Trees

Assylbek Issakhov,
Ph.D., professor

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.
- 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.

Trees

- 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.

Trees

- 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.

Trees

- In this lecture 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.

Trees

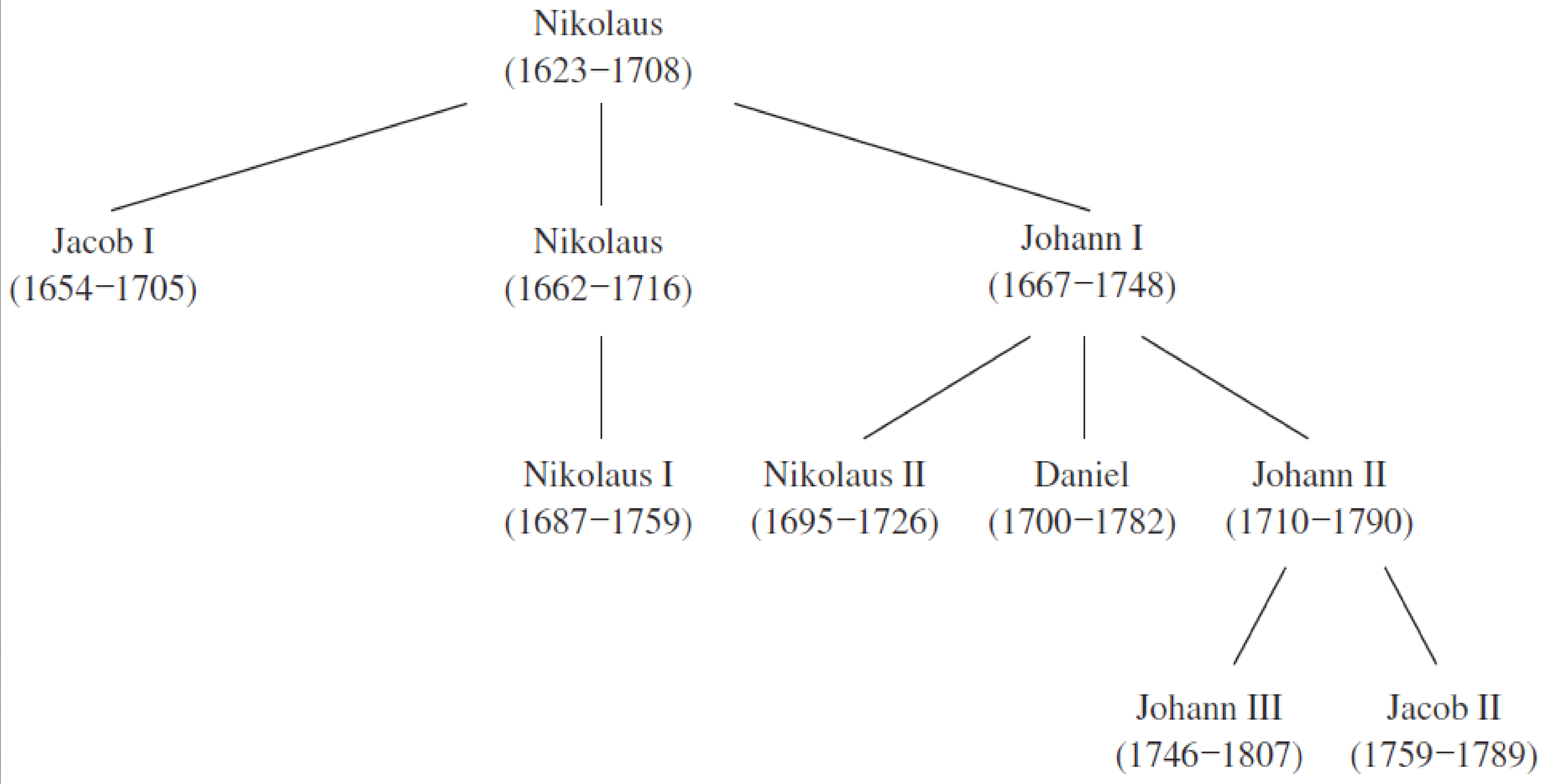


FIGURE 1 The Bernoulli Family of Mathematicians.

Trees

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

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

Trees: Example

- Which of the graphs shown in Figure 2 are trees?

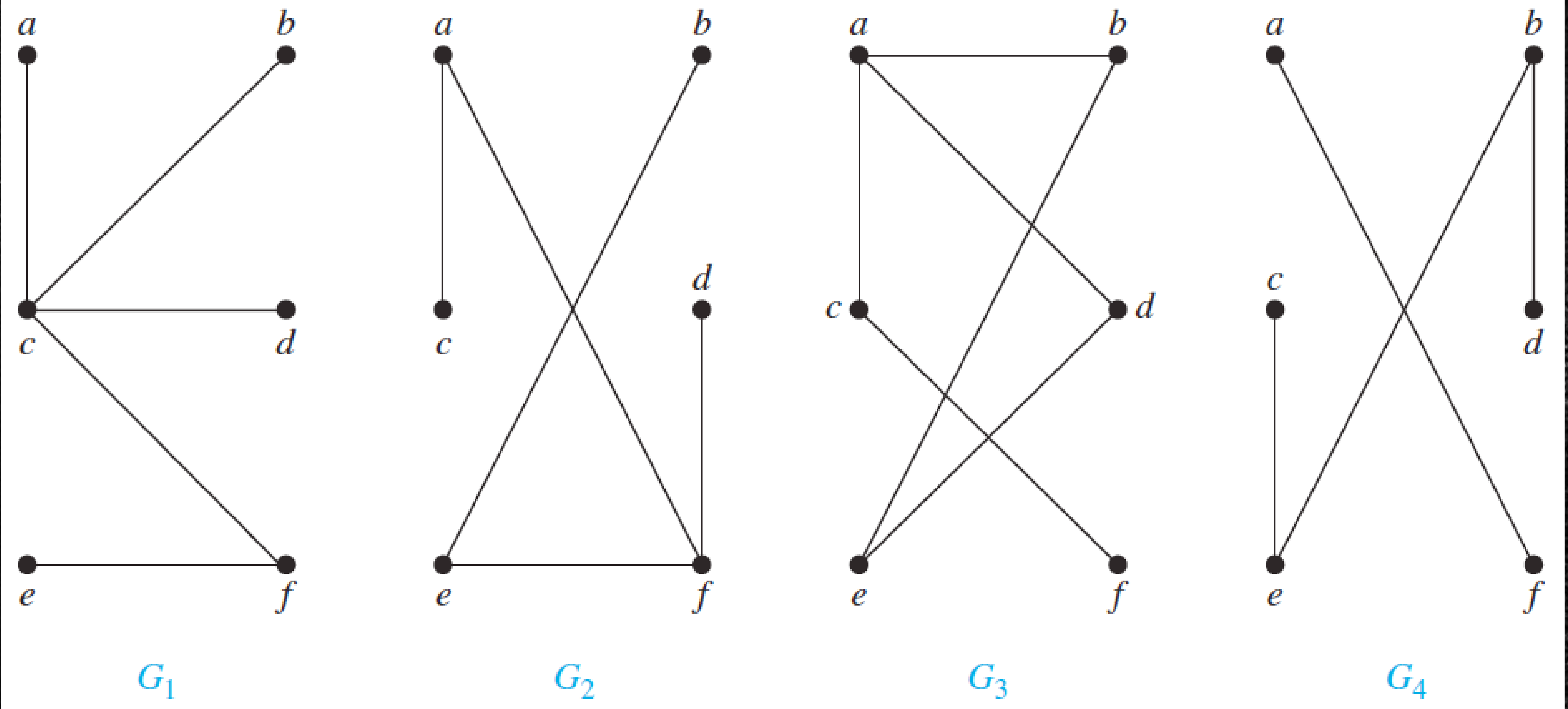


FIGURE 2 **Examples of Trees and Graphs That Are Not Trees.**

Trees: Example

- *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.

Trees

- 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.

Trees

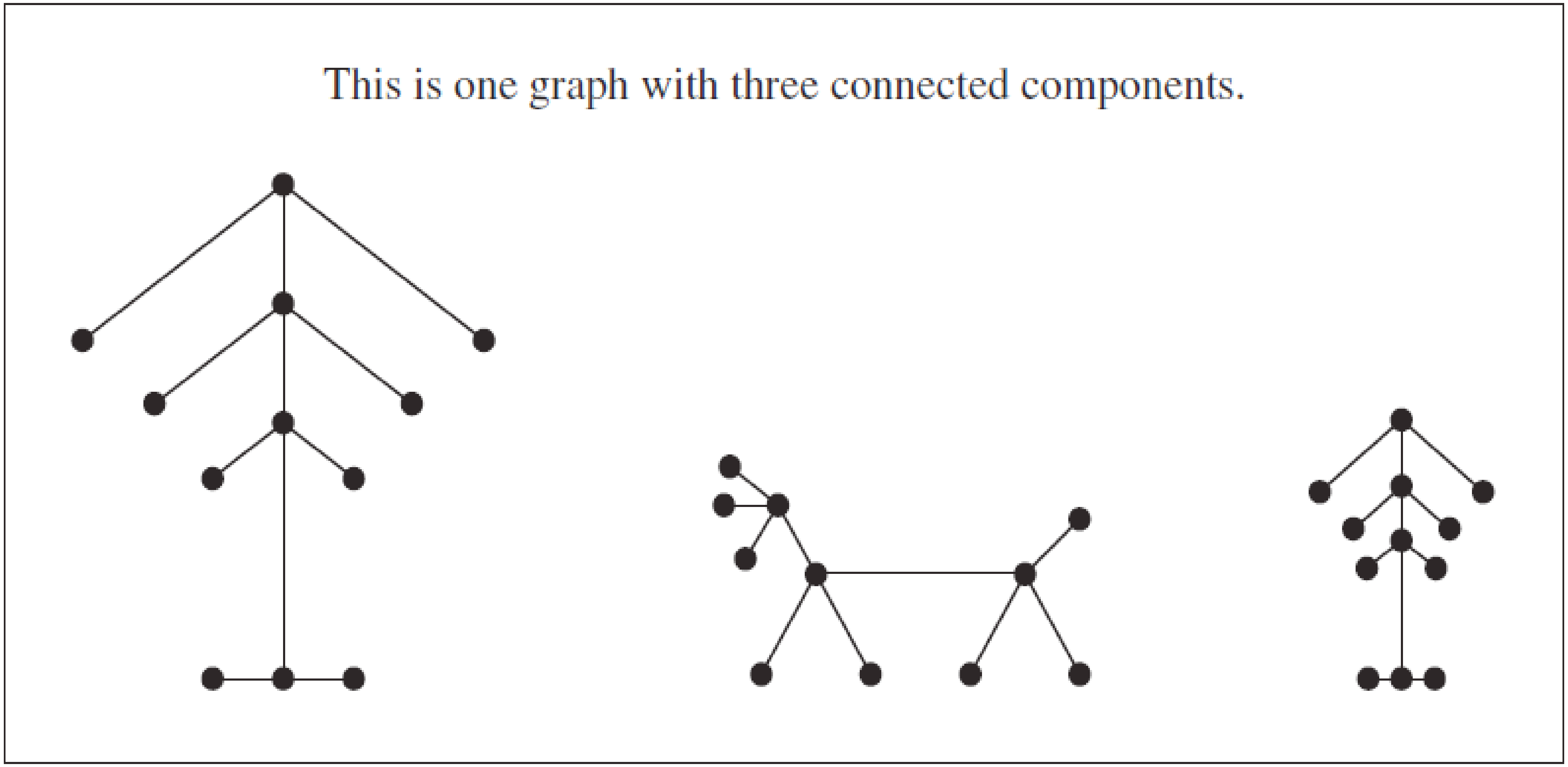


FIGURE 3 **Example of a Forest.**

Trees

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

Rooted Trees

- 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 Theorem 1), we direct each edge away from the root. Thus, a tree together with its root produces a directed graph called a **rooted tree**.

Rooted Trees

- **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. 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.

