees

- Introduction to Trees
- Applications of Trees
- Tree Traversal
- Spanning Trees
- Minimum Spanning Trees

A connected graph that contains no simple circuits is called a tree. Trees were used as long ago as 1857, when the English mathematician Arthur Cayley used them to count certain types of chemical compounds. Since that time, trees have been employed to solve problems in a wide variety of disciplines, as the examples in this chapter will show.

Trees are particularly useful in computer science, where they are employed in a wide range of algorithms. For instance, trees are used to construct efficient algorithms for locating items in a list. They can be used in algorithms, such as Huffman coding, that construct efficient codes saving costs in data transmission and storage. Trees can be used to study games such as checkers and chess and can help determine winning strategies for playing these games. Trees can be used to model procedures carried out using a sequence of decisions. Constructing these models can help determine the computational complexity of algorithms based on a sequence of decisions, such as sorting algorithms.

Procedures for building trees containing every vertex of a graph, including depth-first search and breadth-first search, can be used to systematically explore the vertices of a graph. Exploring the vertices of a graph via depth-first search, also known as backtracking, allows for the systematic search for solutions to a wide variety of problems, such as determining how eight queens can be placed on a chessboard so that no queen can attack another.

We can assign weights to the edges of a tree to model many problems. For example, using weighted trees we can develop algorithms to construct networks containing the least expensive set of telephone lines linking different network nodes.

9.1 INTRODUCTION TO TREES

Links In Chapter 8 we showed how graphs can be used to model and solve many problems. In this chapter we will focus on a particular type of graph called a **tree**, so named because such graphs resemble trees. For example, *family trees* are graphs that represent genealogical charts. Family trees use vertices to represent the members of a family and edges to represent parent–child relationships. The family tree of the male members of the Bernoulli family of Swiss mathematicians is shown in Figure 1. The undirected graph representing a family tree (restricted to people of just one gender and with no inbreeding) is an example of a tree.

Definition 1 A *tree* is a connected undirected graph with no simple circuits.

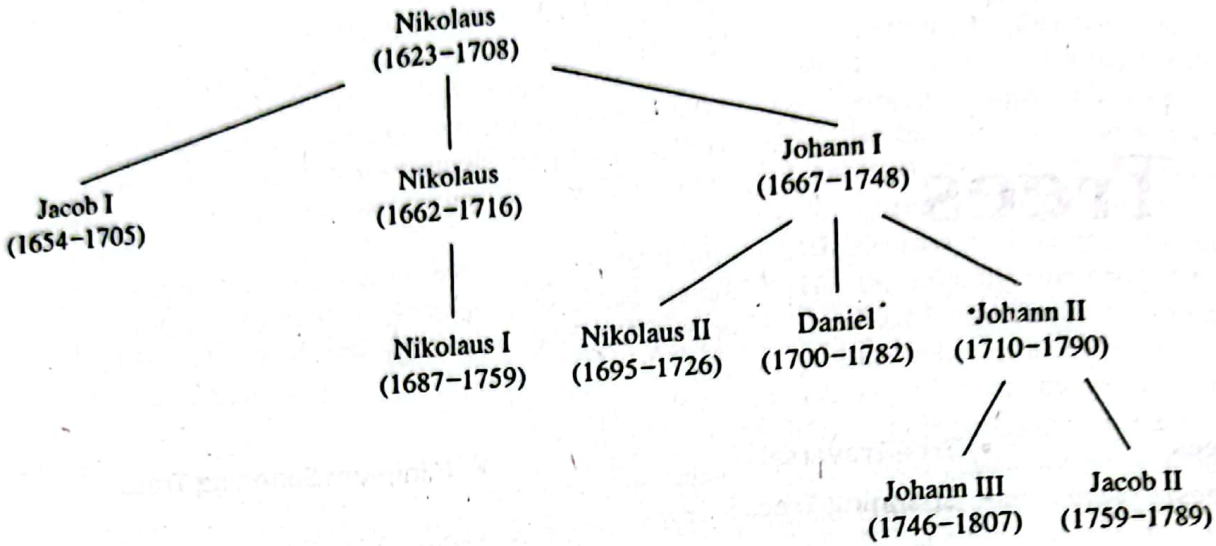


Figure 1 *The Bernoulli Family of Mathematicians.*

Because a tree cannot have a simple circuit, a tree cannot contain multiple edges or loops. Therefore any tree must be a simple graph.

Example 1 *Which of the graphs shown in Figure 2 are trees ?*

Solution G_1 and G_2 are trees, because both are connected graphs with no simple circuits. G_3 is not a tree because e, b, a, d, e is a simple circuit in this graph. Finally, G_4 is not a tree because it is not connected. ◀

Any connected graph that contains no simple circuits is a tree. What about graphs containing no simple circuits that are not necessarily connected? These graphs are called **forests** and have the property that each of their connected components is a tree. Figure 3 displays a forest.

Trees are often defined as undirected graphs with the property that there is a unique simple path between every pair of vertices. Theorem1 shows that this alternative definition is equivalent to our definition.

Theorem 1 An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

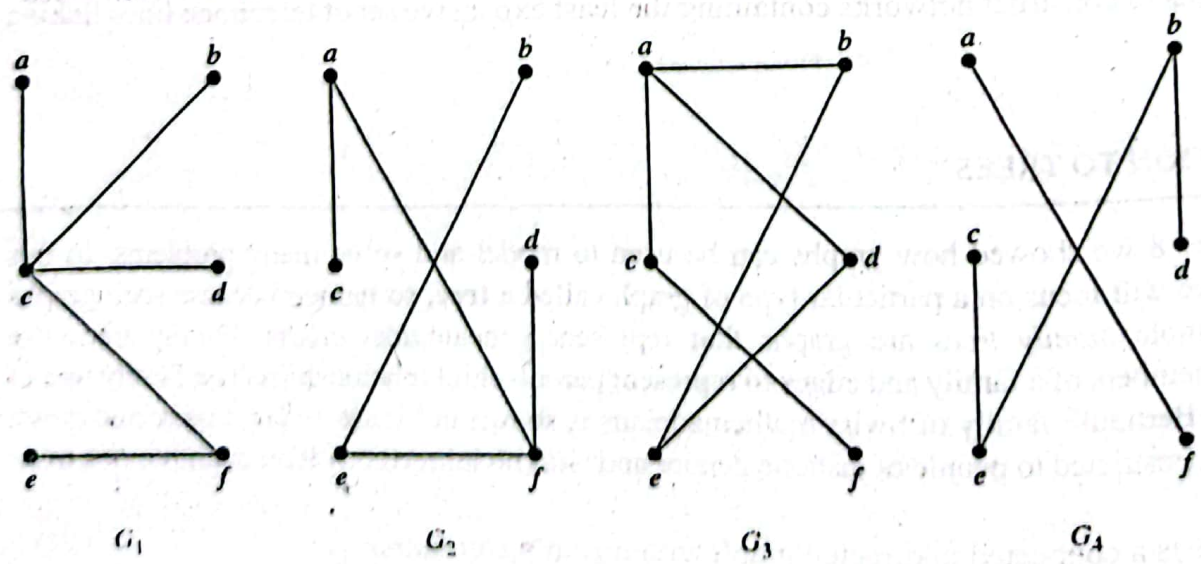


Figure 2 *Examples of Trees and Graphs That Are Not Trees.*

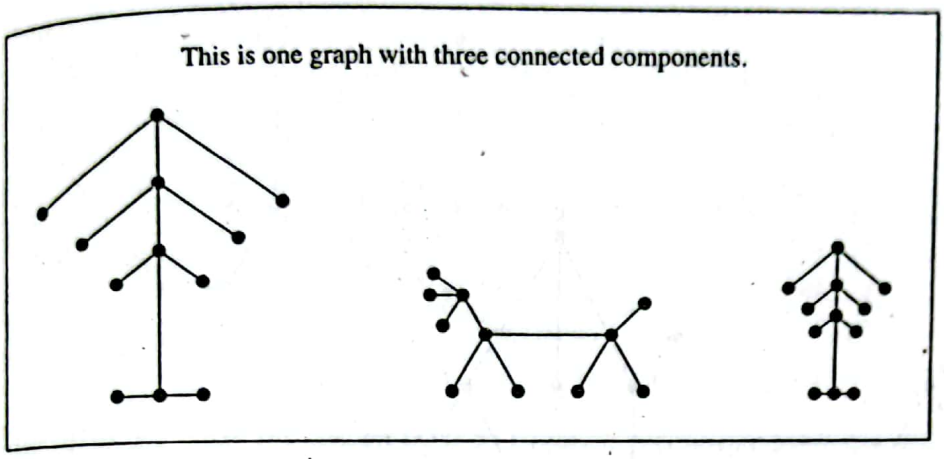


Figure 3 *Example of a Forest.*

Proof: First assume that T is a tree. Then T is a connected graph with no simple circuits. Let x and y be two vertices of T. Because T is connected, by Theorem 1 of Section 8.4 there is a simple path between x and y. Moreover, this path must be unique, for if there were a second such path, the path formed by combining the first path from x to y followed by the path from y to x obtained by reversing the order of the second path from x to y would form a circuit. This implies, using Exercise 49 of Section 8.4, that there is a simple circuit in T. Hence, there is a unique simple path between any two vertices of a tree.

Now assume that there is a unique simple path between any two vertices of a graph T. Then T is connected, because there is a path between any two of its vertices. Furthermore, T can have no simple circuits. To see that this is true, suppose T had a simple circuit that contained the vertices x and y. Then there would be two simple paths between x and y, because the simple circuit is made up of a simple path from x to y and a second simple path from y to x. Hence, a graph with a unique simple path between any two vertices is a tree. ◀

In many applications of trees, a particular vertex of a tree is designated as the **root.** Once we specify a root, we can assign a direction to each edge as follows. Because there is a unique path from the root to each vertex of the graph (by Theorem1), we direct each edge away from the root. Thus, a tree together with its root produces a directed graph called a **rooted tree.**

Definition 2 A *rooted tree* is a tree in which one vertex has been designated as the root and every edge is directed away from the root.

Rooted trees can also be defined recursively. Refer to Section 4.3 to see how this can be done. We can change an unrooted tree into a rooted tree by choosing any vertex as the root. Note that different choices of the root produce different rooted trees. For instance, Figure 4 displays the rooted trees formed by designating a to be the root and c to be the root, respectively, in the tree T. We usually draw a rooted tree with its root at

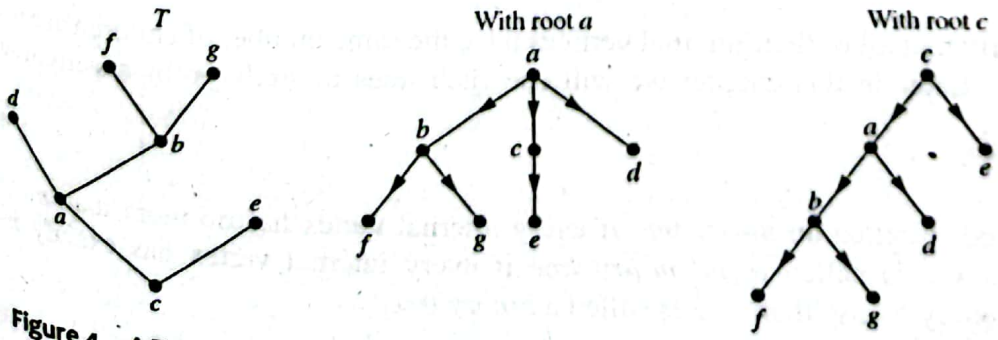


Figure 4 *A Tree and Rooted Trees Formed by Designating Two Roots.*

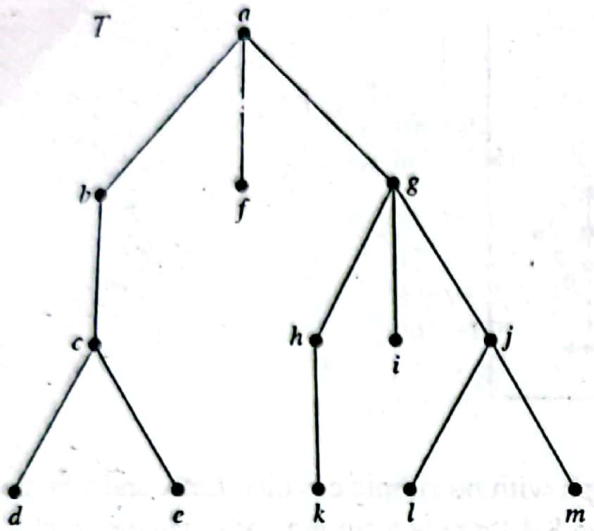


Figure 5 A Rooted Tree T.

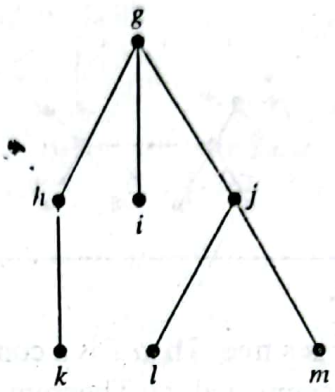


Figure 6 The Subtree Rooted at g.

the top of the graph. The arrows indicating the directions of the edges in a rooted tree can be omitted, because the choice of root determines the directions of the edges.