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.

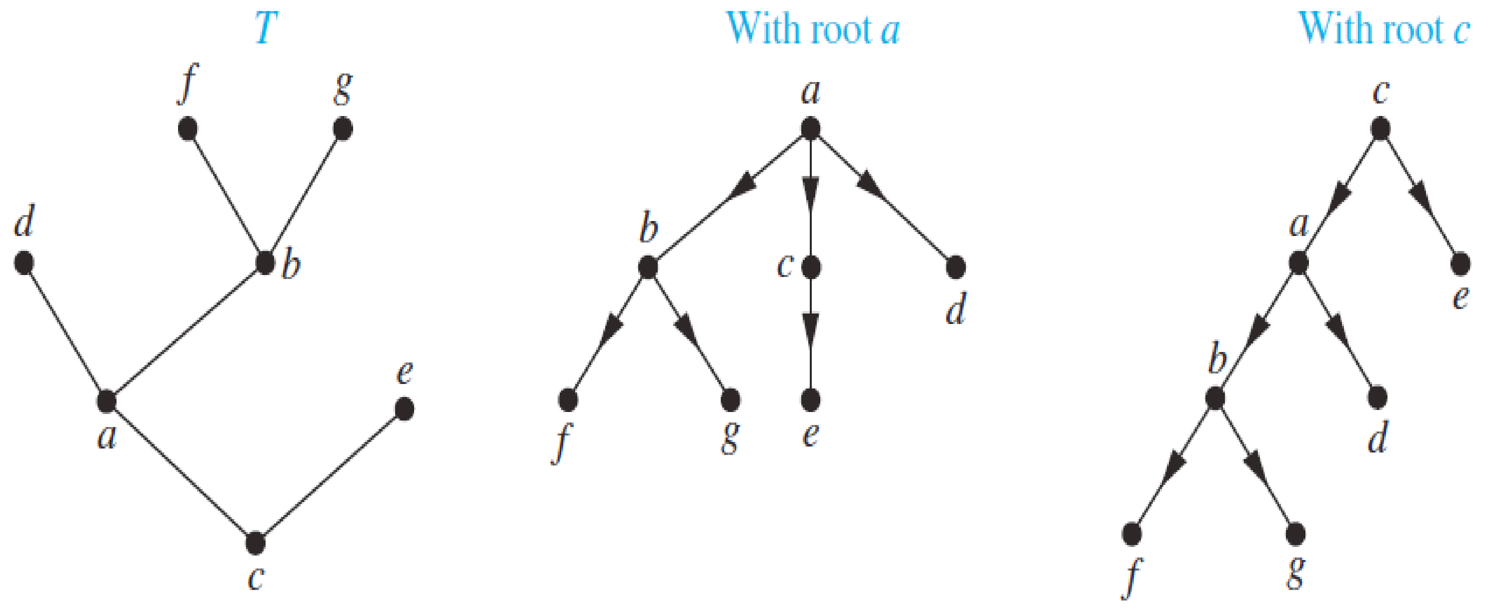


FIGURE 4 A Tree and Rooted Trees Formed by Designating Two Different Roots.

Rooted Trees

- We usually draw a rooted tree with its root at 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 v 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.

Rooted Trees

- 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 rooted tree is called a **leaf** if it has no children.

Rooted Trees

- 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.

Rooted Trees: Example

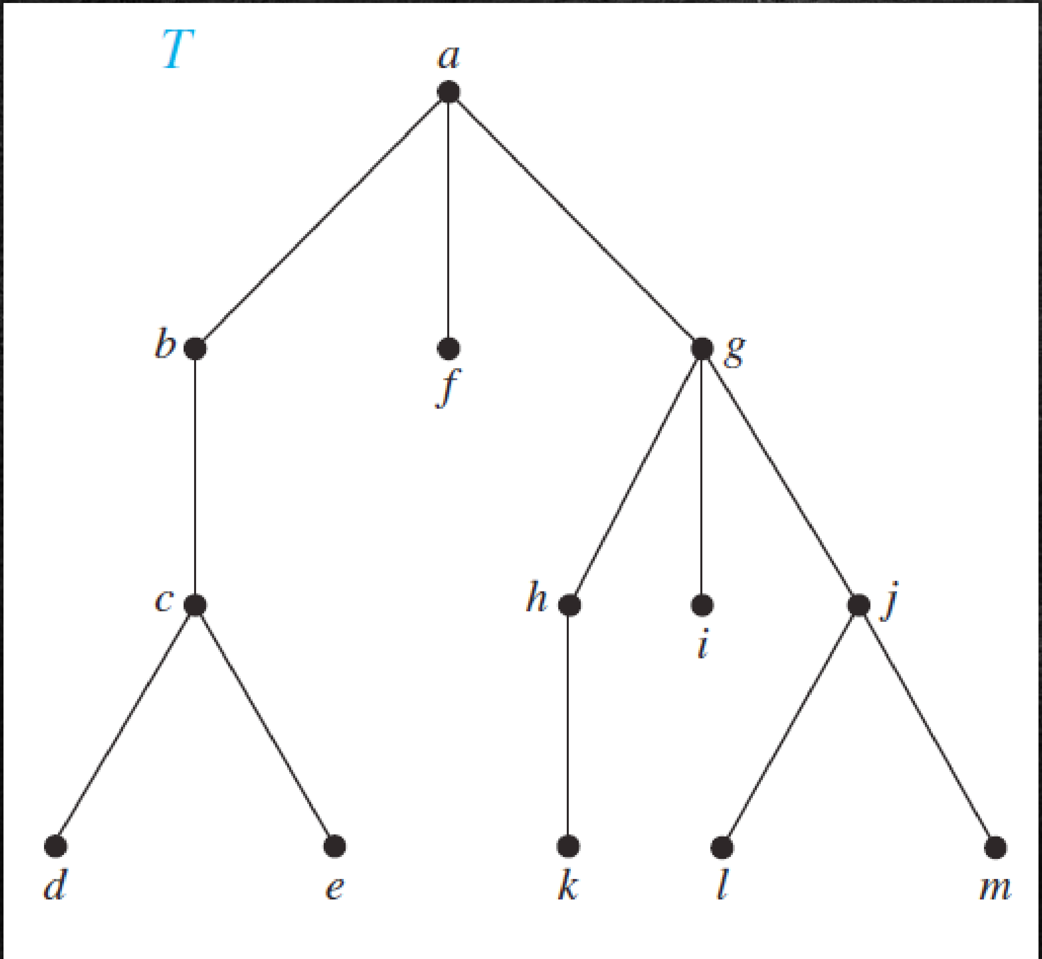


FIGURE 5 A Rooted Tree T.

- 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?

Rooted Trees: Example

- *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.

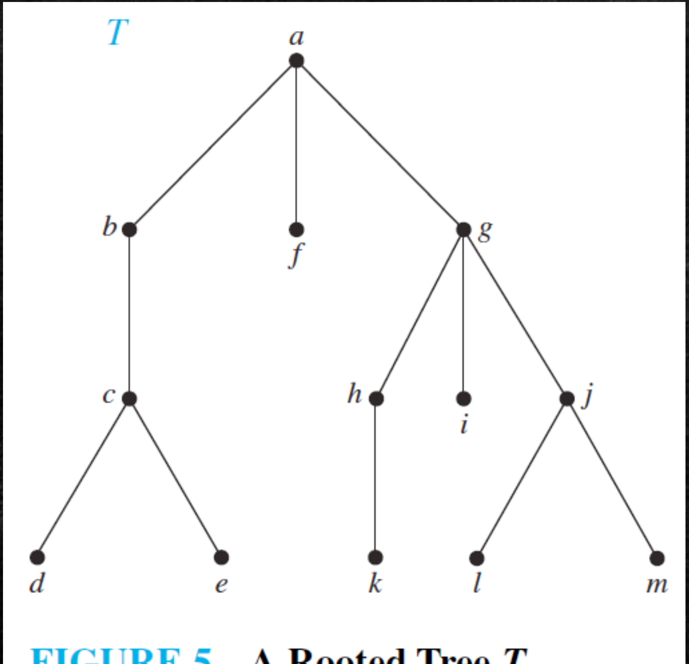


FIGURE 5 A Rooted Tree T.

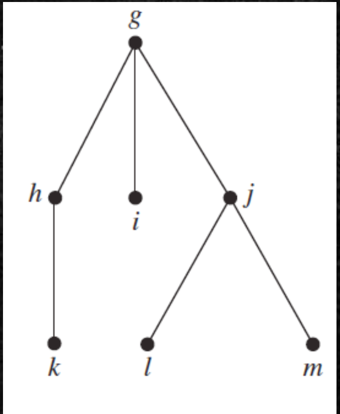


FIGURE 6 The Subtree Rooted at g.

Rooted Trees

- 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 lecture 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*.

Rooted Trees: Example

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

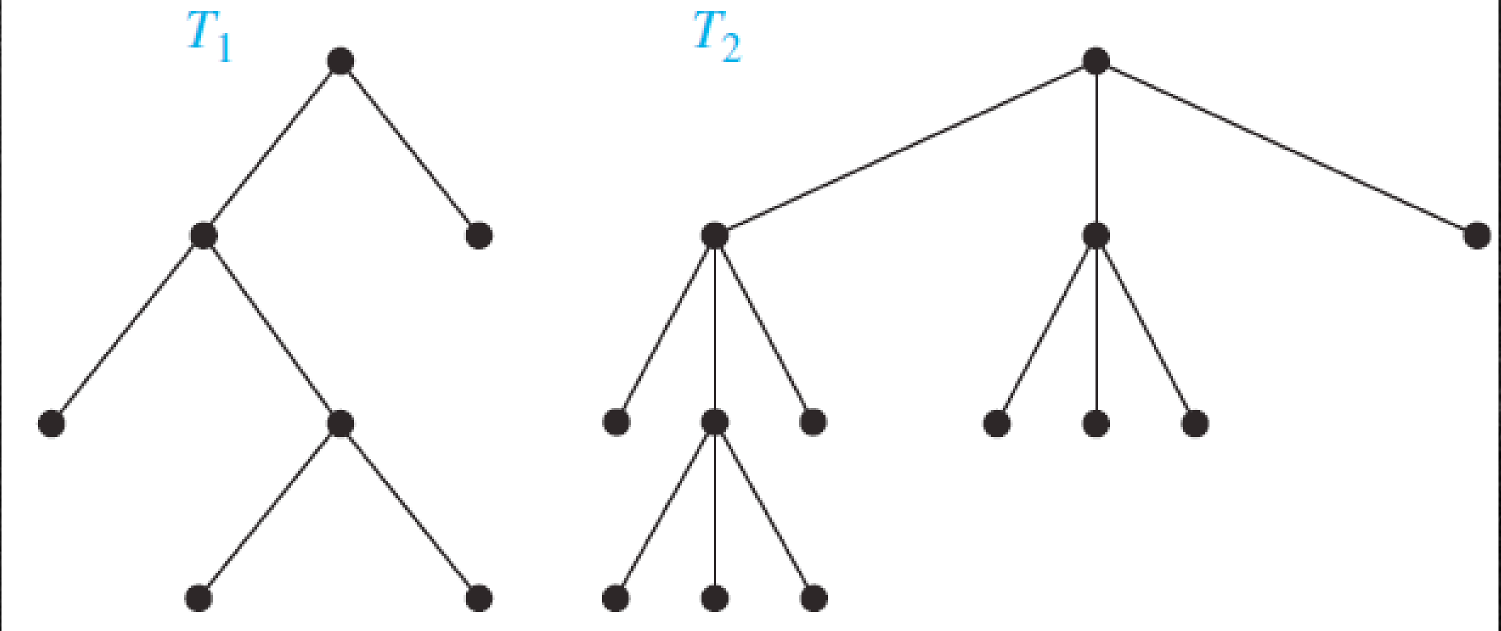
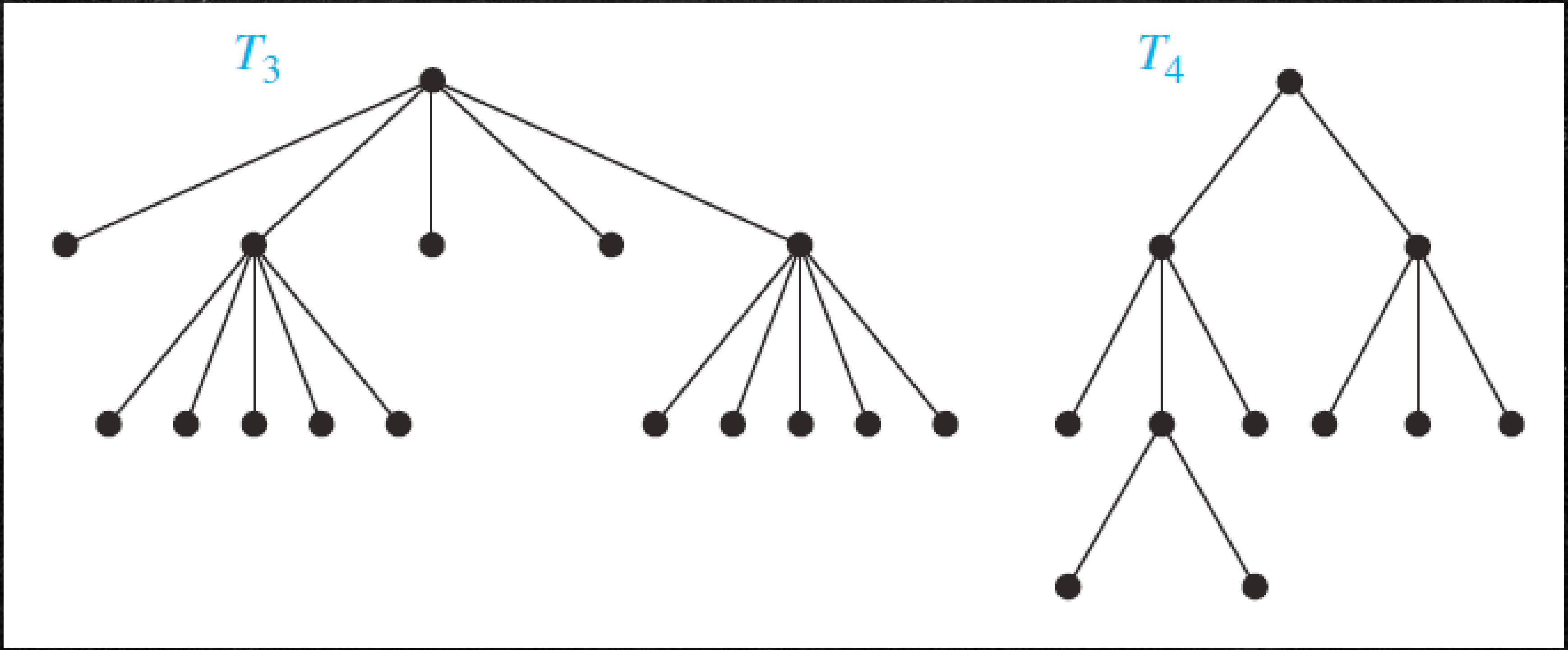