The terminology for trees has botanical and genealogical origins. Suppose that T is a rooted tree. If is a vertex in T other than the root, the **parent** of v is the unique vertex u such that there is a directed edge from u to v (the reader should show that such a vertex is unique). When u is the parent of v, v is called a **child** of u. Vertices with the same parent are called **siblings**. The **ancestors** of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root (that is, its parent, its parent's parent, and so on, until the root is reached). The **descendants** of a vertex v are those vertices that have v as an ancestor. A vertex of a tree is called a **leaf** if it has no children. Vertices that have children are called **internal vertices**. The root is an internal vertex unless it is the only vertex in the graph, in which case it is a leaf.

If a is a vertex in a tree, the **subtree** with a as its root is the subgraph of the tree consisting of a and its descendants and all edges incident to these descendants.

Example 2 In the rooted tree T (with root a) shown in Figure 5, find the parent of c, the children of g, the siblings of h, all ancestors of e, all descendants of b, all internal vertices, and all leaves. What is the subtree rooted at g?

Extra Examples

Solution The parent of c is b. The children of g are h, i, and j. The siblings of h are i and j. The ancestors of e are c, b, and a. The descendants of b are c, d, and e. The internal vertices are a, b, c, g, h, and j. The leaves are d, e, f, i, k, l, and m. The subtree rooted at g is shown in Figure 6. ◀

Rooted trees with the property that all of their internal vertices have the same number of children are used in many different applications. Later in this chapter we will use such trees to study problems involving searching, sorting, and coding.

Definition 3 A rooted tree is called an *m-ary* tree if every internal vertex has no more than m children. The tree is called a *full m-ary tree* if every internal vertex has exactly m children. An m-ary tree with $m = 2$ is called a *binary tree*.

Links

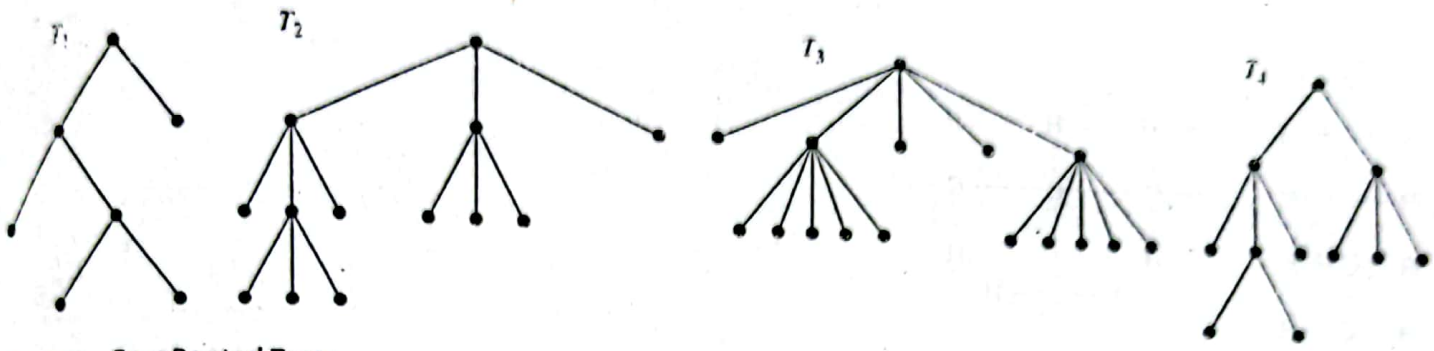


Figure 7 Four Rooted Trees.

Example 3 *Are the rooted trees in Figure 7 full m-ary trees for some positive integer m?*

Solution T_1 is a full binary tree because each of its internal vertices has two children. T_2 is a full 3-ary tree because each of its internal vertices has three children. In T_3 each internal vertex has five children, so T_3 is a full 5-ary tree. T_4 is not a full m-ary tree for any m because some of its internal vertices have two children and others have three children. ◀

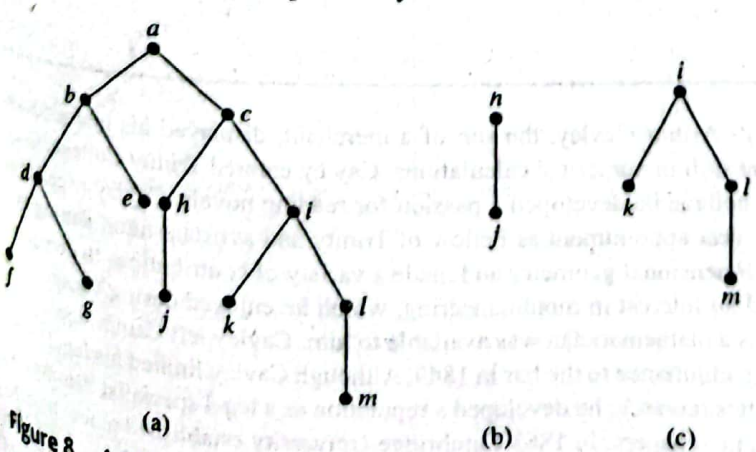
An **ordered rooted tree** is a rooted tree where the children of each internal vertex are ordered. Ordered rooted trees are drawn so that the children of each internal vertex are shown in order from left to right. Note that a representation of a rooted tree in the conventional way determines an ordering for its edges. We will use such orderings of edges in drawings without explicitly mentioning that we are considering a rooted tree to be ordered.

In an ordered binary tree (usually called just a **binary tree**), if an internal vertex has two children, the first child is called the **left child** and the second child is called the **right child.** The tree rooted at the left child of a vertex is called the **left subtree** of this vertex, and the tree rooted at the right child of a vertex is called the **right subtree** of the vertex. The reader should note that for some applications every vertex of a binary tree, other than the root, is designated as a right or a left child of its parent. This is done even when some vertices have only one child. We will make such designations whenever it is necessary, but not otherwise.

Ordered rooted trees can be defined recursively. Binary trees, a type of ordered rooted trees, were defined this way in Section 4.3.

Example 4 *What are the left and right children of d in the binary tree T shown in Figure 8(a) (where the order is that implied by the drawing)? What are the left and right subtrees of c?*

Solution The left child of d is f and the right child is g. We show the left and right subtrees of c in Figures 8(b) and 8(c), respectively. ◀



(a) (b) (c)

Figure 8 A Binary Tree T and Left and Right Subtrees of the Vertex c.

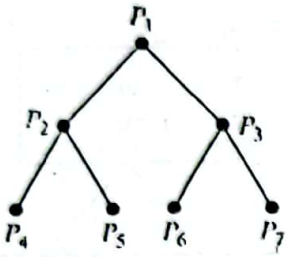


Figure 12 A Tree-Connected Network of Seven Processors.

interconnect processors. The graph representing such a network is a complete binary tree. Such a network interconnects $n = 2^k - 1$ processors, where k is a positive integer. A processor represented by the vertex that is not a root or a leaf has three two-way connections—one to the processor represented by the parent of and two to the processors represented by the two children of. The processor represented by the root has two two-way connections to the processors represented by its two children. A processor represented by a leaf has a single two-way connection to the parent of. We display a tree-connected network with seven processors in Figure 12.

We will illustrate how a tree-connected network can be used for parallel computation. In particular, we will show how the processors in Figure 12 can be used to add eight numbers, using three steps. In the first step, we add x_1 and x_2 using P_4, x_3 and x_4 using P_5, x_5 and x_6 using P_6, and x_7 and x_8 using P_7. In the second step, we add $x_1 + x_2$ and $x_3 + x_4$ using P_2 and $x_5 + x_6$ and $x_7 + x_8$ using P_3. Finally, in the third step, we add $x_1 + x_2 + x_3 + x_4$ and $x_5 + x_6 + x_7 + x_8$ using P_1. The three steps used to add eight numbers compares favorably to the seven steps required to add eight numbers serially, where the steps are the addition of one number to the sum of the previous numbers in the list. ◀

Properties of Trees We will often need results relating the numbers of edges and vertices of various types in trees.

Theorem 2 A tree with n vertices has $n - 1$ edges.

Proof: We will use mathematical induction to prove this theorem. Note that for all the trees here we can choose a root and consider the tree rooted.

Links

Basis step: When $n = 1$, a tree with $n = 1$ vertex has no edges. It follows that the theorem is true for $n = 1$.

Inductive step: The induction hypothesis states that every tree with k vertices has $k - 1$ edges, where k is a positive integer. Suppose that a tree T has $k + 1$ vertices and that is a leaf of T (which must exist because the tree is finite), and let w be the parent of. Removing from T the vertex and the edge connecting w to produces a tree T' with k vertices, because the resulting graph is still connected and has no simple circuits. By the induction hypothesis, T' has $k - 1$ edges. It follows that T has k edges because it has one more edge than T, the edge connecting and w. This completes the induction step. ◀

The number of vertices in a full m-ary tree with a specified number of internal vertices is determined, as Theorem 3 shows. As in Theorem 2, we will use n to denote the number of vertices in a tree.

Theorem 3 A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices.

Proof: Every vertex, except the root, is the child of an internal vertex. Because each of the i internal vertices has m children, there are mi vertices in the tree other than the root. Therefore, the tree contains $n = mi + 1$ vertices. ◀

Suppose that T is a full m-ary tree. Let i be the number of internal vertices and l the number of leaves in this tree. Once one of n, i, and l is known, the other two quantities are determined. Theorem 4 explains how to find the other two quantities from the one that is known.

Theorem 4 A full m-ary tree with
(*i*) n vertices has $i = (n - 1)/m$ internal vertices and $l = [(m - 1)n + 1]/m$ leaves,
(ii) i internal vertices has $n = mi + 1$ vertices and $l = (m - 1)i + 1$ leaves,
(iii) l leaves has $n = (ml - 1)/(m - 1)$ vertices and $i = (l - 1)/(m - 1)$ internal vertices.

Proof: Let n represent the number of vertices, i the number of internal vertices, and l the number of leaves. The three parts of the theorem can all be proved using the equality given in Theorem 3, that is, $n = mi + 1$, together with the equality $n = l + i$, which is true because each vertex is either a leaf or an internal vertex. We will prove part (*i*) here. The proofs of parts (*ii*) and (*iii*) are left as exercises for the reader.

Solving for i in $n = mi + 1$ gives $i = (n - 1)/m$. Then inserting this expression for i into the equation $n = l + i$ shows that $l = n - i = n - (n - 1)/m = [(m - 1)n + 1]/m$. ◀

Example 9 illustrates how Theorem 4 can be used.

Example 9 *Suppose that someone starts a chain letter. Each person who receives the letter is asked to send it on to four other people. Some people do this, but others do not send any letters. How many people have seen the letter, including the first person, if no one receives more than one letter and if the chain letter ends after there have been 100 people who read it but did not send it out? How many people sent out the letter?*

Solution The chain letter can be represented using a 4-ary tree. The internal vertices correspond to people who sent out the letter, and the leaves correspond to people who did not send it out. Because 100 people did not send out the letter, the number of leaves in this rooted tree is $l = 100$. Hence, part (*iii*) of Theorem 4 shows that the number of people who have seen the letter is $n = (4 \cdot 100 - 1)/(4 - 1) = 133$. Also, the number of internal vertices is $133 - 100 = 33$, so 33 people sent out the letter. ◀

It is often desirable to use rooted trees that are "balanced" so that the subtrees at each vertex contain paths of approximately the same length. Some definitions will make this concept clear. The **level** of a vertex in a rooted tree is the length of the unique path from the root to this vertex. The level of the root is defined to be zero. The **height** of a rooted tree is the maximum of the levels of vertices. In other words, the height of a rooted tree is the length of the longest path from the root to any vertex.

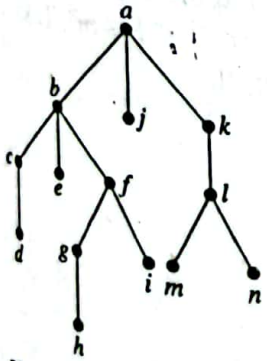


Figure 13 A Rooted Tree.

Example 10 *Find the level of each vertex in the rooted tree shown in Figure 13. What is the height of this tree?*

Solution The root a is at level 0. Vertices b, j, and k are at level 1. Vertices c, e, f, and l are at level 2. Vertices d, g, i, m, and n are at level 3. Finally, vertex h is at level 4. Because the largest level of any vertex is 4, this tree has height 4. ◀

A rooted m-ary tree of height h is **balanced if** all leaves are at levels h or $h - 1$.

Example 11 *Which of the rooted trees shown in Figure 14 are balanced?*

Solution T_1 is balanced, because all its leaves are at levels 3 and 4. However, T_2 is not balanced, because it has leaves at levels 2, 3, and 4. Finally, T_3 is balanced, because all its leaves are at level 3. ◀

The results in Theorem 5 relate the height and the number of leaves in m-ary trees.

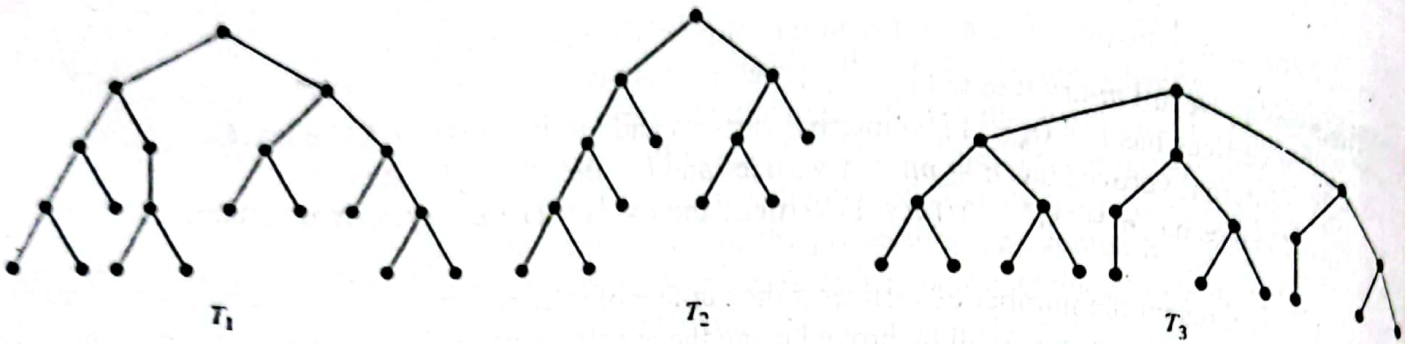


Figure 14 Some Rooted Trees.

Theorem 5 There are at most m^h leaves in an m-ary tree of height h.

Proof: The proof uses mathematical induction on the height. First, consider m-ary trees of height 1. These trees consist of a root with no more than m children, each of which is a leaf. Hence, there are no more than $m^1 = m$ leaves in an m-ary tree of height 1. This is the basis step of the inductive argument.

Now assume that the result is true for all m-ary trees of height less than h; this is the inductive hypothesis. Let T be an m-ary tree of height h. The leaves of T are the leaves of the subtrees of T obtained by deleting the edges from the root to each of the vertices at level 1, as shown in Figure 15.

Each of these subtrees has height less than or equal to $h - 1$. So by the inductive hypothesis, each of these rooted trees has at most m^{h-1} leaves. Because there are at most m such subtrees, each with a maximum of m^{h-1} leaves, there are at most $m \cdot m^{h-1} = m^h$ leaves in the rooted tree. This finishes the inductive argument. ◀

Corollary 1 If an m-ary tree of height h has l leaves, then $h \geq \lceil \log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil \log_m l \rceil$. (We are using the ceiling function here. Recall that $\lceil x \rceil$ is the smallest integer greater than or equal to x.)

Proof: We know that $l \leq m^h$ from Theorem 5. Taking logarithms to the base m shows that $\log_m l \leq h$. Because h is an integer, we have $h \geq \lceil \log_m l \rceil$. Now suppose that the tree is balanced. Then each leaf is at level h or $h - 1$, and because the height is h, there is at least one leaf at level h. It follows that there must be more than m^{h-1} leaves (see Exercise 30 at the end of this section). Because $l \leq m^h$, we have $m^{h-1} < l \leq m^h$. Taking logarithms to the base m in this inequality gives $h - 1 < \log_m l \leq h$. Hence, $h = \lceil \log_m l \rceil$. ◀

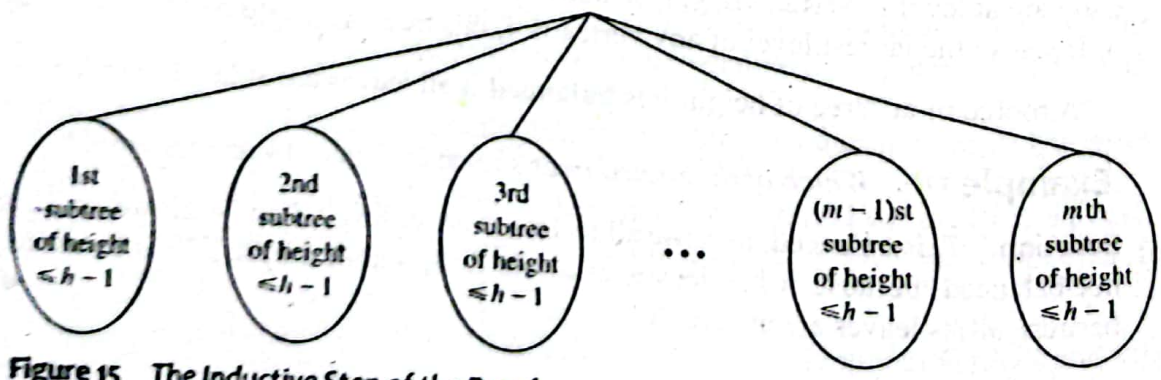
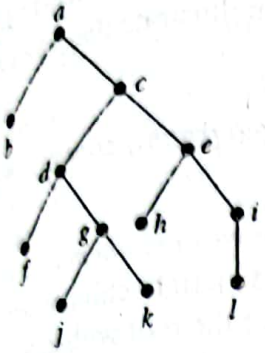
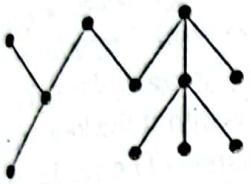
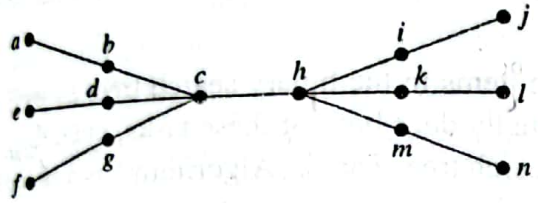


Figure 15 The Inductive Step of the Proof.

39.


40.


41.


42. Show that a center should be chosen as the root to produce a rooted tree of minimal height from an unrooted tree.

*43. Show that a tree has either one center or two centers that are adjacent.

44. Show that every tree can be colored using two colors.

The **rooted Fibonacci trees** T_n are defined recursively in the following way. T_1 and T_2 are both the rooted tree consisting of a single vertex, and for $n = 3, 4, \ldots$, the rooted tree T_n is constructed from a root with T_{n-1} as its left subtree and T_{n-2} as its right subtree.

45. Draw the first seven rooted Fibonacci trees.

*46. How many vertices, leaves, and internal vertices does the rooted Fibonacci tree T_n have, where n is a positive integer? What is its height?

47. What is wrong with the following "proof" using mathematical induction of the statement that every tree with n vertices has a path of length $n - 1$. *Basis step:* Every tree with one vertex clearly has a path of length 0. *Inductive step:* Assume that a tree with n vertices has a path of length $n - 1$, which has u as its terminal vertex. Add a vertex and the edge from u to. The resulting tree has $n + 1$ vertices and has a path of length n. This completes the induction step.

48. Show that the average depth of a leaf in a binary tree with n vertices is $\Omega(\log n)$.

9.2 APPLICATIONS OF TREES

Introduction We will discuss three problems that can be studied using trees. The first problem is: How should items in a list be stored so that an item can be easily located? The second problem is: What series of decisions should be made to find an object with a certain property in a collection of objects of a certain type? The third problem is: How should a set of characters be efficiently coded by bit strings?

Binary Search Trees Searching for items in a list is one of the most important tasks that arises in computer science. Our primary goal is to implement a searching algorithm that finds items efficiently when the items are totally ordered. This can be accomplished through the use of a **binary search tree**, which is a binary tree in which each child of a vertex is designated as a right or left child, no vertex has more than one right child or left child, and each vertex is labeled with a key, which is one of the items. Furthermore, vertices are assigned keys so that the key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.

This recursive procedure is used to form the binary search tree for a list of items. Start with a tree containing just one vertex, namely, the root. The first item in the list is assigned as the key of the root. To add a new item, first compare it with the keys of vertices already in the tree, starting at the root and moving to the left if the item is less than the key of the respective vertex if this vertex has a left child, or moving to the right if the item is greater than the key of the respective vertex if this vertex has a right child. When the item is less than the respective vertex and this vertex has no left child, then a new vertex with this item as its key is inserted

as a new left child. Similarly, when the item is greater than the respective vertex and this vertex has no right child, then a new vertex with this item as its key is inserted as a new right child. We illustrate this procedure with Example 1.

Example 1 *Forma binary search tree for the words mathematics, physics, geography, zoology, meteorology, geology, psychology, and chemistry (using alphabetical order).*

Solution Figure 1 displays the steps used to construct this binary search tree. The word *mathematics* is the key of the root. Because *physics* comes after *mathematics* (in alphabetical order), add a right child of the root with key *physics*. Because *geography* comes before *mathematics*, add a left child of the root with key *geography*. Next, add a right child of the vertex with key *physics*, and assign it the key *zoology*, because *zoology* comes after *mathematics* and after *physics*. Similarly, add a left child of the vertex with key *physics* and assign this new vertex the key *meteorology*. Add a right child of the vertex with key *geography* and assign this new vertex the key *geology*. Add a left child of the vertex with key *zoology* and assign it the key *psychology*. Add a left child of the vertex with key *geography* and assign it the key *chemistry*. (The reader should work through all the comparisons needed at each step.) ◀

Once we have a binary search tree, we need a way to locate items in the binary search tree, as well as a way to add new items. Algorithm 1, an insertion algorithm, actually does both of these tasks, even though it may appear that it is only designed to add vertices to a binary search tree. That is, Algorithm 1 is a procedure that locates an item x in a binary search tree if it is present, and adds a new vertex with x as its key if x is not present. In the pseudocode, v is the vertex currently under examination and *label*(v) represents the key of this vertex. The algorithm begins by examining the root. If the x equals the key of v, then the algorithm has found the location of x and terminates; if x is less than the key of v, we move to the left child of v and repeat the procedure; and if x is greater than the key of v, we move to the right child of and repeat the procedure. If at any step we attempt to move to a child that is not present, we know that x is not present in the tree, and we add a new vertex as this child with x as its key.

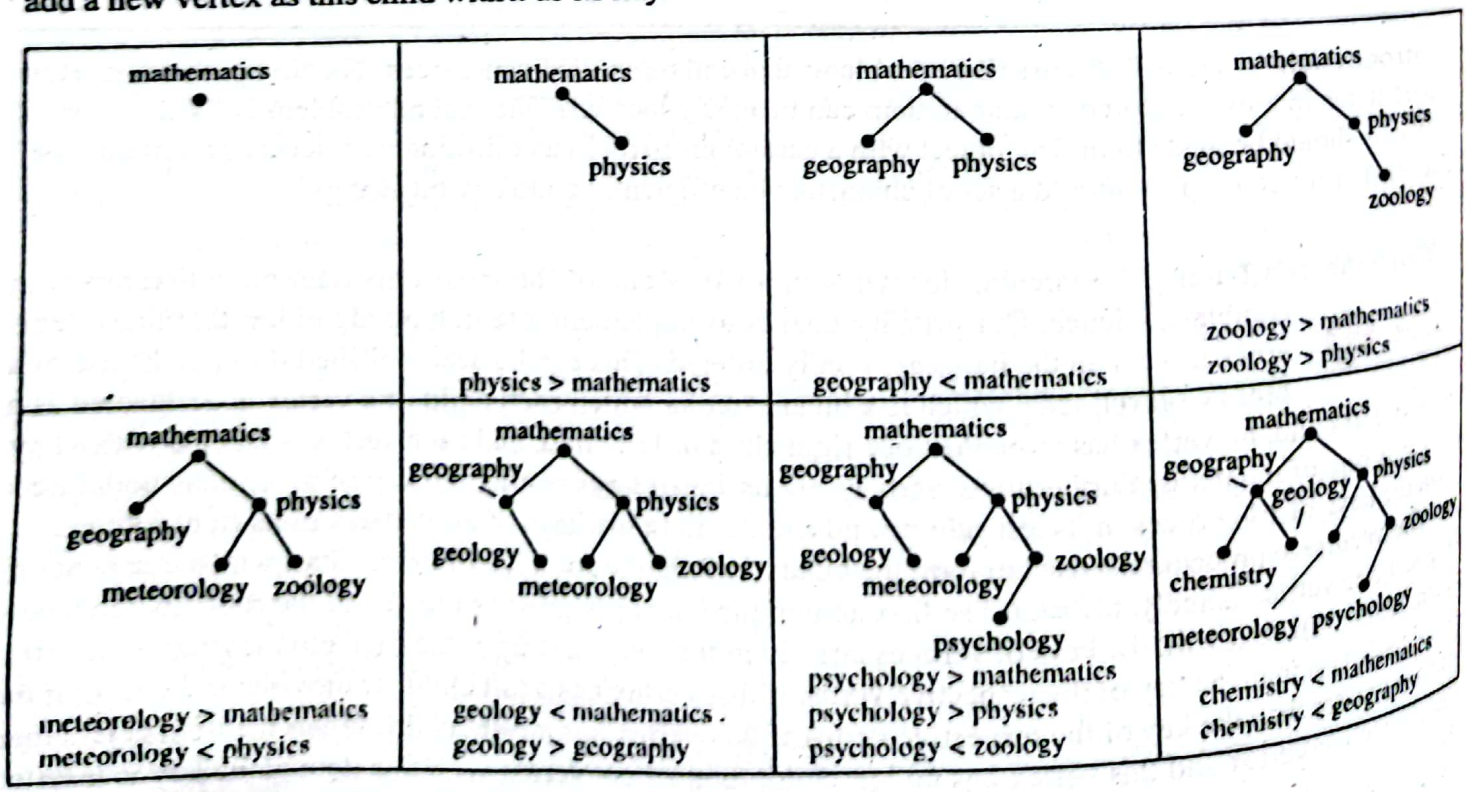


Figure 1 *Constructing a Binary Search Tree.*

ALGORITHM 1 Locating and Adding Items to a Binary Search Tree.

procedure *insertion*(T: binary search tree, x: item)
v := root of T
{a vertex not present in T has the value *null*}
while $v \neq null$ and $label(v) \neq x$
begin
  if $x < label(v)$ **then**
    if left child of $v \neq null$ **then** $v :=$ left child of v
    else add *new vertex* as a left child of v and set $v := null$
  else
    if right child of $v \neq null$ **then** $v :=$ right child of v
    else add *new vertex* as a right child of v to T and set $v := null$
end
if root of $T = null$ **then** add a vertex v to the tree and label it with x
else if v is null or $label(v) \neq x$ **then** label *new vertex* with x and let v be this new vertex
{v = location of x}

Example 2 illustrates the use of Algorithm1 to insert a new item into a binary search tree.