FIGURE 7 Four Rooted Trees.

Rooted Trees: Example

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



Rooted Trees: Example

- *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.

Ordered Rooted Trees

- 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.

Ordered Rooted Trees

- 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.

Ordered Rooted Trees: Example

- 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?

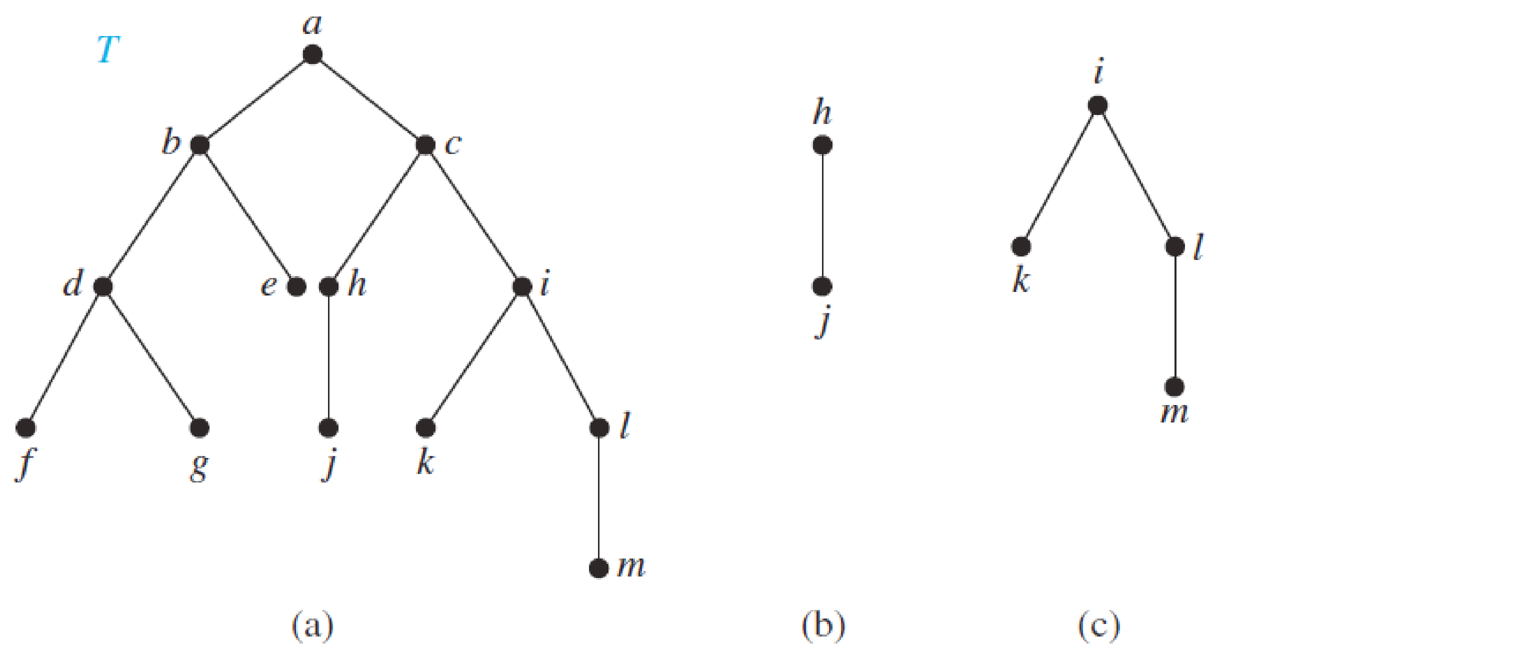


FIGURE 8 A Binary Tree T and Left and Right Subtrees of the Vertex c.

Trees as Models

- **Saturated Hydrocarbons and Trees.** Graphs can be used to represent molecules, where atoms are represented by vertices and bonds between them by edges. The English mathematician Arthur Cayley discovered trees in 1857 when he was trying to enumerate the isomers of compounds of the form C_nH_{2n+2}, which are called *saturated hydrocarbons*. In graph models of saturated hydrocarbons, each carbon atom is represented by a vertex of degree 4, and each hydrogen atom is represented by a vertex of degree 1.

Trees as Models

- There are $3n + 2$ vertices in a graph representing a compound of the form C_nH_{2n+2}. The number of edges in such a graph is half the sum of the degrees of the vertices. Hence, there are $(4n + 2n + 2)/2 = 3n + 1$ edges in this graph. Because the graph is connected and the number of edges is one less than the number of vertices, it must be a tree. The nonisomorphic trees with n vertices of degree 4 and $2n + 2$ of degree 1 represent the different isomers of C_nH_{2n+2}.

Trees as Models

- For instance, when $n = 4$, there are exactly two nonisomorphic trees of this type. Hence, there are exactly two different isomers of C_4H_{10}. Their structures are displayed in Figure 9. These two isomers are called butane and isobutane.

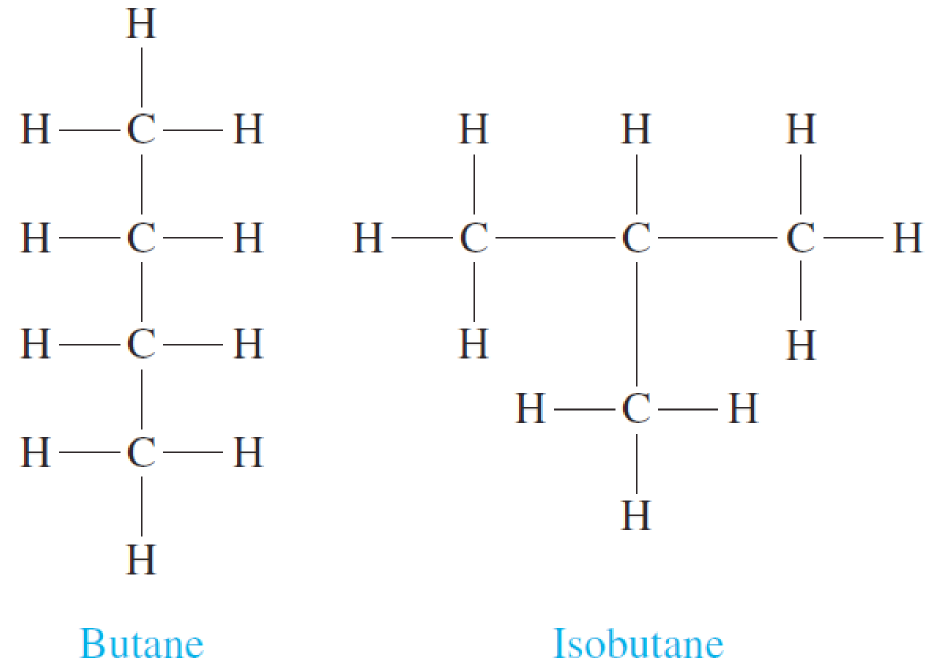


FIGURE 9 The Two Isomers of Butane.

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.
- Recall that a tree is a connected undirected graph with no simple circuits. So, when G is an undirected graph with n vertices, Theorem 2 tells us that the two conditions *(i)* G is connected and *(ii)* G has no simple circuits, imply *(iii)* G has $n - 1$ edges.

Counting vertices in full m-ary trees

- 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 $n = m \cdot i + 1$ vertices.

Counting vertices in full m-ary trees

- 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.

Counting vertices in full m-ary trees

- **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.

Counting vertices in full m-ary trees: Example

- 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?

Counting vertices in full m-ary trees: Example

- *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.

Balanced m-ary Trees

- 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 v 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.

Balanced m-ary Trees: Example

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

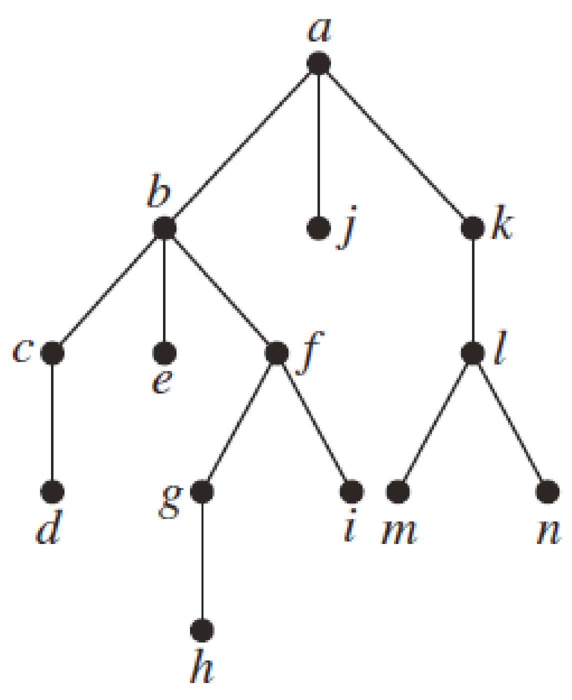


FIGURE 13 A Rooted Tree.