Example 2 *Use Algorithm*1 *to insert the word oceanography into the binary search tree in Example* 1.

Solution Algorithm 1 begins with v, the vertex under examination, equal to the root of T, so $label(v) =$ *mathematics*. Because $v \neq null$ and $label(v) =$ *mathematics* $<$ *oceanography*, we next examine the right child of the root. This right child exists, so we set v, the vertex under examination, to be this right child. At this step we have $v \neq null$ and $label(v) =$ *physics* $>$ *oceanography*, so we examine the left child of . This left child exists, so we set , the vertex under examination, to this left child. At this step, we also have $v \neq null$ and $label(v) =$ *metereology* $<$ *oceanography*, so we try to examine the right child of . However, this right child does not exist, so we add a new vertex as the right child of v (which at this point is the vertex with the key *metereology*) and we set $v := null$. We now exit the **while** loop because $v = null$. Because the root of T is not *null* and $v = null$, we use the **else if** statement at the end of the algorithm to label our new vertex with the key *oceanography*. ◀

We will now determine the computational complexity of this procedure. Suppose we have a binary search tree T for a list of n items. We can form a full binary tree U from T by adding unlabeled vertices whenever

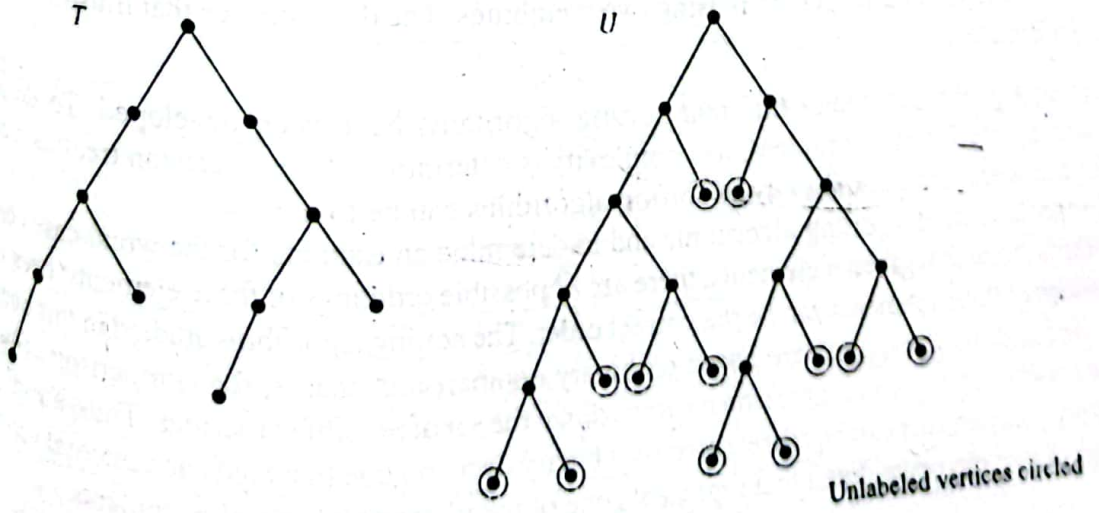


Figure 2 Adding Unlabeled Vertices to Make a Binary Search Tree Full.

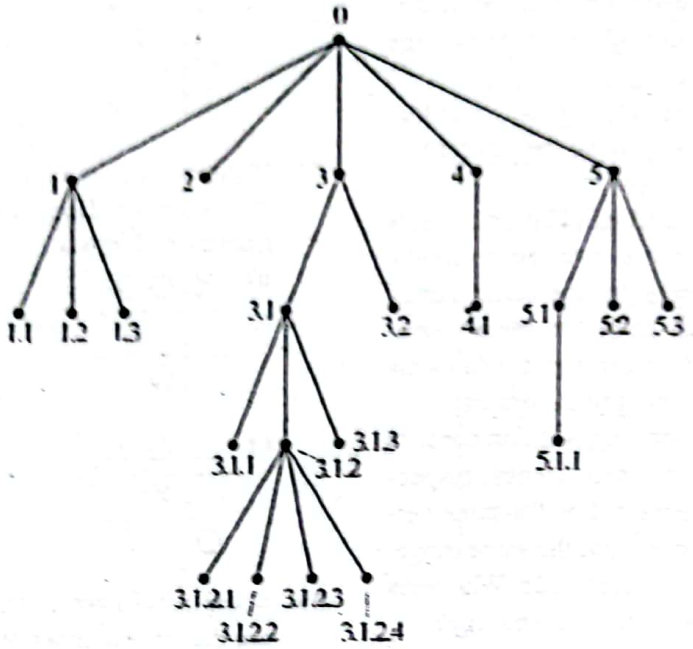


Figure 1 *The Universal Address System of an Ordered Rooted Tree.*

numbers, variables, and operations. The different listings of the vertices of ordered rooted trees used to represent expressions are useful in the evaluation of these expressions.

Universal Address Systems Procedures for traversing all vertices of an ordered rooted tree rely on the orderings of children. In ordered rooted trees, the children of an internal vertex are shown from left to right in the drawings representing these directed graphs.

We will describe one way we can totally order the vertices of an ordered rooted tree. To produce this ordering, we must first label all the vertices. We do this recursively:

1. Label the root with the integer 0. Then label its k children (at level 1) from left to right with $1, 2, 3, \ldots, k$.
2. For each vertex at level n with label A, label its k children, as they are drawn from left to right, with $A.1, A.2, \ldots, A.k$.

Following this procedure, a vertex at level n, for $n \geq 1$, is labeled $x_1 . x_2, \ldots, x_n$, where the unique path from the root to goes through the x_1st vertex at level 1, the x_2nd vertex at level 2, and so on. This labeling is called the **universal address system** of the ordered rooted tree.

We can totally order the vertices using the lexicographic ordering of their labels in the universal address system. The vertex labeled $x_1 . x_2, \ldots, x_n$ is less than the vertex labeled $y_1 . y_2, \ldots, y_m$ if there is an i, $0 \leq i \leq n$, with $x_1 = y_1, x_2 = y_2, \ldots, x_{i-1} = y_{i-1}$, and $x_i < y_i$; or if $n < m$ and $x_i = y_i$ for $i = 1, 2, \ldots, n$.

Example 1 We display the labelings of the universal address system next to the vertices in the ordered rooted tree shown in Figure 1. The lexicographic ordering of the labelings is

Extra Examples

$$0 < 1 < 1.1 < 1.2 < 1.3 < 2 < 3 < 3.1 < 3.1.1 < 3.1.2 < 3.1.2.1 < 3.1.2.2 < 3.1.2.3 < 3.1.2.4 < 3.1.3 < 3.2 < 4 < 4.1 < 5 < 5.1 < 5.1.1 < 5.2 < 5.3$$

Traversal Algorithms Procedures for systematically visiting every vertex of an ordered rooted tree are called **traversal algorithms**. We will describe three of the most commonly used such algorithms, **preorder traversal**, **inorder traversal**, and **postorder traversal**. Each of these algorithms can be defined recursively. We first define preorder traversal.

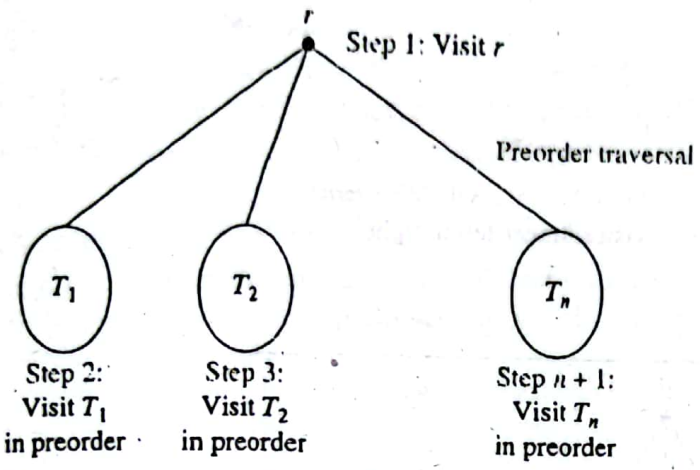


Figure 2 *Preorder Traversal.*

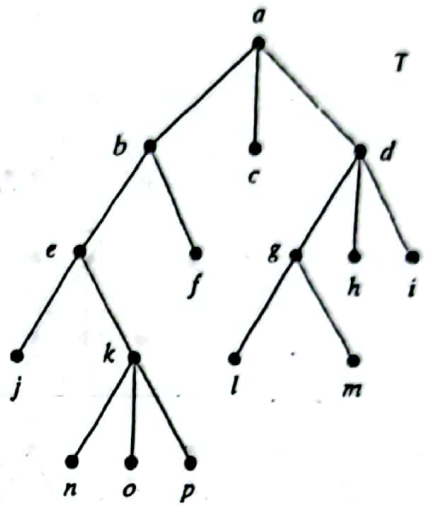


Figure 3 *The Ordered Rooted Tree T.*

Definition 1 Let T be an ordered rooted tree with root r. If T consists only of r, then r is the *preorder traversal* of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right in T. The *preorder traversal* begins by visiting r. It continues by traversing T_1 in preorder, then T_2 in preorder, and so on, until T_n is traversed in preorder.

The reader should verify that the preorder traversal of an ordered rooted tree gives the same ordering of the vertices as the ordering obtained using a universal address system. Figure 2 indicates how a preorder traversal is carried out.

Example 2 illustrates preorder traversal.

Example 2 *In which order does a preorder traversal visit the vertices in the ordered rooted tree T shown in Figure 3?*

Extra Examples

Solution The steps of the preorder traversal of T are shown in Figure 4. We traverse T in preorder by first listing the root a, followed by the preorder list of the subtree with root b, the preorder list of the subtree with root c (which is just c) and the preorder list of the subtree with root d.

The preorder list of the subtree with root b begins by listing b, then the vertices of the subtree with root e in preorder, and then the subtree with root f in preorder (which is just f). The preorder list of the subtree with root d begins by listing d, followed by the preorder list of the subtree with root g, followed by the subtree with root h (which is just h), followed by the subtree with root i (which is just i). ◀

The preorder list of the subtree with root e begins by listing e, followed by the preorder listing of the subtree with root j (which is just j), followed by the preorder listing of the subtree with root k. The preorder listing of the subtree with root g is g followed by l, followed by m. The preorder listing of the subtree with root k is k, n, o, p. Consequently, the preorder traversal of T is $a, b, e, j, k, n, o, p, f, c, d, g, l, m, h, i$. ◀

We will now define inorder traversal.

Definition 2 Let T be an ordered rooted tree with root r. If T consists only of r, then r is the *inorder traversal* of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right. The *inorder traversal* begins by traversing T_1 in inorder, then visiting r. It continues by traversing T_2 in inorder, then T_3 in inorder, ..., and finally T_n in inorder.

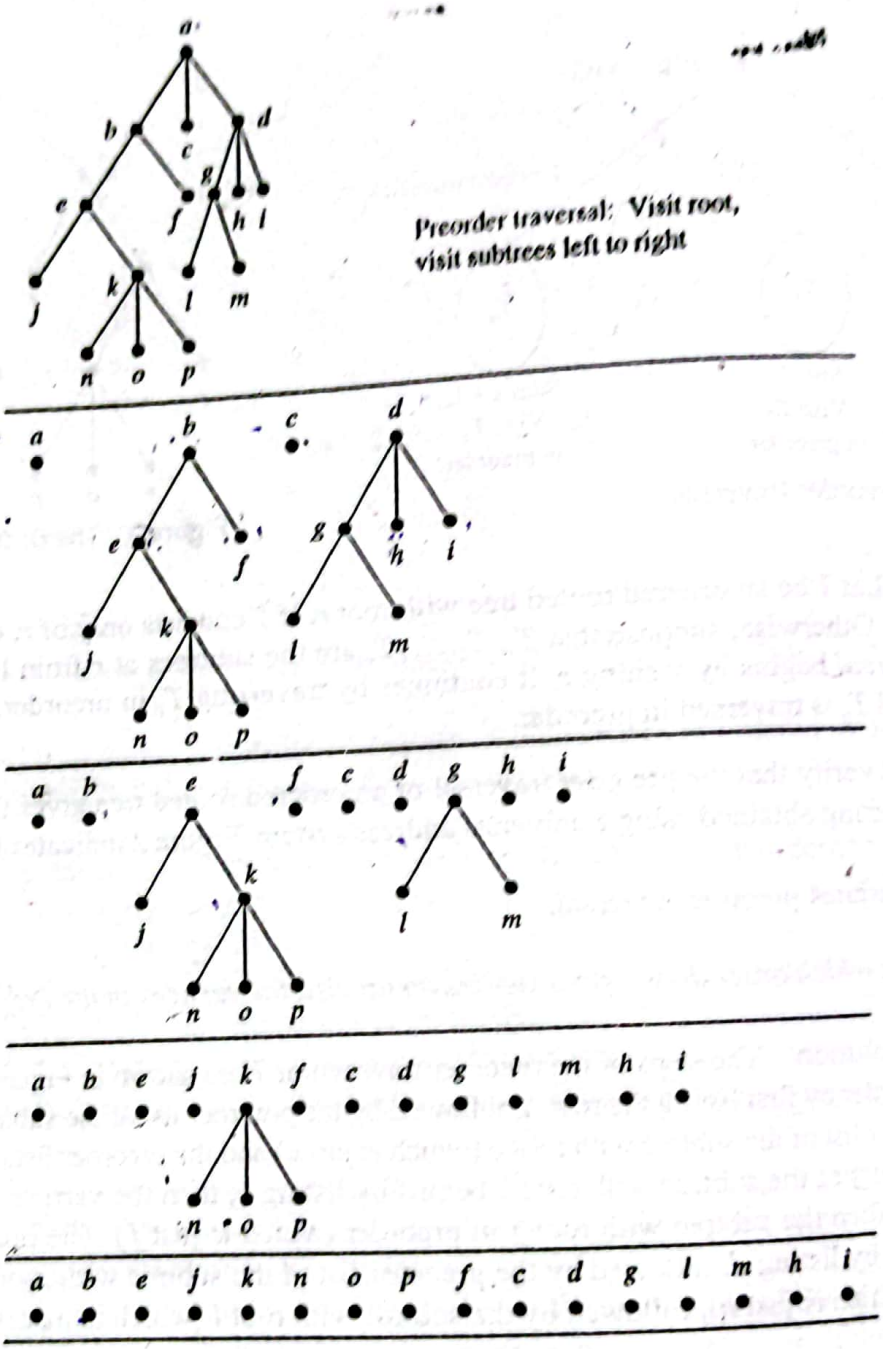


Figure 4 The Preorder Traversal of T.

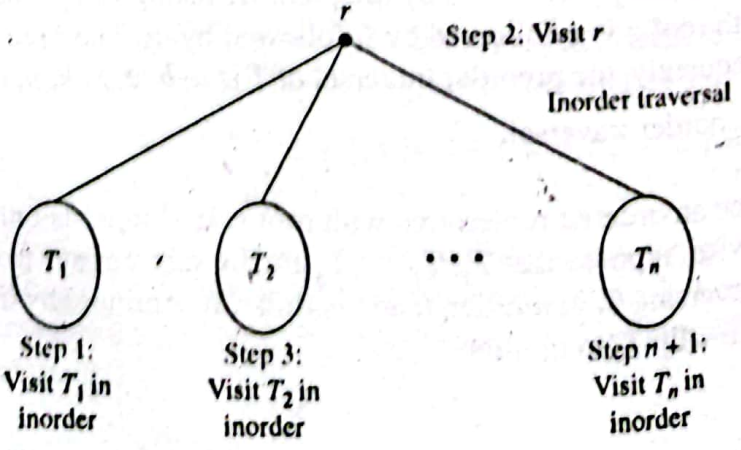


Figure 5 Inorder Traversal.

Figure 5 indicates how inorder traversal is carried out. Example 3 illustrates how inorder traversal is carried out for a particular tree.

Example 3 *In which order does an inorder traversal visit the vertices of the ordered rooted tree T in Figure 3?*

Extra Examples

Solution The steps of the inorder traversal of the ordered rooted tree T are shown in Figure 6. The inorder traversal begins with an inorder traversal of the subtree with root b, the root a, the inorder listing of the subtree with root c, which is just c, and the inorder listing of the subtree with root d.

The inorder listing of the subtree with root b begins with the inorder listing of the subtree with root e, the root b, and f. The inorder listing of the subtree with root d begins with the inorder listing of the subtree with root g, followed by the root d, followed by h, followed by i.

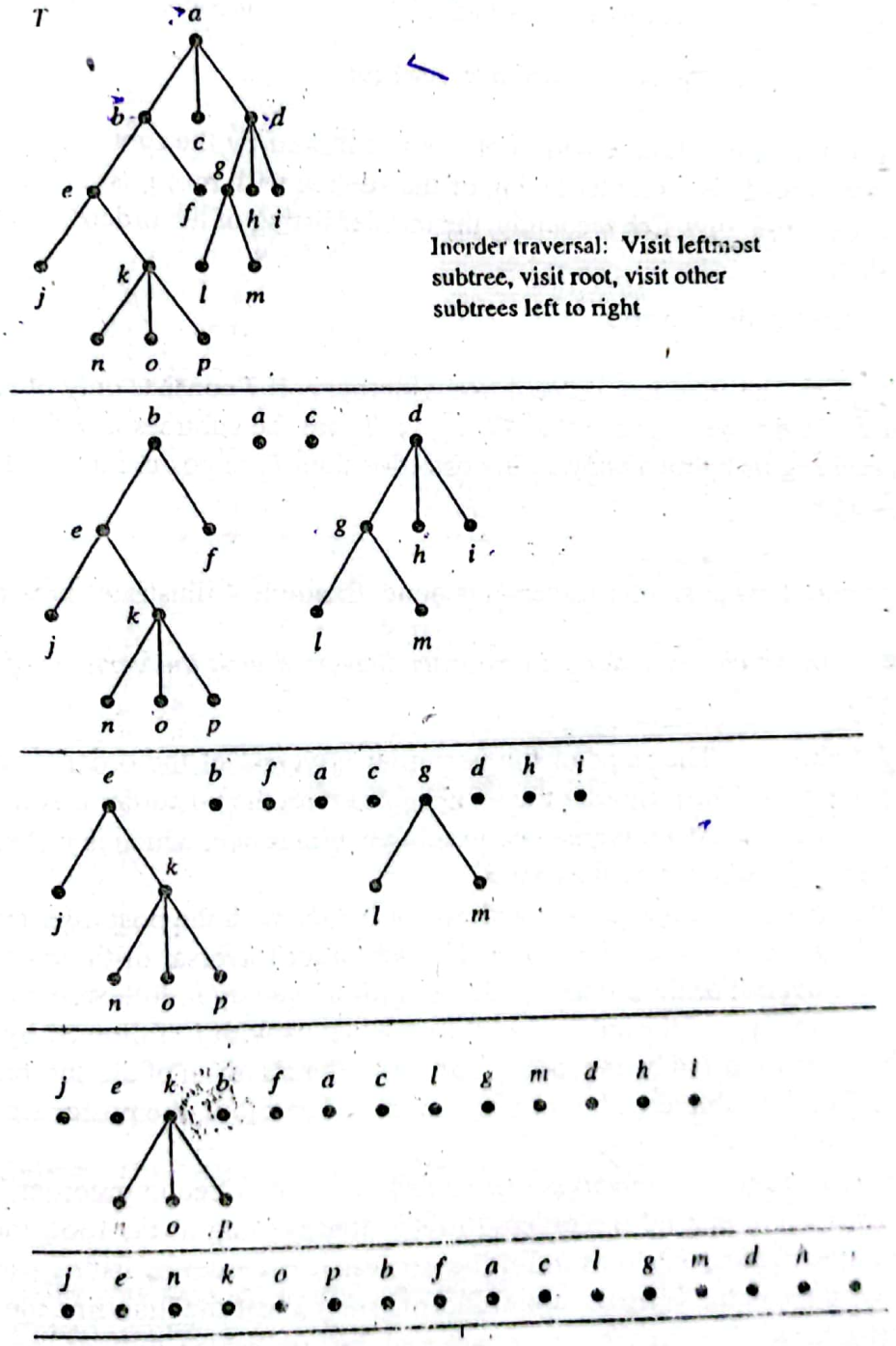


Figure 6 The Inorder Traversal of T.

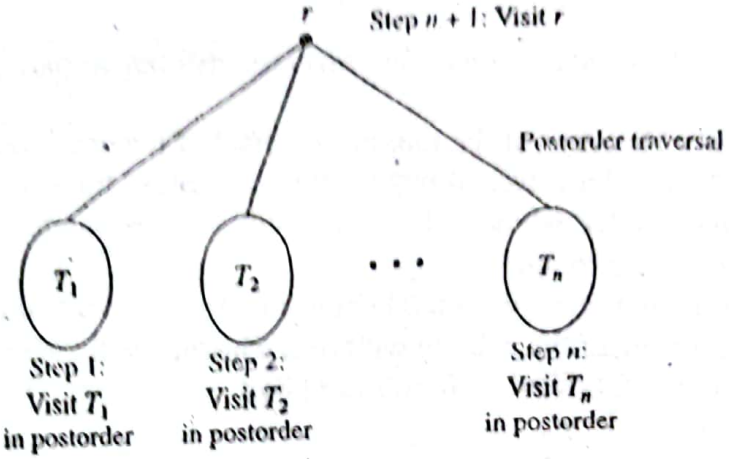


Figure 7 Postorder Traversal.

The inorder listing of the subtree with root e is j, followed by the root e, followed by the inorder listing of the subtree with root k. The inorder listing of the subtree with root g is l, g, m. The inorder listing of the subtree with root k is n, k, o, p. Consequently, the inorder listing of the ordered rooted tree is j, e, n, k, o, p, b, f, a, c, l, g, m, d, h, i.

We now define postorder traversal.

Definition 3 Let T be an ordered rooted tree with root r. If T consists only of r, then r is the *postorder traversal* of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees at r from left to right. The *postorder traversal* begins by traversing T_1 in postorder, then T_2 in postorder, ..., then T_n in postorder, and ends by visiting r.

Figure 7 illustrates how postorder traversal is done. Example 4 illustrates how postorder traversal works.

Example 4 *In which order does a postorder traversal visit the vertices of the ordered rooted tree T shown in Figure 3?*

Extra Examples

Solution The steps of the postorder traversal of the ordered rooted tree T are shown in Figure 8. The postorder traversal begins with the postorder traversal of the subtree with root b, the postorder traversal of the subtree with root c, which is just c, the postorder traversal of the subtree with root d, followed by the root a.

The postorder traversal of the subtree with root b begins with the postorder traversal of the subtree with root e, followed by f, followed by the root b. The postorder traversal of the rooted tree with root d begins with the postorder traversal of the subtree with root g, followed by h, followed by i, followed by the root d.

The postorder traversal of the subtree with root e begins with j, followed by the postorder traversal of the subtree with root k, followed by the root e. The postorder traversal of the subtree with root g is l, m, g. The postorder traversal of the subtree with root k is n, o, p, k. Therefore, the postorder traversal of T is j, n, o, p, k, e, f, b, c, l, m, g, h, i, d, a.

There are easy ways to list the vertices of an ordered rooted tree in preorder, inorder, and postorder. To do this, first draw a curve around the ordered rooted tree starting at the root, moving along the edges, as shown in the example in Figure 9. We can list the vertices in preorder by listing each vertex the first time this curve passes it. We can list the vertices in inorder by listing a leaf the first time the curve passes it and listing each internal vertex the second time the curve passes it. We can list the vertices in postorder by listing a vertex

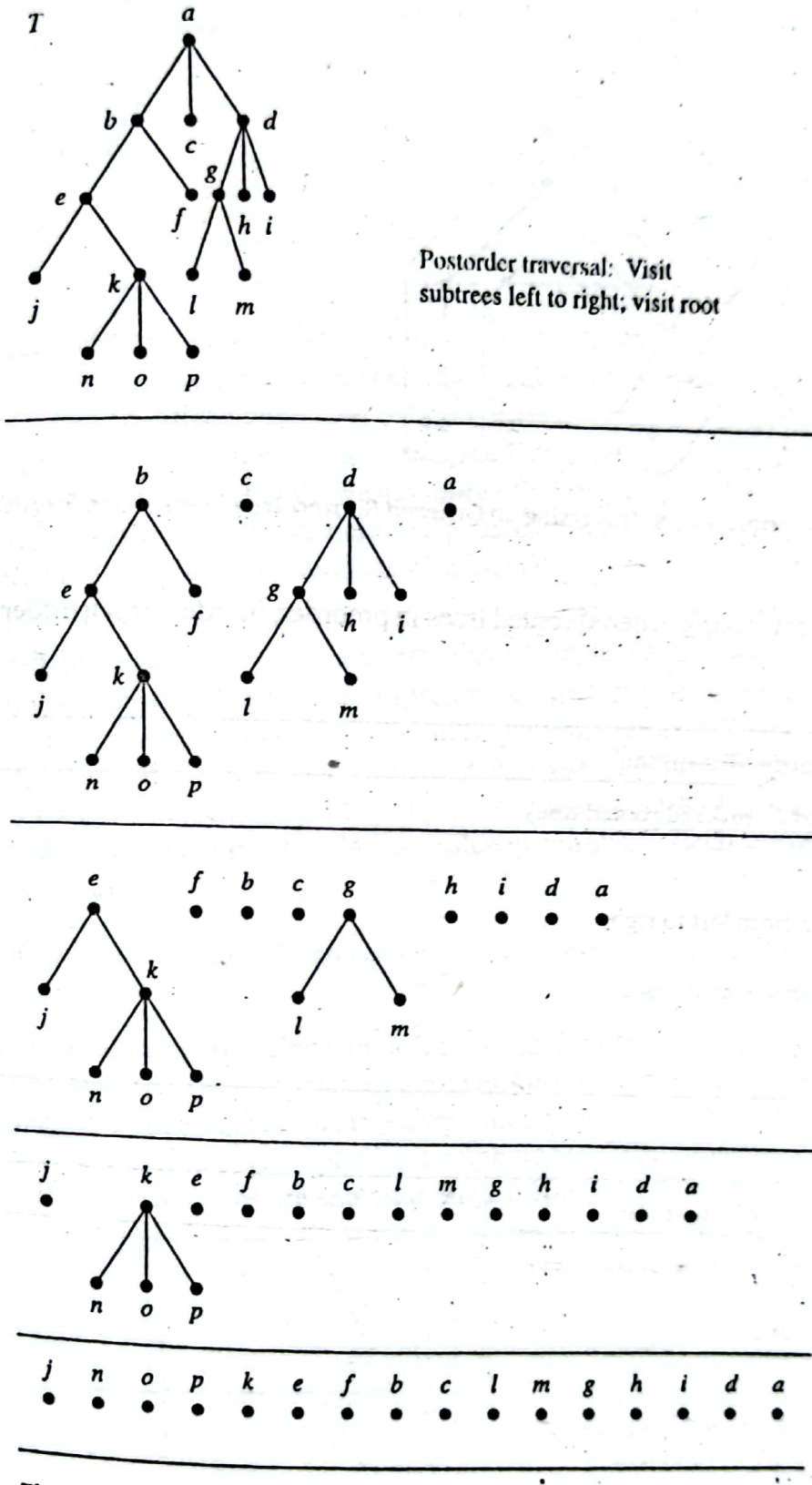


Figure 8 *The Postorder Traversal of T.*

the last time it is passed on the way back up to its parent. When this is done in the rooted tree in Figure 9, i follows that the preorder traversal gives $a, b, d, h, e, i, j, c, f, g, k$, the inorder traversal gives h, d, b, i, e, j a, f, c, k, g; and the postorder traversal gives $h, d, i, j, e, b, f, k, g, c, a$.