Balanced m-ary Trees: Example

- *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 $h - 1$.

Balanced m-ary Trees

- Which of the rooted trees shown in Figure 14 are balanced?

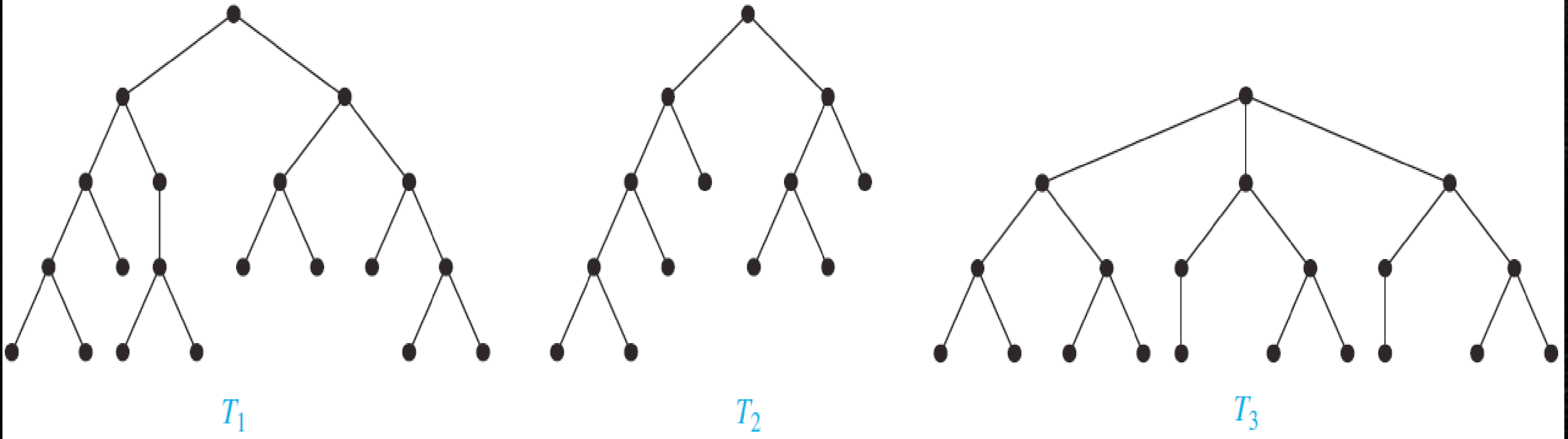


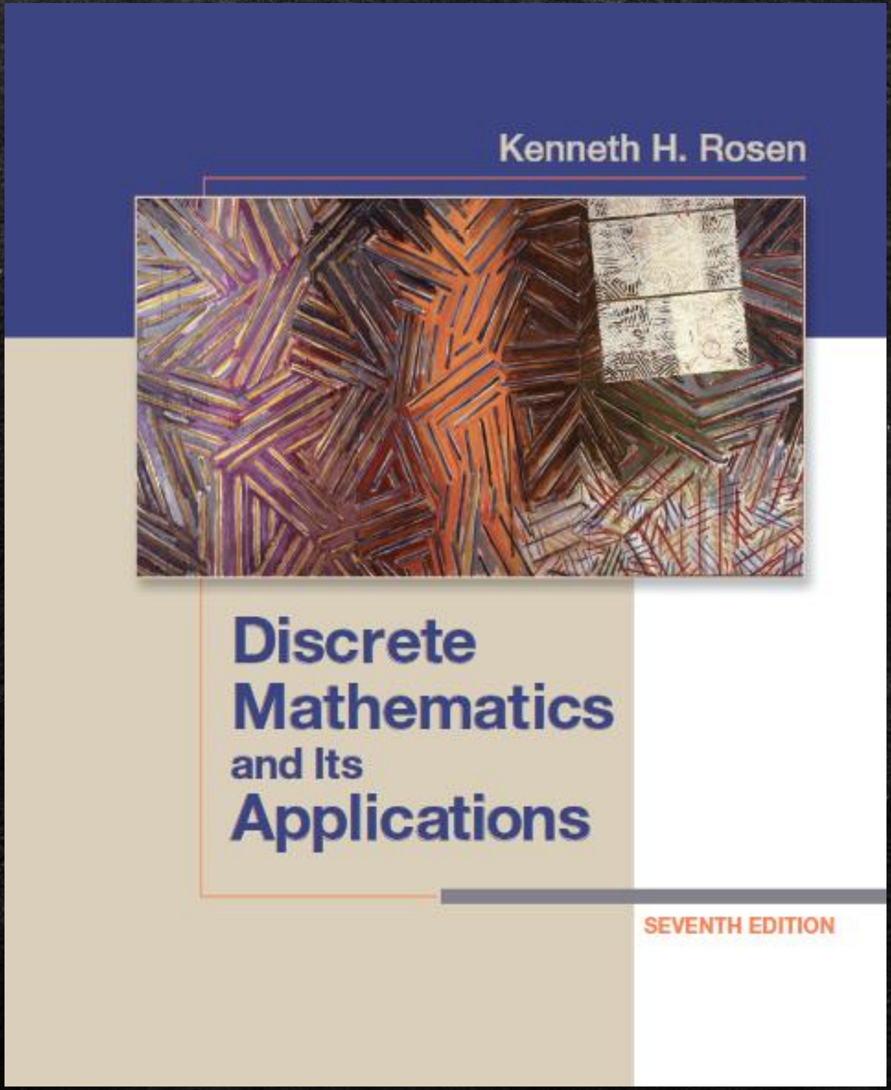
FIGURE 14 Some Rooted Trees.

A bound for the number of leaves in an m-ary tree

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

- **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.)

HOMEWORK: Exercises 2, 4, 6, 8, 10, 18, 20, 22, 40 on pp. 755-757



Tree Traversal

- Ordered rooted trees are often used to store information. We need procedures for visiting each vertex of an ordered rooted tree to access data. We will describe several important algorithms for visiting all the vertices of an ordered rooted tree. Ordered rooted trees can also be used to represent various types of expressions, such as arithmetic expressions involving 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.

Universal Address Systems

- 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 v at level n with label A, label its k_v children, as they are drawn from left to right, with $A.1, A.2, \ldots, A.k_v$.

Universal Address Systems

- Following this procedure, a vertex v at level n, for $n \geq 1$, is labeled $x_1.x_2.\ldots.x_n$, where the unique path from the root to v 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.

Universal Address Systems

- 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 $i = 1, 2, \ldots, n$.

Universal Address Systems: Example

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

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

Universal Address Systems: Example

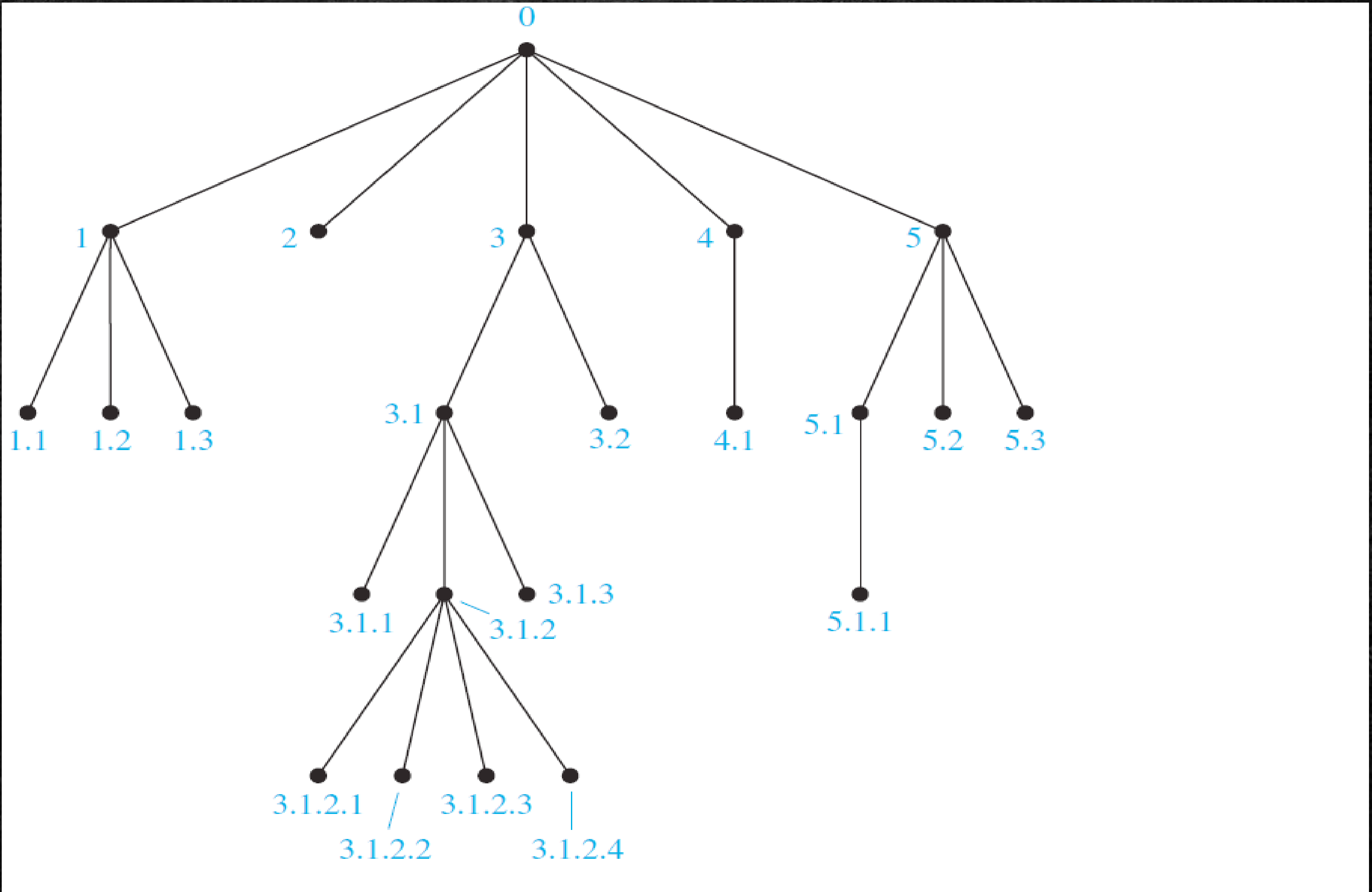


FIGURE 1 **The Universal Address System of an Ordered Rooted Tree.**

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.

Traversal Algorithms

- **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.