ALGORITHM 3 Postorder Traversal.

```
procedure postorder(T: ordered rooted tree)
r := root of T
for each child c of r from left to right begin
    T(c) := subtree with c as its root
    postorder(T(c))
end
list r
```

Note that both the preorder traversal and the postorder traversal encode the structure of an ordered rooted tree when the number of children of each vertex is specified. That is, an ordered rooted tree is uniquely determined when we specify a list of vertices generated by a preorder traversal or by a postorder traversal of the tree, together with the number of children of each vertex (see Exercises 26 and 27). In particular, both a preorder traversal and a postorder traversal encode the structure of a full ordered m-ary tree. However, when the number of children of vertices is not specified, neither a preorder traversal nor a postorder traversal encodes the structure of an ordered rooted tree (see Exercises 28 and 29).

Infix, Prefix, and Postfix Notation We can represent complicated expressions, such as compound propositions, combinations of sets, and arithmetic expressions using ordered rooted trees. For instance, consider the representation of an arithmetic expression involving the operators + (addition), − (subtraction), * (multiplication), / (division), and ↑ (exponentiation). We will use parentheses to indicate the order of the operations. An ordered rooted tree can be used to represent such expressions, where the internal vertices represent operations, and the leaves represent the variables or numbers. Each operation operates on its left and right subtrees (in that order).

Example 5 *What is the ordered rooted tree that represents the expression* $((x + y) \uparrow 2) + ((x - 4)/3)$?

Solution The binary tree for this expression can be built from the bottom up. First, a subtree for the expression $x + y$ is constructed. Then this is incorporated as part of the larger subtree representing $(x + y) \uparrow 2$. Also, a subtree for $x - 4$ is constructed, and then this is incorporated into a subtree representing $(x - 4)/3$. Finally the subtrees representing $(x + y) \uparrow 2$ and $(x - 4)/3$ are combined to form the ordered rooted tree representing $((x + y) \uparrow 2) + ((x - 4)/3)$. These steps are shown in Figure 10. ◀

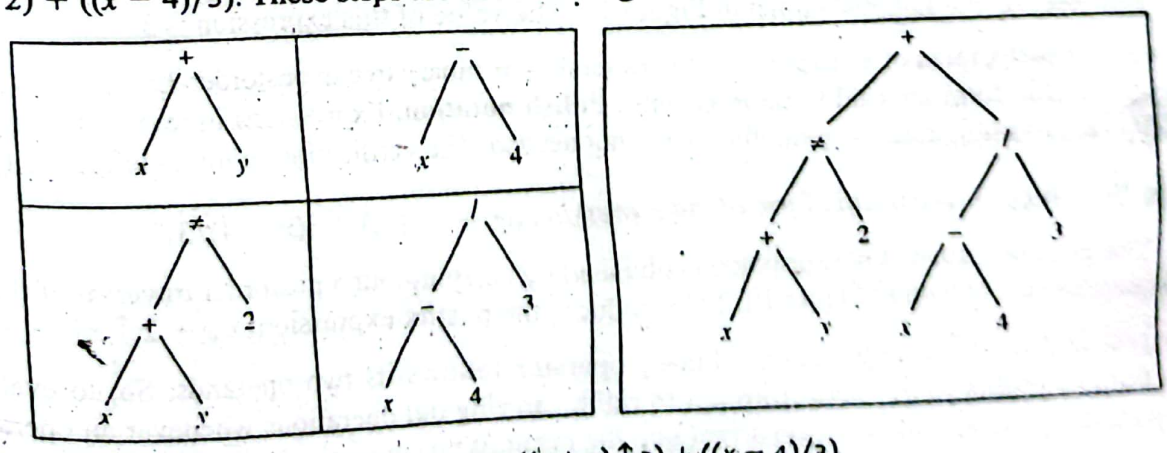


Figure 10 A Binary Tree Representing $((x + y) \uparrow 2) + ((x - 4)/3)$.

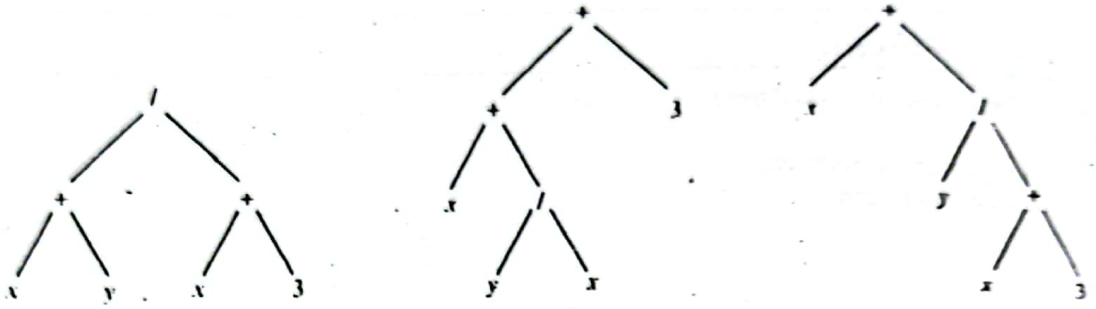


Figure 11 *Rooted Trees Representing* $(x+y)/(x+3)$, $(x+(y/x))+3$, and $x+(y/(x+3))$.

An inorder traversal of the binary tree representing an expression produces the original expression with the elements and operations in the same order as they originally occurred, except for unary operations, which instead immediately follow their operands. For instance, inorder traversals of the binary trees in Figure 11, which represent the expressions $(x + y)/(x + 3)$, $(x + (y/x)) + 3$, and $x + (y/(x + 3))$, all lead to the infix expression $x + y/x + 3$. To make such expressions unambiguous it is necessary to include parentheses in the inorder traversal whenever we encounter an operation. The fully parenthesized expression obtained in this way is said to be in **infix form**.

We obtain the **prefix form** of an expression when we traverse its rooted tree in preorder. Expressions written in prefix form are said to be in **Polish notation**, which is named after the Polish logician Jan Łukasiewicz. An expression in prefix notation (where each operation has a specified number of operands), is unambiguous, so no parentheses are needed in such an expression. The verification of this is left as an exercise for the reader.

Example 6 *What is the prefix form for* $((x + y) \uparrow 2) + ((x - 4)/3)$?

Solution We obtain the prefix form for this expression by traversing the binary tree that represents it, shown in Figure 10. This produces $+ \uparrow + x\, y\, 2\, /\, -x\, 4\, 3$. ◀

In the prefix form of an expression, a binary operator, such as $+$, precedes its two operands. Hence, we can evaluate an expression in prefix form by working from right to left. When we encounter an operator, we perform the corresponding operation with the two operands immediately to the right of this operand. Also, whenever an operation is performed, we consider the result a new operand.

Example 7 *What is the value of the prefix expression* $+\ -\ *\,2\,3\,5/\uparrow 2\,3\,4$?

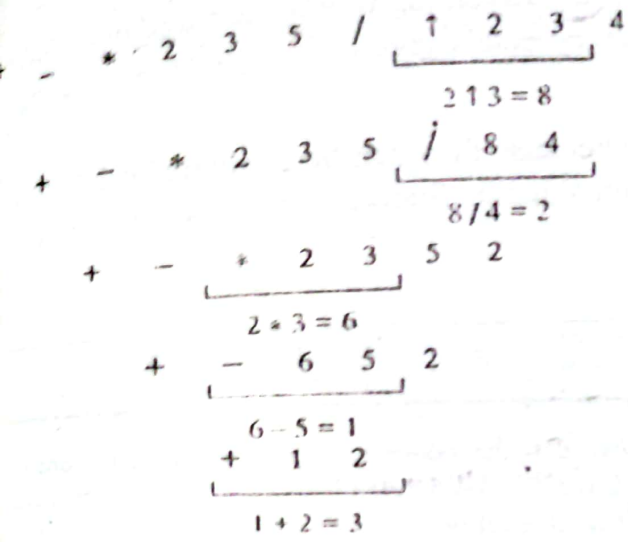
Solution The steps used to evaluate this expression by working right to left, and performing operations using the operands on the right, are shown in Figure 12. The value of this expression is 3. ◀

We obtain the **postfix form** of an expression by traversing its binary tree in postorder. Expressions written in postfix form are said to be in **reverse Polish notation**. Expressions in reverse Polish notation are unambiguous, so parentheses are not needed. The verification of this is left to the reader.

Example 8 *What is the postfix form of the expression* $((x + y) \uparrow 2) + ((x - 4)/3)$?

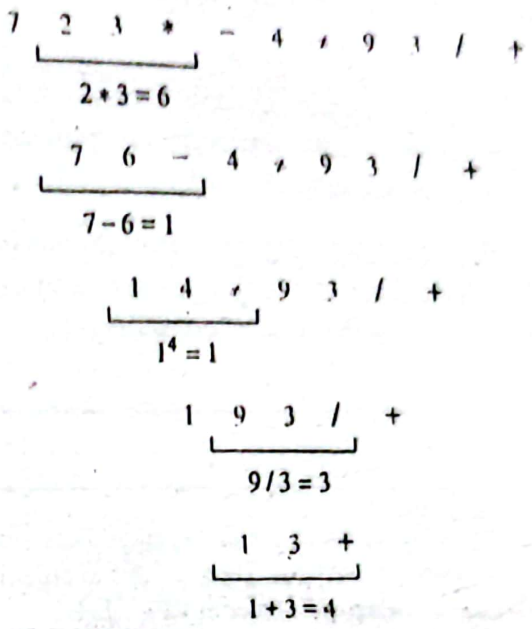
Solution The postfix form of the expression is obtained by carrying out a postorder traversal of the binary tree for this expression, shown in Figure 10. This produces the postfix expression: $x\, y + 2 \uparrow x\, 4 - 3/+$. ◀

In the postfix form of an expression, a binary operator follows its two operands. So, to evaluate an expression from its postfix form, work from left to right, carrying out operations whenever an operator follows two operands. After an operation is carried out, the result of this operation becomes a new operand.



Value of expression: 3

Figure 12 *Evaluating a Prefix Expression.*



Value of expression: 4

Figure 13 *Evaluating a Postfix Expression.*

Example 9 *What is the value of the postfix expression* 7 2 3 * − 4 ↑ 9 3/+?

Solution: The steps used to evaluate this expression by starting at the left and carrying out operations when two operands are followed by an operator are shown in Figure 13. The value of this expression is 4. ◀

Rooted trees can be used to represent other types of expressions, such as those representing compound propositions and combinations of sets. In these examples unary operators, such as the negation of a proposition, occur. To represent such operators and their operands, a vertex representing the operator and a child of this vertex representing the operand are used.

Example 10 *Find the ordered rooted tree representing the compound proposition* $(\neg(p \wedge q)) \leftrightarrow (\neg p \vee \neg q)$. *Then use this rooted tree to find the prefix, postfix, and infix forms of this expression.*

Extra Examples

Solution: The rooted tree for this compound proposition is constructed from the bottom up. First, subtrees for $\neg p$ and $\neg q$ are formed (where $\neg$ is considered a unary operator). Also, a subtree for $p \wedge q$ is formed. Then subtrees for $\neg(p \wedge q)$ and $(\neg p) \vee (\neg q)$ are constructed. Finally, these two subtrees are used to form the final rooted tree. The steps of this procedure are shown in Figure 14.