Traversal Algorithms: Example

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

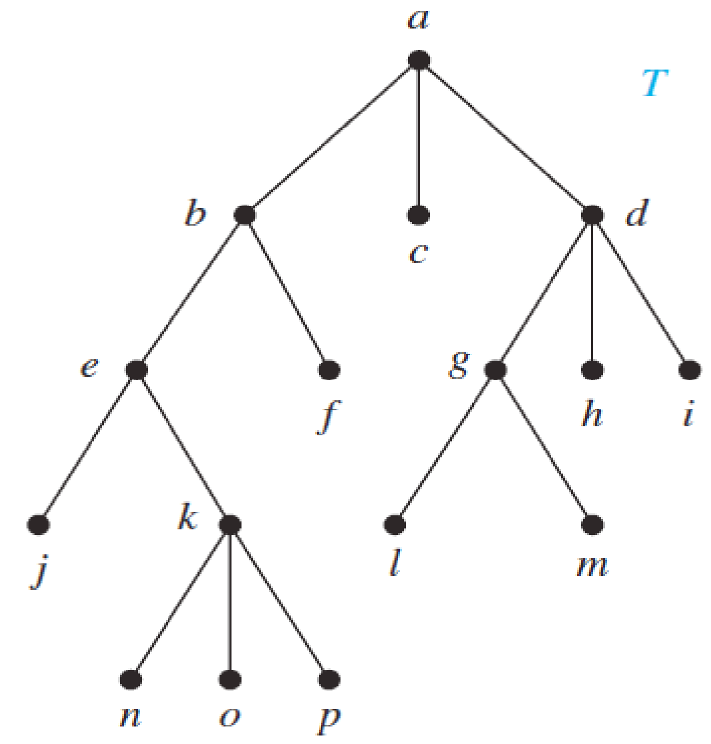
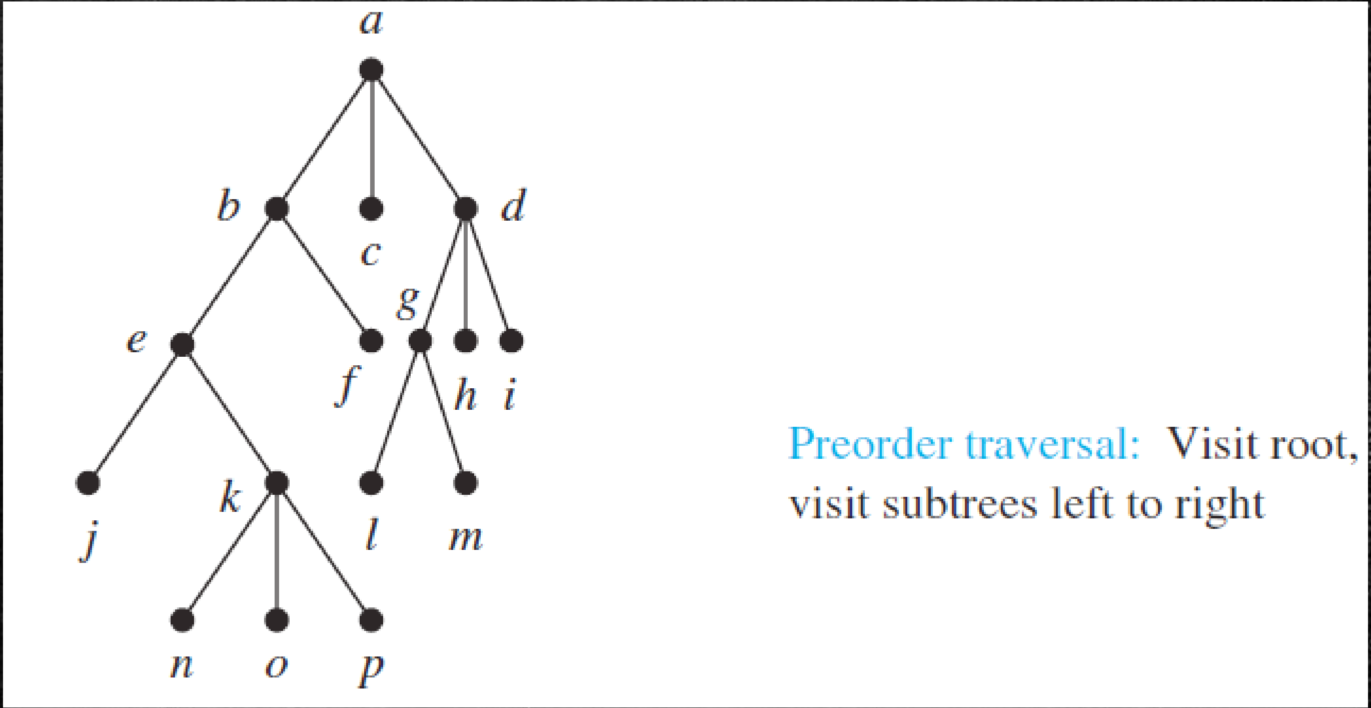


FIGURE 3 **The Ordered Rooted Tree *T*.**

Traversal Algorithms: Example

- *Solution:* The steps of the preorder traversal of T are shown in Figure 4.


a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
Preorder traversal: Visit root, visit subtrees left to right

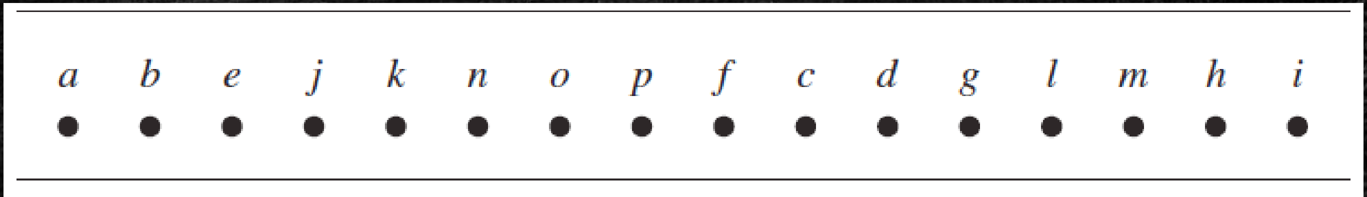


FIGURE 4 The Preorder Traversal of T.

Traversal Algorithms

- **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.

Traversal Algorithms: Example

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

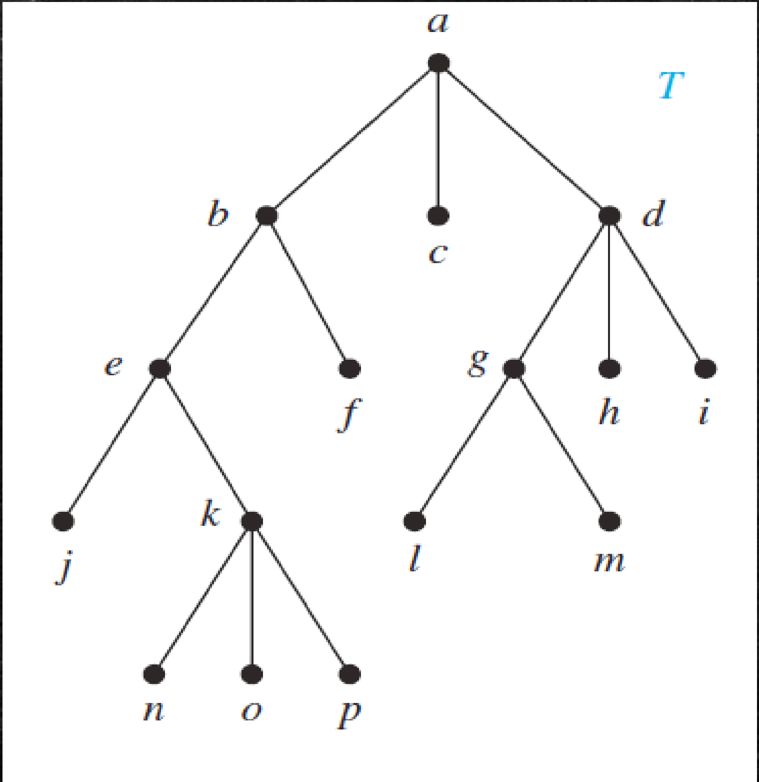
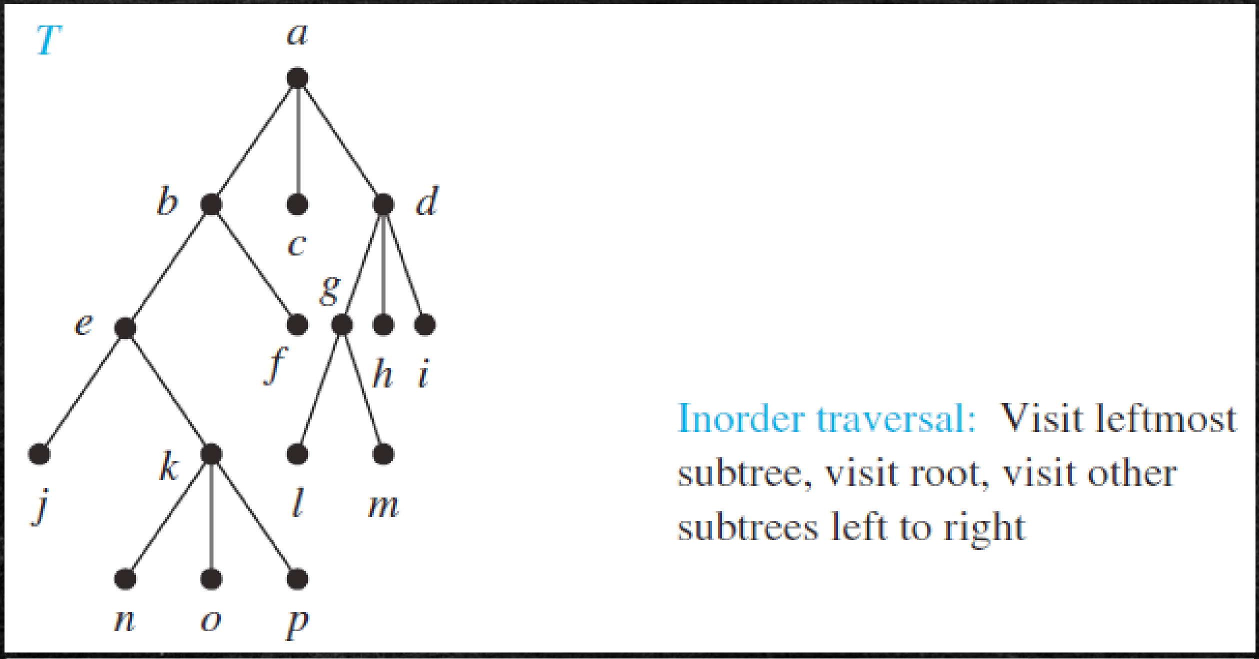


FIGURE 3 The Ordered Rooted Tree T.

Traversal Algorithms: Example

- *Solution:* The steps of the inorder traversal of the ordered rooted tree T are shown in Figure 6.



Inorder traversal: Visit leftmost subtree, visit root, visit other subtrees left to right

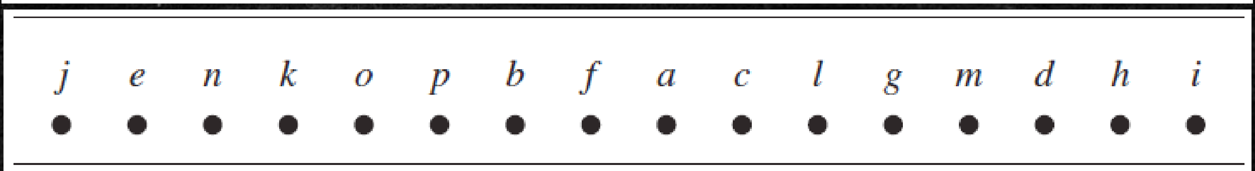


FIGURE 6 The Inorder Traversal of T.

Traversal Algorithms

- **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.

Traversal Algorithms: Example

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

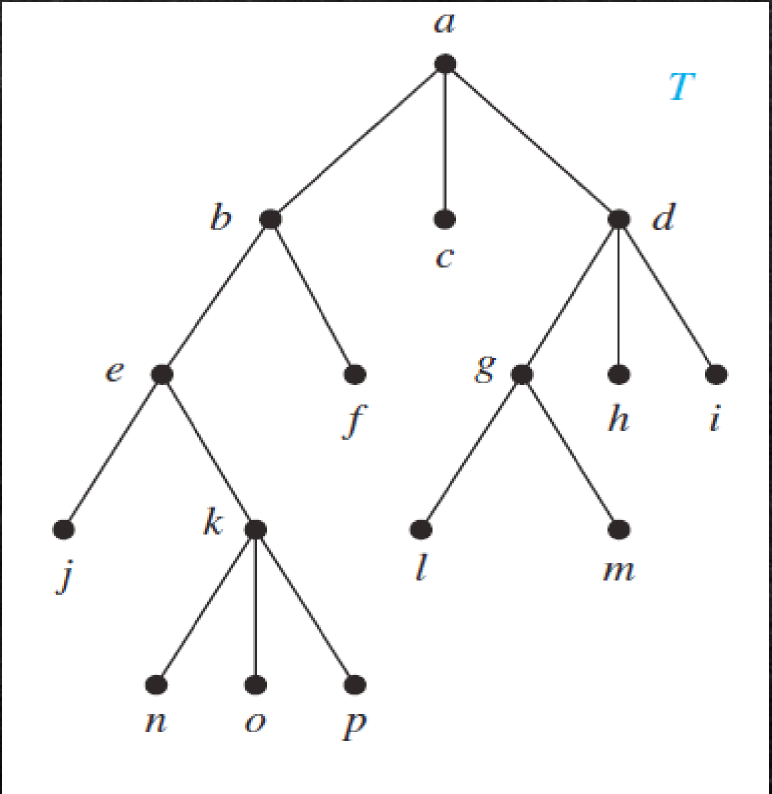


FIGURE 3 The Ordered Rooted Tree T.

Traversal Algorithms: Example

- *Solution:* The steps of the postorder traversal of the ordered rooted tree T are shown in Figure 8.

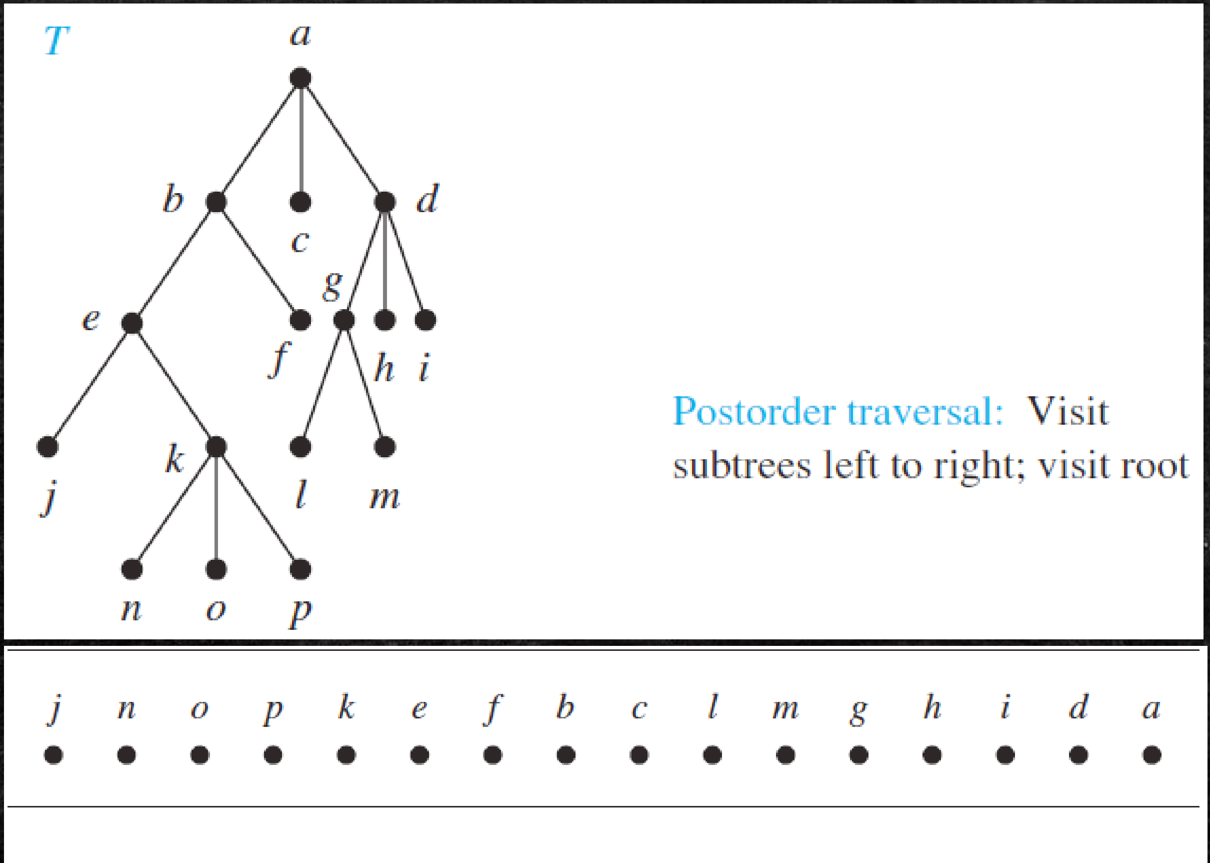

T
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
Postorder traversal: Visit subtrees left to right; visit root
j n o p k e f b c l m g h i d a

FIGURE 8 The Postorder Traversal of T.

Graph representation

- 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 $\uparrow$ (exponentiation). We will use parentheses to indicate the order of the operations.

Graph representation

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

Graph representation

- *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 $\big((x + y) \uparrow 2\big) + ((x - 4)/3)$. These steps are shown in Figure 10

Graph representation

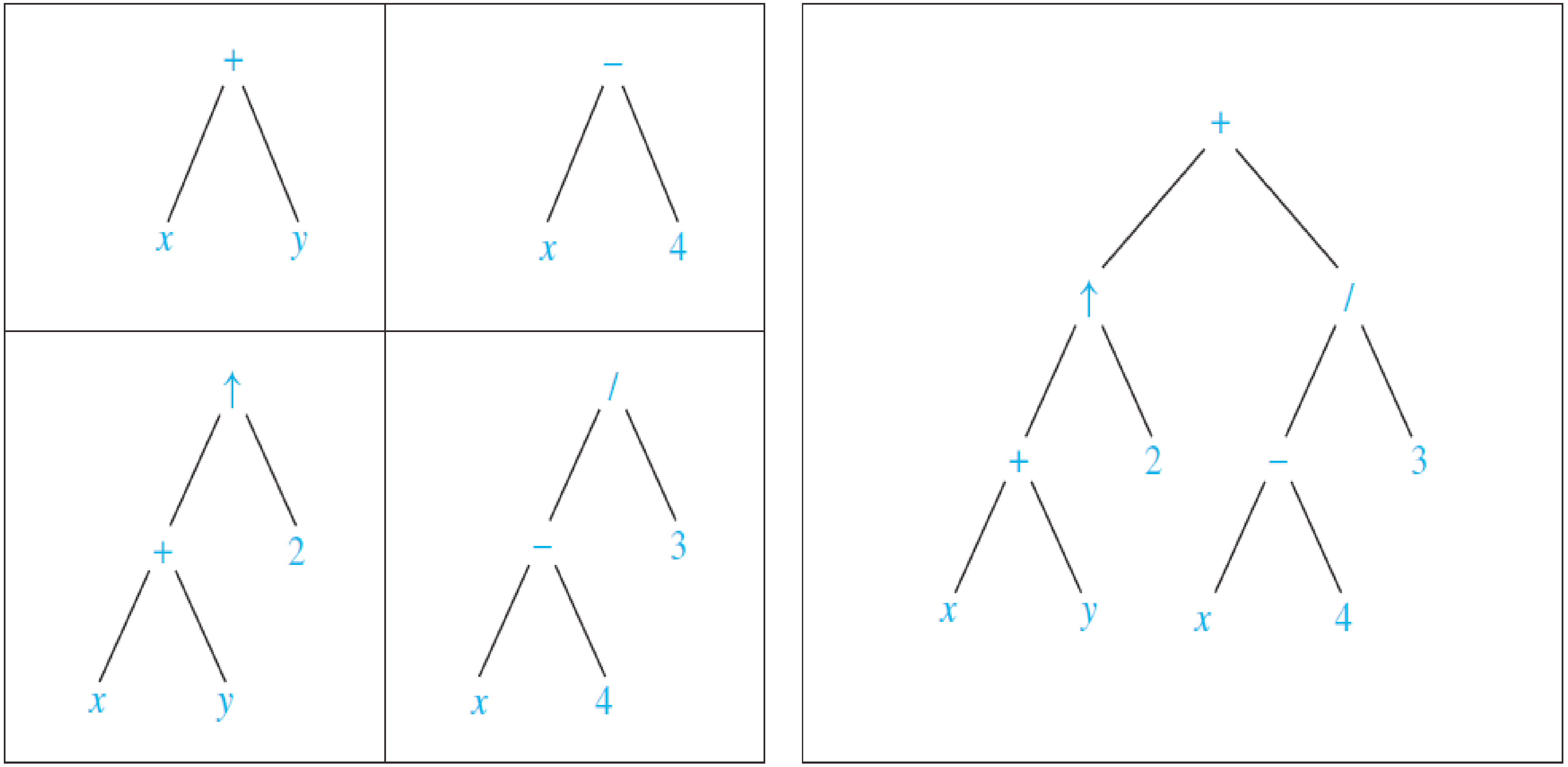
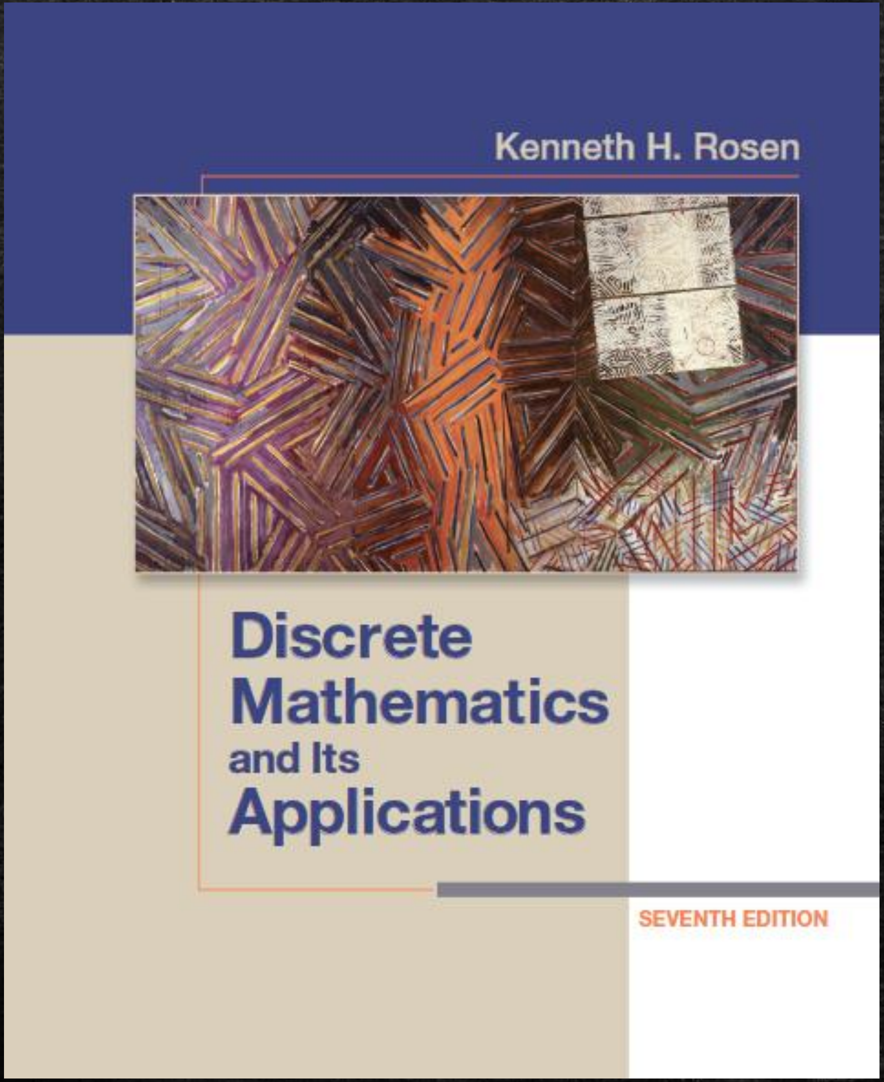


FIGURE 10 A Binary Tree Representing $((x + y) \uparrow 2) + ((x - 4)/3)$.

HOMEWORK: Exercises 2, 4, 8, 10, 14 on p. 783



Spanning Trees

- Consider the system of roads in Maine represented by the simple graph shown in Figure 1(a). The only way the roads can be kept open in the winter is by frequently plowing them. The highway department wants to plow the fewest roads so that there will always be cleared roads connecting any two towns. How can this be done?