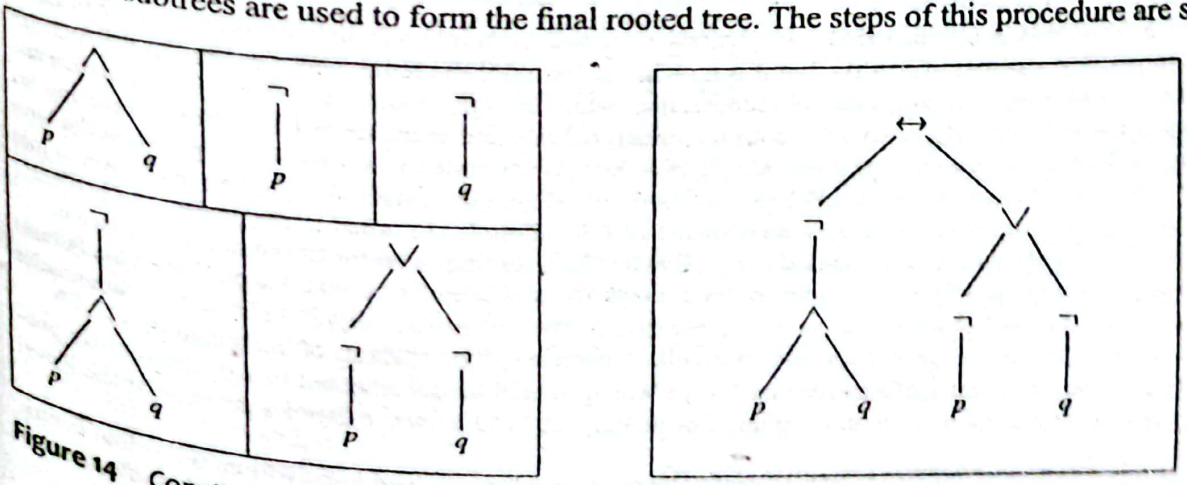


Figure 14 *Constructing the Rooted Tree for a Compound Proposition.*

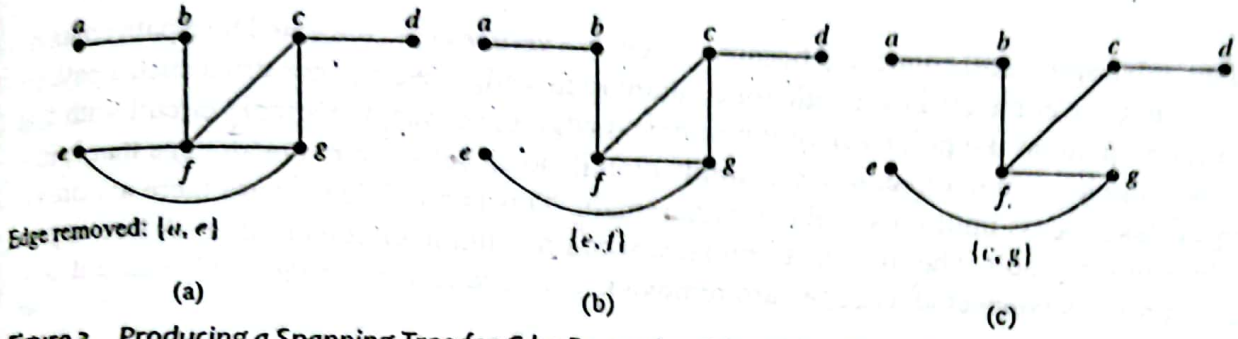


Figure 3 Producing a Spanning Tree for G by Removing Edges That Form Simple Circuits.

Definition 1 Let G be a simple graph. A *spanning tree* of G is a subgraph of G that is a tree containing every vertex of G.

A simple graph with a spanning tree must be connected, because there is a path in the spanning tree between any two vertices. The converse is also true; that is, every connected simple graph has a spanning tree. We will give an example before proving this result.

Example 1 *Find a spanning tree of the simple graph G shown in Figure 2.*

Solution The graph G is connected, but it is not a tree because it contains simple circuits. Remove the edge $\{a, e\}$. This eliminates one simple circuit, and the resulting subgraph is still connected and still contains every vertex of G. Next remove the edge $\{e, f\}$ to eliminate a second simple circuit. Finally, remove edge $\{c, g\}$ to produce a simple graph with no simple circuits. This subgraph is a spanning tree, because it is a tree that contains every vertex of G. The sequence of edge removals used to produce the spanning tree is illustrated in Figure 3.

The tree shown in Figure 3 is not the only spanning tree of G. For instance, each of the trees shown in Figure 4 is a spanning tree of G. ◀

Theorem 1 A simple graph is connected if and only if it has a spanning tree.

Proof: First, suppose that a simple graph G has a spanning tree T. T contains every vertex of G. Furthermore, there is a path in T between any two of its vertices. Because T is a subgraph of G, there is a path in G between any two of its vertices. Hence, G is connected.

Now suppose that G is connected. If G is not a tree, it must contain a simple circuit. Remove an edge from one of these simple circuits. The resulting subgraph has one fewer edge but still contains all the vertices of G

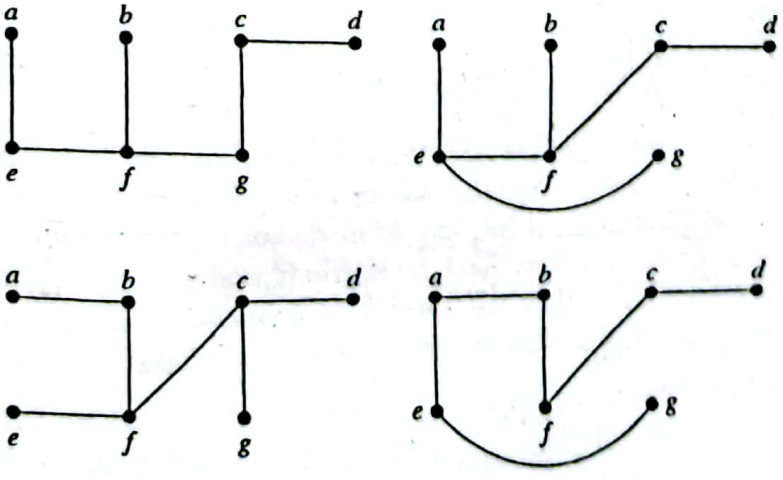


Figure 4 Spanning Trees of G.

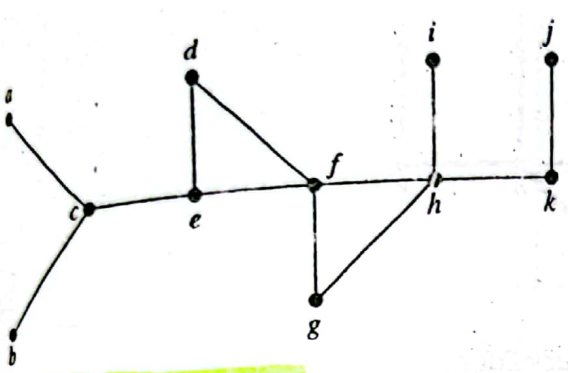


Figure 6 The Graph G

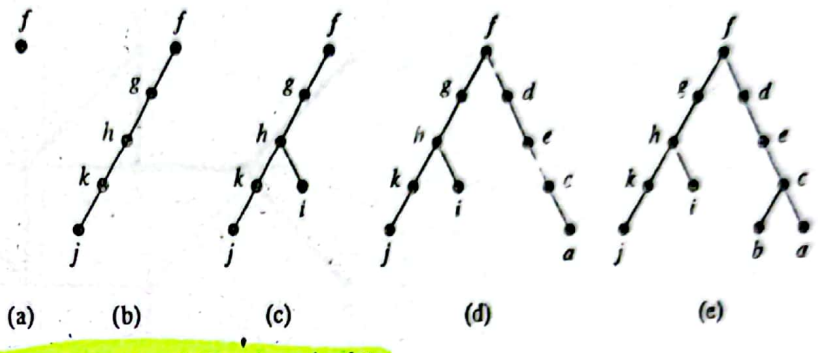


Figure 7 Depth-First Search of G

Depth-First Search

Links

The proof of Theorem1 gives an algorithm for finding spanning trees by removing edges from simple circuits. This algorithm is inefficient, because it requires that simple circuits be identified. Instead of constructing spanning trees by removing edges, spanning trees can be built up by successively adding edges. Two algorithms based on this principle will be presented here.

Demo

We can build a spanning tree for a connected simple graph using **depth-first search.** We will forma rooted tree, and the spanning tree will be the underlying undirected graph of this rooted tree. Arbitrarily choose a vertex of the graph as the root. Form a path starting at this vertex by successively adding vertices and edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding vertices and edges to this path as long as possible. If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree. However, if the path does not go through all vertices, more vertices and edges must be added. Move back to the next to last vertex in the path, and, if possible, forma new path starting at this vertex passing through vertices that were not already visited. If this cannot be done, move back another vertex in the path, that is, two vertices back in the path, and try again.

Repeat this procedure, beginning at the last vertex visited, moving back up the path one vertex at a time, forming new paths that are as long as possible until no more edges can be added. Because the graph has a finite number of edges and is connected, this process ends with the production of a spanning tree. Each vertex that ends a path at a stage of the algorithm will be a leaf in the rooted tree, and each vertex where a path is constructed starting at this vertex will be an internal vertex.

The reader should note the recursive nature of this procedure. Also, note that if the vertices in the graph are ordered, the choices of edges at each stage of the procedure are all determined when we always choose the first vertex in the ordering that is available. However, we will not always explicitly order the vertices of a graph.

Depth-first search is also called **backtracking,** because the algorithm returns to vertices previously visited to add paths. Example 3 illustrates backtracking.

Example 3 *Use depth-first search to find a spanning tree for the graph G shown in Figure 6.*

Extra Examples

Solution: The steps used by depth-first search to produce a spanning tree of G are shown in Figure 7. We arbitrarily start with the vertex f. A path is built by successively adding edges incident with vertices not already in the path, as long as this is possible. This produces a path f, g, h, k, j (note that other paths could have been built). Next, backtrack to k. There is no path beginning at k containing vertices not already visited. So we backtrack to h. Form the path h, i. Then backtrack to h, and then to f. From f build the path f, d, e, c, a. Then backtrack to c and form the path c, b. This produces the spanning tree. ◀

The edges selected by depth-first search of a graph are called **tree edges.** All other edges of the graph must connect a vertex to an ancestor or descendant of this vertex in the tree. These edges are called **back edges.** (Exercise 39 asks for a proof of this fact.)

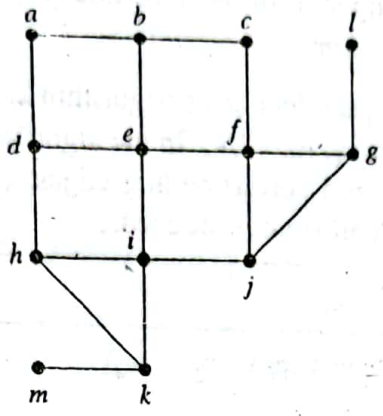


Figure 9 A Graph G.

tree as it is being built and adding this vertex and the corresponding edge if the vertex is not already in the tree. We have also made use of the inequality $e \leq n(n-1)/2$, which holds for any simple graph.]

Depth-first search can be used as the basis for algorithms that solve many different problems. For example, it can be used to find paths and circuits in a graph, it can be used to determine the connected components of a graph, and it can be used to find the cut vertices of a connected graph. As we will see, depth-first search is the basis of backtracking techniques used to search for solutions of computationally difficult problems. (See [GrYe99], [Ma89], and [CoLeRiSt01] for a discussion of algorithms based on depth-first search.)

Breadth-First Search We can also produce a spanning tree of a simple graph by the use of **breadth-first search.** Again, a rooted tree will be constructed, and the underlying undirected graph of this rooted tree forms the spanning tree. Arbitrarily choose a root from the vertices of the graph. Then add all edges incident to this vertex. The new vertices added at this stage become the vertices at level 1 in the spanning tree. Arbitrarily order them. Next, for each vertex at level 1, visited in order, add each edge incident to this vertex to the tree as long as it does not produce a simple circuit. Arbitrarily order the children of each vertex at level 1. This produces the vertices at level 2 in the tree. Follow the same procedure until all the vertices in the tree have been added. The procedure ends because there are only a finite number of edges in the graph. A spanning tree is produced because we have produced a tree containing every vertex of the graph. An example of breadth-first search is given in Example 5.

Demo Links

Example 5 *Use breadth-first search to find a spanning tree for the graph shown in Figure 9.*

Extra Examples

Solution The steps of the breadth-first search procedure are shown in Figure 10. We choose the vertex e to be the root. Then we add edges incident with all vertices adjacent to e, so edges from e to b, d, f, and i are added. These four vertices are at level 1 in the tree. Next, add the edges from these vertices at level 1 to adjacent vertices not already in the tree. Hence, the edges

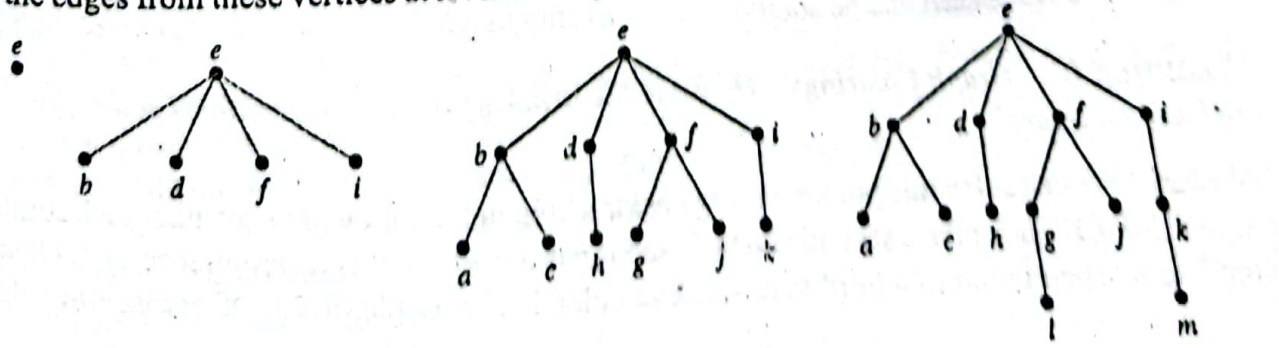


Figure 10 Breadth-First Search of G.