Spanning Trees

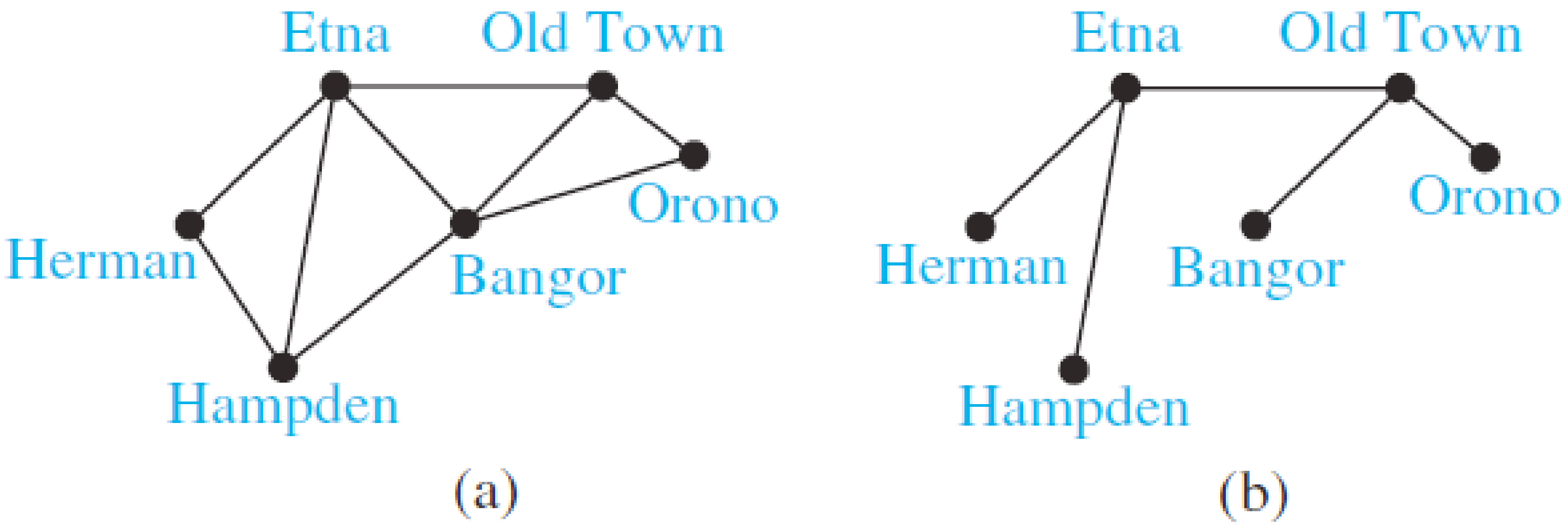


FIGURE 1 **(a) A Road System and (b) a Set of Roads to Plow.**

Spanning Trees

- At least five roads must be plowed to ensure that there is a path between any two towns. Figure 1(b) shows one such set of roads. Note that the subgraph representing these roads is a tree, because it is connected and contains six vertices and five edges.
- This problem was solved with a connected subgraph with the minimum number of edges containing all vertices of the original simple graph. Such a graph must be a tree.

Spanning Trees

- **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.

Spanning Trees: Example

- Find a spanning tree of the simple graph G shown in Figure 2.

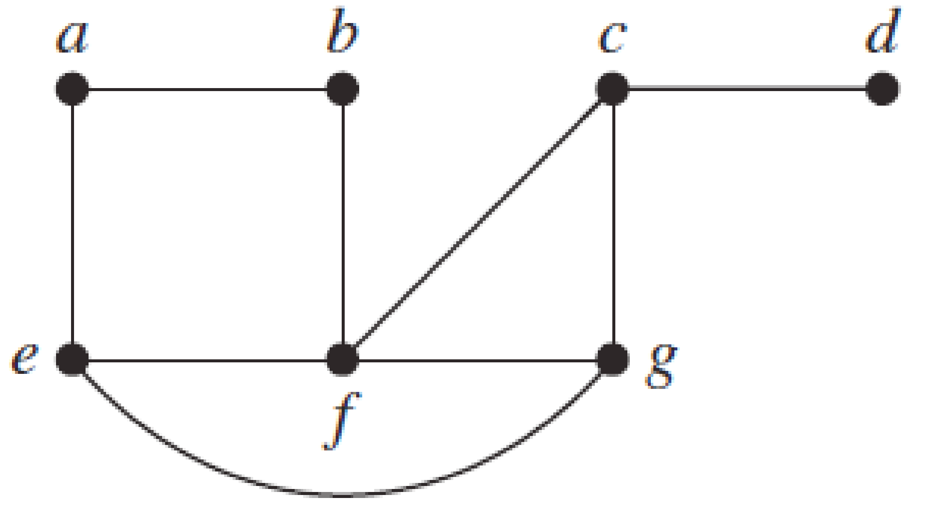


FIGURE 2 The Simple Graph G.

Spanning Trees: Example

- *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.

Spanning Trees: Example

- 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.

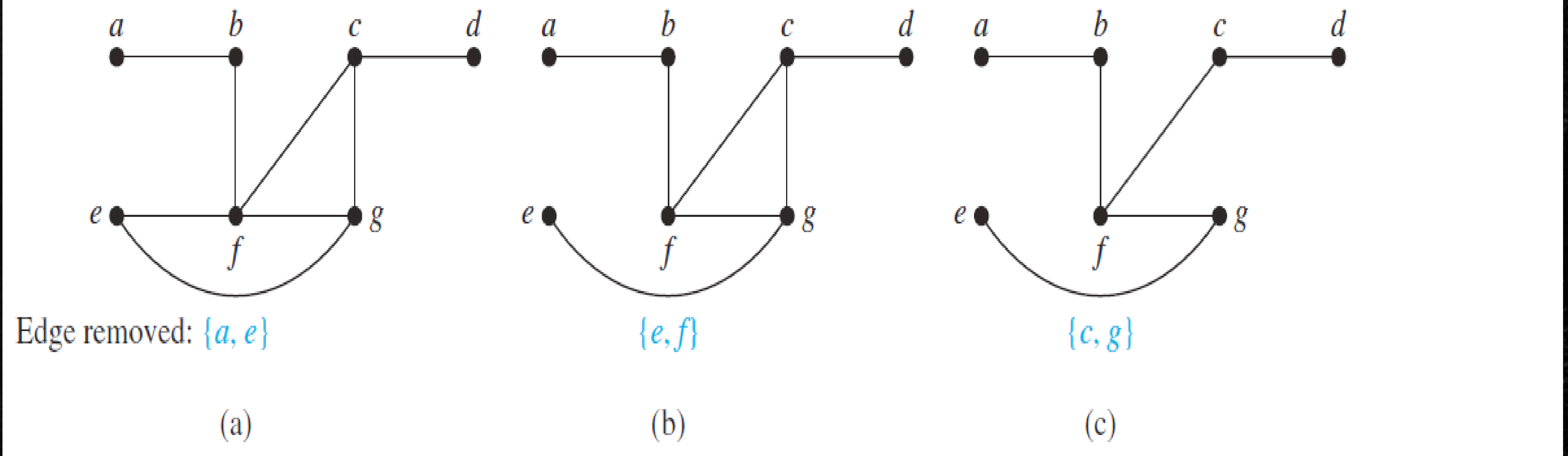


FIGURE 3 Producing a Spanning Tree for G by Removing Edges That Form Simple Circuits.

Spanning Trees: Example

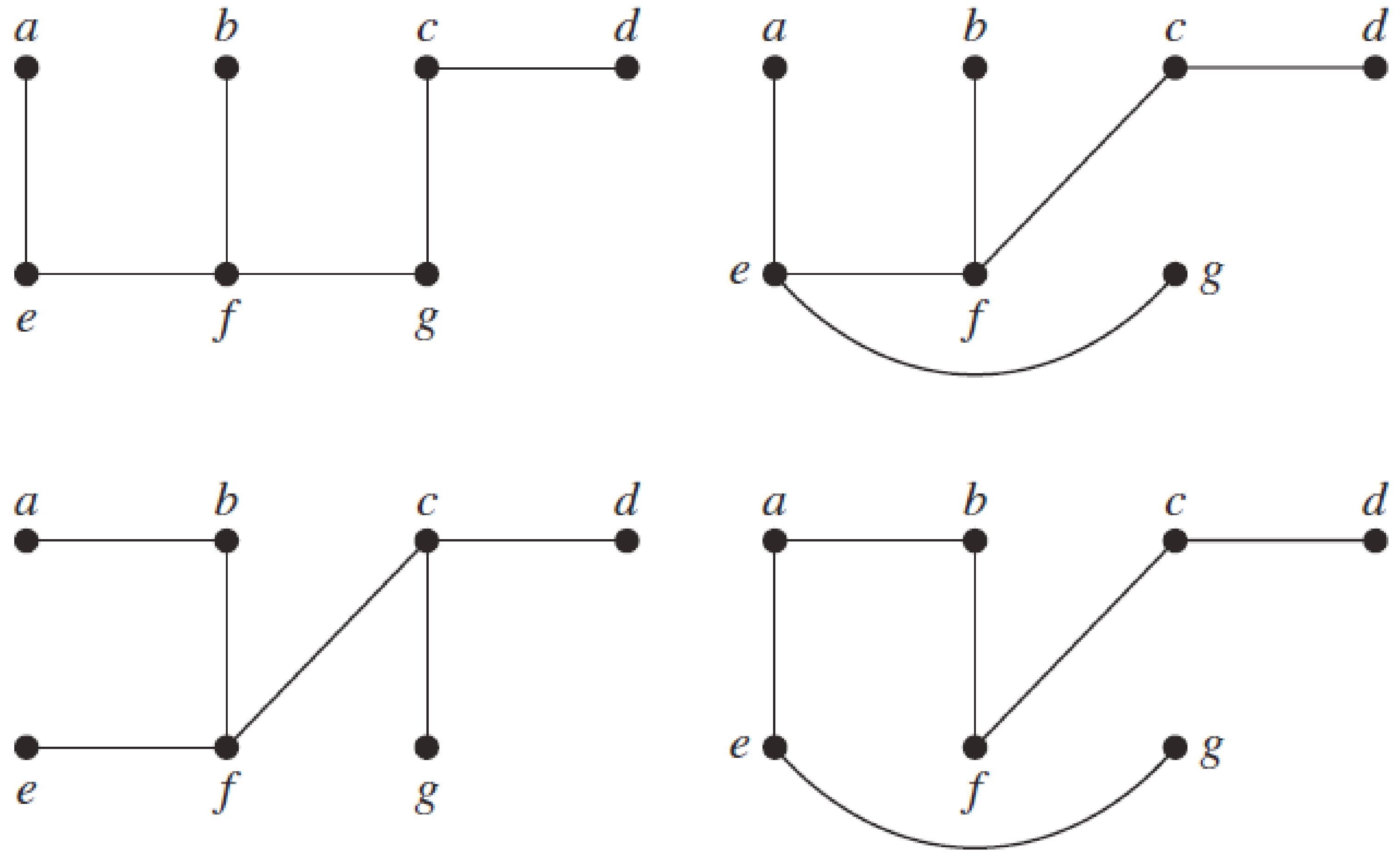


FIGURE 4 Spanning Trees of *G*.

Spanning Trees

- **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.

Spanning Trees

- ***Proof (cont.):*** The resulting subgraph has one fewer edge but still contains all the vertices of G and is connected. This subgraph is still connected because when two vertices are connected by a path containing the removed edge, they are connected by a path not containing this edge. We can construct such a path by inserting into the original path, at the point where the removed edge once was, the simple circuit with this edge removed.

Spanning Trees

- ***Proof (cont.):*** If this subgraph is not a tree, it has a simple circuit; so as before, remove an edge that is in a simple circuit. Repeat this process until no simple circuits remain. This is possible because there are only a finite number of edges in the graph. The process terminates when no simple circuits remain. A tree is produced because the graph stays connected as edges are removed. This tree is a spanning tree because it contains every vertex of G.

Depth-First Search

- The proof of Theorem 1 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.

Depth-First Search

- We can build a spanning tree for a connected simple graph using **depth-first search**. We will form a 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.

Depth-First Search

- 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, form a 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.

Depth-First Search

- 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.