9.5 MINIMUM SPANNING TREES

Introduction A company plans to build a communications network connecting its five computer centers. Any pair of these centers can be linked with a leased telephone line. Which links should be made to ensure that there is a path between any two computer centers so that the total cost of the network is minimized? We can model this problem using the weighted graph shown in Figure 1, where vertices represent computer centers, edges represent possible leased lines, and the weights on edges are the monthly lease rates of the lines represented by the edges. We can solve this problem by finding a spanning tree so that the sum of the weights of the edges of the tree is minimized. Such a spanning tree is called a **minimum spanning tree.**

Algorithms for Minimum Spanning Trees A wide variety of problems are solved by finding a spanning tree in a weighted graph such that the sum of the weights of the edges in the tree is a minimum.

Definition 1 *A minimum spanning tree* in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.

We will present two algorithms for constructing minimum spanning trees. Both proceed by successively adding edges of smallest weight from those edges with a specified property that have not already been used. Both are greedy algorithms. Recall from Section 3.1 that a greedy algorithm is a procedure that makes an optimal choice at each of its steps. Optimizing at each step does not guarantee that the optimal overall solution is produced. However, the two algorithms presented in this section for constructing minimum spanning trees are greedy algorithms that do produce optimal solutions.

The first algorithm that we will discuss was given by Robert Prim in 1957, although the basic ideas of this algorithm have an earlier origin. To carry out **Prim's algorithm,** begin by choosing any edge with smallest weight, putting it into the spanning tree. Successively add to the tree edges of minimum weight that are incident to a vertex already in the tree and not forming a simple circuit with those edges already in the tree. Stop when $n - 1$ edges have been added.

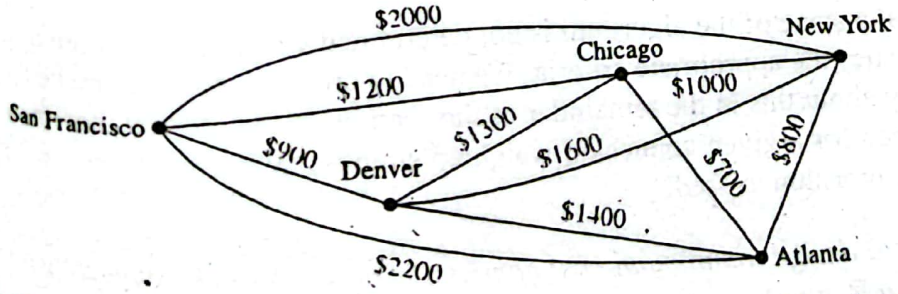
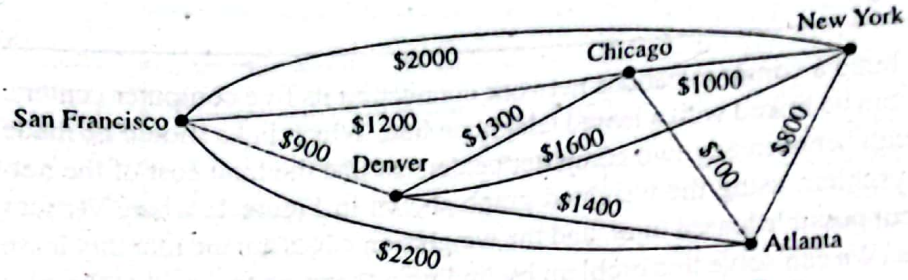


Figure 1 A Weighted Graph Showing Monthly Lease Costs for Lines in a Computer Network.

ROBERT CLAY PRIM (BORN 1921) Robert Prim, born in Sweet water, Texas, received his B.S. in electrical engineering in 1941 and his Ph.D. in mathematics from Princeton University in 1949. He was an engineer at the General Electric Company from 1941 until 1944, an engineer and mathematician at the United States Naval Ordnance Lab from 1944 until 1949, and a research associate at Princeton University from 1948 until 1949. Among the other positions he has held are director of mathematics and mechanics research at Bell Telephone Laboratories from 1958 until 1961 and vice president of research at Sandia Corporation. He is currently retired.



Choice	Edge	Cost
1	{Chicago, Atlanta}	$ 700
2	{Atlanta, New York}	$ 800
3	{Chicago, San Francisco}	$1200
4	{San Francisco, Denver}	$ 900
	Total:	$3600

Figure 2 *A Minimum Spanning Tree for the Weighted Graph in Figure 1.*

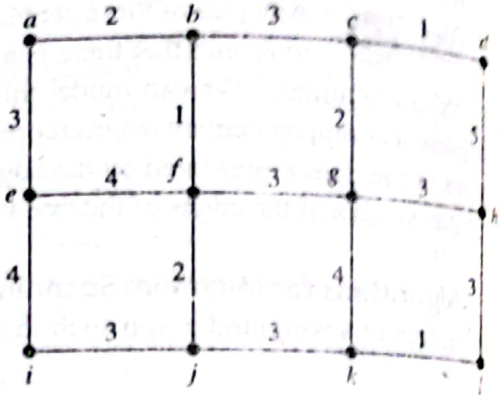


Figure 3 *A Weighted Graph.*

Later in this section, we will prove that this algorithm produces a minimum spanning tree for any connected weighted graph. Algorithm 1 gives a pseudocode description of Prim's algorithm.

ALGORITHM 1 Prim's Algorithm.

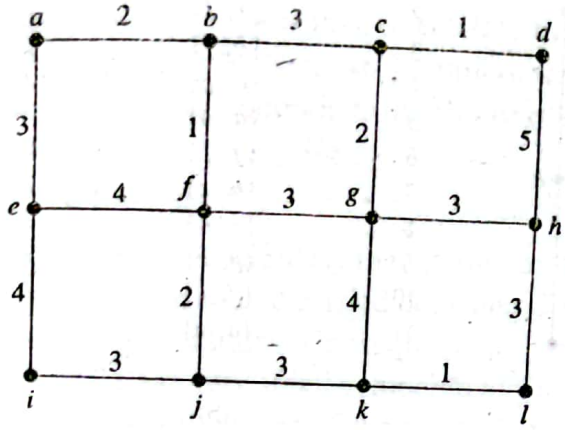
procedure *Prim*(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for $i := 1$ **to** $n - 2$
begin
  e := an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T
  $T := T$ with e added
end {T is a minimum spanning tree of G}

Note that the choice of an edge to add at a stage of the algorithm is not determined when there is more than one edge with the same weight that satisfies the appropriate criteria. We need to order the edges to make the choices deterministic. We will not worry about this in the remainder of the section. Also note that there may be more than one minimum spanning tree for a given connected weighted simple graph. (See Exercise 9.) Examples 1 and 2 illustrate how Prim's algorithm is used.

Example 1 *Use Prim's algorithm to design a minimum-cost communications network connecting all the computers represented by the graph in Figure* 1.

Links

JOSEPH BERNARD KRUSKAL (BORN 1928) Joseph Kruskal, born in New York City, attended the University of Chicago and received his Ph.D. from Princeton University in 1954. He was an instructor in mathematics at Princeton and at the University of Wisconsin, and later he was an assistant professor at the University of Michigan. In 1959 he became a member of the technical staff at Bell Laboratories, a position he continues to hold. His current research interests include statistical linguistics and psychometrics. Besides his work on minimum spanning trees, Kruskal is also known for contributions to multidimensional scaling. Kruskal discovered his algorithm for producing minimum spanning trees when he was a second-year graduate student. He was not sure his $2\frac{1}{2}$-page paper on this subject was worthy of publication, but was convinced by others to submit it.



(a)

Choice	Edge	Weight
1	{b, f}	1
2	{a, b}	2
3	{f, j}	2
4	{a, e}	3
5	{i, j}	3
6	{f, g}	3
7	{c, g}	2
8	{c, d}	1
9	{g, h}	3
10	{h, l}	3
11	{k, l}	1
	Total:	24

(b)

Figure 4 *A Minimum Spanning Tree Produced Using Prim's Algorithm.*

Solution We solve this problem by finding a minimum spanning tree in the graph in Figure 1. Prim's algorithm is carried out by choosing an initial edge of minimum weight and successively adding edges of minimum weight that are incident to a vertex in the tree and that do not form simple circuits. The edges in color in Figure 2 show a minimum spanning tree produced by Prim's algorithm, with the choice made at each step displayed. ◀

Example 2 *Use Prim's algorithm to find a minimum spanning tree in the graph shown in Figure 3.*

Solution A minimum spanning tree constructed using Prim's algorithm is shown in Figure 4. The successive edges chosen are displayed. ◀

Links The second algorithm we will discuss was discovered by Joseph Kruskal in 1956, although the basic ideas it uses were described much earlier. To carry out **Kruskal's algorithm**, choose an edge in the graph with minimum weight.

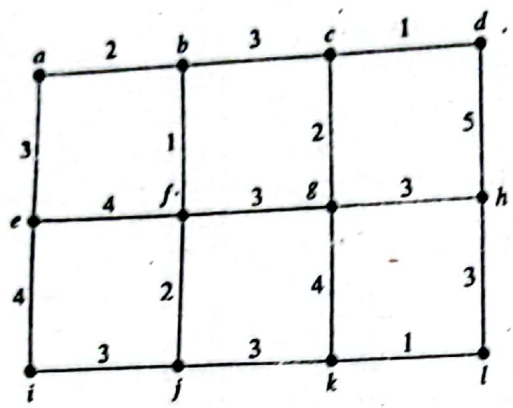
Successively add edges with minimum weight that do not form a simple circuit with those edges already chosen. Stop after $n - 1$ edges have been selected.

The proof that Kruskal's algorithm produces a minimum spanning tree for every connected weighted graph is left as an exercise at the end of this section. Pseudocode for Kruskal's algorithm is given in Algorithm 2.

ALGORITHM 2 Kruskal's Algorithm.

```
procedure Kruskal(G: weighted connected undirected graph with n vertices)
T := empty graph
for i := 1 to n - 1
begin
  e := any edge in G with smallest weight that does not form a simple circuit
    when added to T
  T := T with e added
end {T is a minimum spanning tree of G}
```

The reader should note the difference between Prim's and Kruskal's algorithms. In Prim's algorithm edges of minimum weight that are incident to a vertex already in the tree, and not forming a circuit, are chosen; whereas in Kruskal's algorithm edges of minimum weight that are not necessarily incident to a vertex already in the tree, and that do not form a circuit, are chosen. Note that as in Prim's algorithm, if the edges are not ordered, there



(a)

Choice	Edge	Weight
1	{c, d}	1
2	{k, l}	1
3	{b, f}	1
4	{c, g}	2
5	{a, b}	2
6	{f, j}	2
7	{b, c}	3
8	{j, k}	3
9	{g, h}	3
10	{i, j}	3
11	{a, e}	3
	Total:	24

(b)

Figure 5 *A Minimum Spanning Tree Produced by Kruskal's Algorithm.*

may be more than one choice for the edge to add at a stage of this procedure. Consequently, the edges need to be ordered for the procedure to be deterministic. Example 3 illustrates how Kruskal's algorithm is used.

Example 3 *Use Kruskal's algorithm to find a minimum spanning tree in the weighted graph shown in Figure 3.*

Extra Examples

Solution A minimum spanning tree and the choices of edges at each stage of Kruskal's algorithm are shown in Figure 5.

We will now prove that Prim's algorithm produces a minimum spanning tree of a connected weighted graph.

Proof: Let G be a connected weighted graph. Suppose that the successive edges chosen by Prim's algorithm are $e_1, e_2, \ldots, e_{n-1}$. Let S be the tree with $e_1, e_2, \ldots, e_{n-1}$ as its edges, and let S_k be the tree with $e_1, e_2, \ldots, e_k$ as its edges. Let T be a minimum spanning tree of G containing the edges $e_1, e_2, \ldots, e_k$, where k is the maximum integer with the property that a minimum spanning tree exists containing the first k edges chosen by Prim's algorithm. The theorem follows if we can show that $S = T$.

Suppose that $S \neq T$, so that $k < n - 1$. Consequently, T contains $e_1, e_2, \ldots, e_k$, but not e_{k+1}. Consider the graph made up of T together with e_{k+1}. Because this graph is connected and has n edges, too many edges to be a tree, it must contain a simple circuit. This simple circuit must contain e_{k+1} because there was no simple circuit in T. Furthermore, there must be an edge in the simple circuit that does not belong to S_{k+1} because S_{k+1} is a tree. By starting at an endpoint of e_{k+1} that is also an endpoint of one of the edges $e_1, \ldots, e_k$, and following the circuit until it reaches an edge not in S_{k+1}, we can find an edge e not in S_{k+1} that has an endpoint that is also an endpoint of one of the edges $e_1, e_2, \ldots, e_k$. By deleting e from T and adding e_{k+1}, we obtain a tree T'

HISTORICAL NOTE *Joseph Kruskal and Robert Prim developed their algorithms for constructing minimum spanning trees in the mid-1950s. However, they were not the first people to discover such algorithms. For example, the work of the anthropologist Jan Czekanowski, in 1909, contains many of the ideas required to find minimum spanning trees. In 1926, Otakar Borůvka described methods for constructing minimum spanning trees in work relating to the construction of electric power networks.*