Depth-First Search

- 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.

Depth-First Search: Example

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

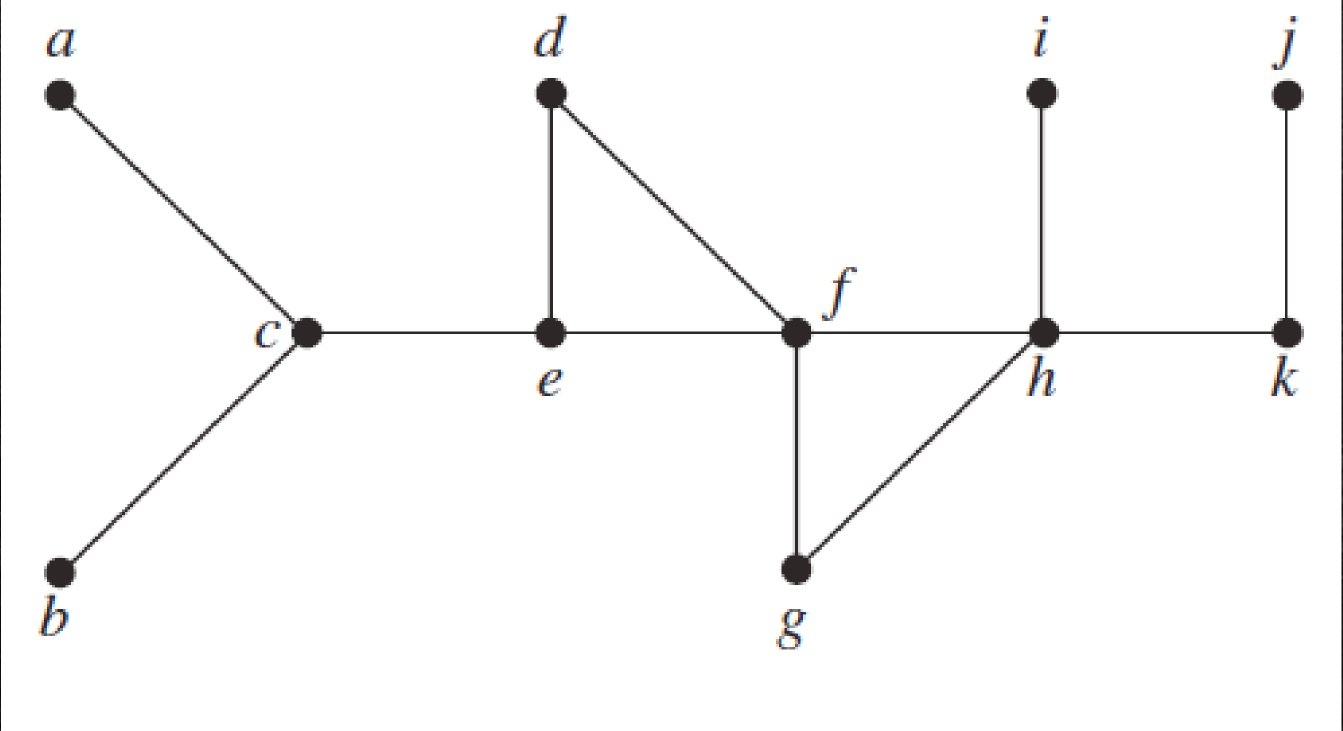


FIGURE 6 The Graph G.

Depth-First Search: Example

- *Solution:* The steps used by depth-first search to produce a spanning tree of G are shown in Figure 7.

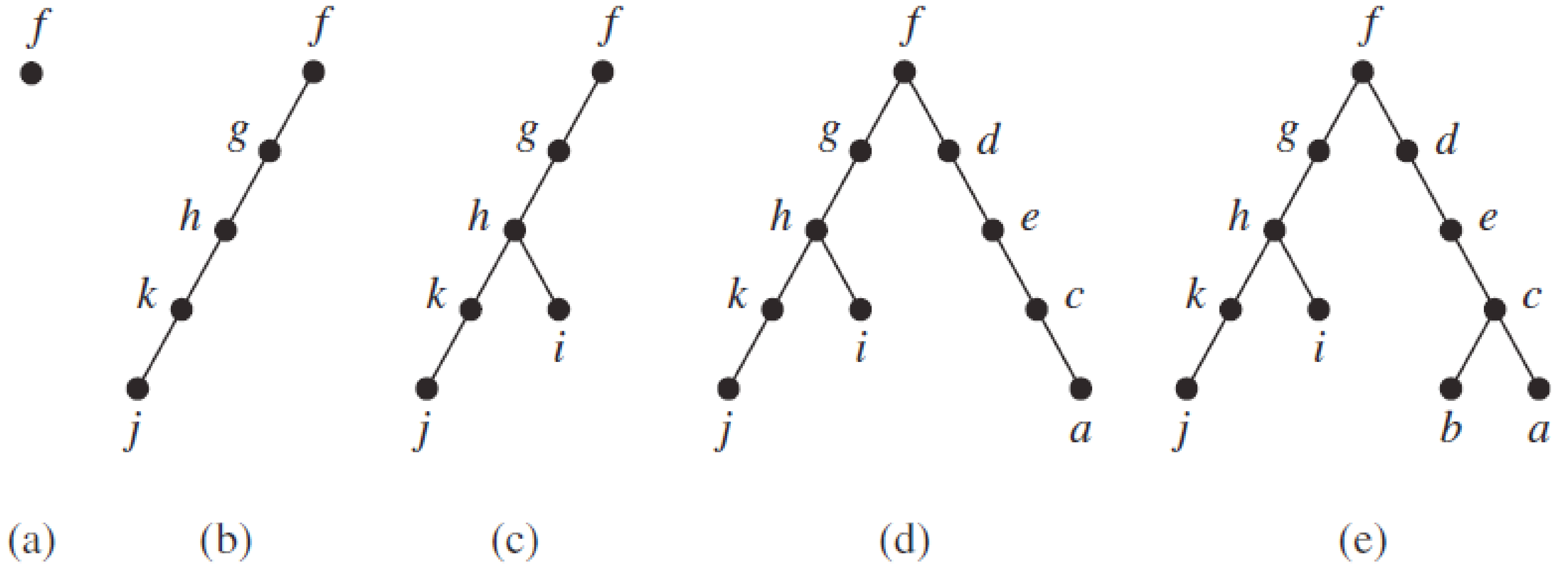


FIGURE 7 **Depth-First Search of G.**

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.

Breadth-First Search

- 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.

Breadth-First Search: Example

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

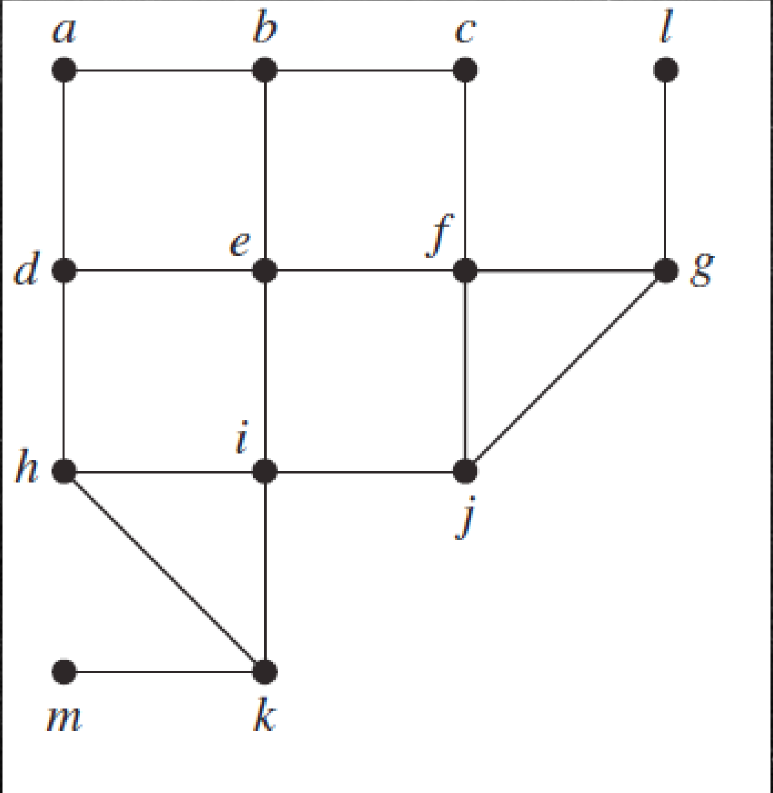


FIGURE 9 A Graph G.

Breadth-First Search: Example

- *Solution:* The steps of the breadth-first search procedure are shown in Figure 10.

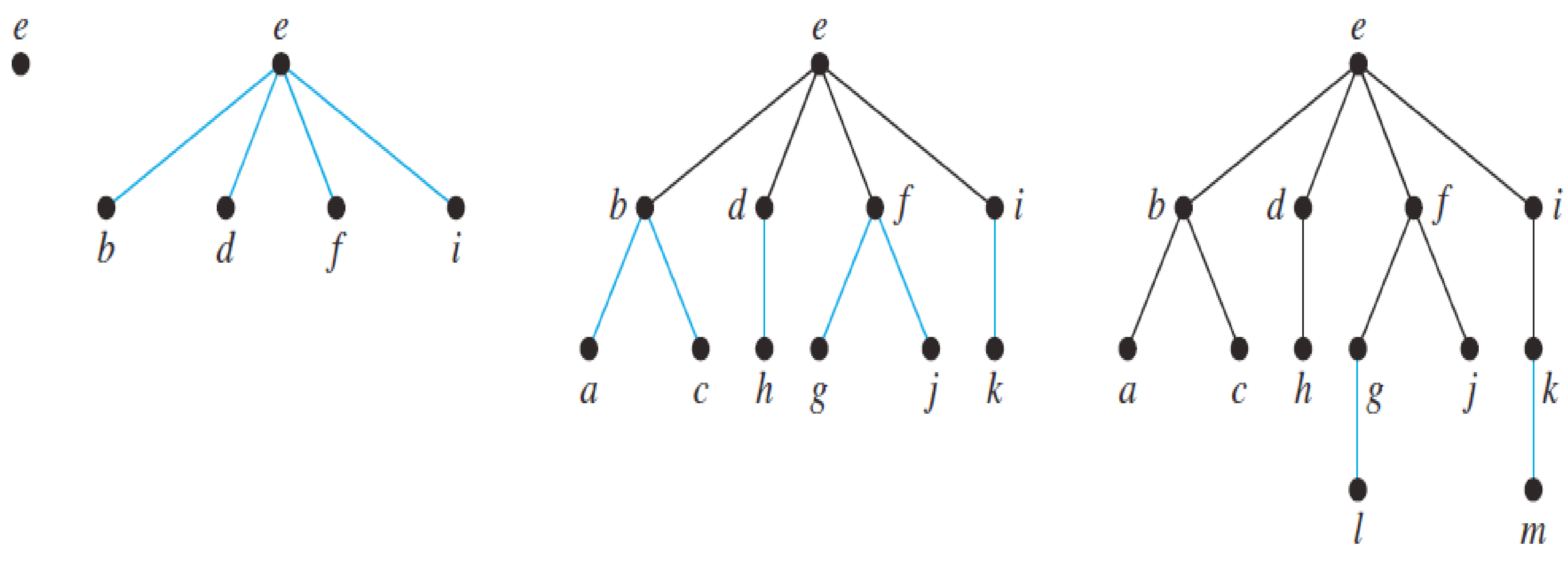


FIGURE 10 Breadth-First Search of G.

Backtracking Applications

- **The n-Queens Problem.** The n-queens problem asks how n queens can be placed on an $n \times n$ chessboard so that no two queens can attack one another. How can backtracking be used to solve the n-queens problem?
- *Solution:* To solve this problem we must find n positions on an $n \times n$ chessboard so that no two of these positions are in the same row, same column, or in the same diagonal [a diagonal consists of all positions (i, j) with $i + j = m$ for some m, or $i - j = m$ for some m].

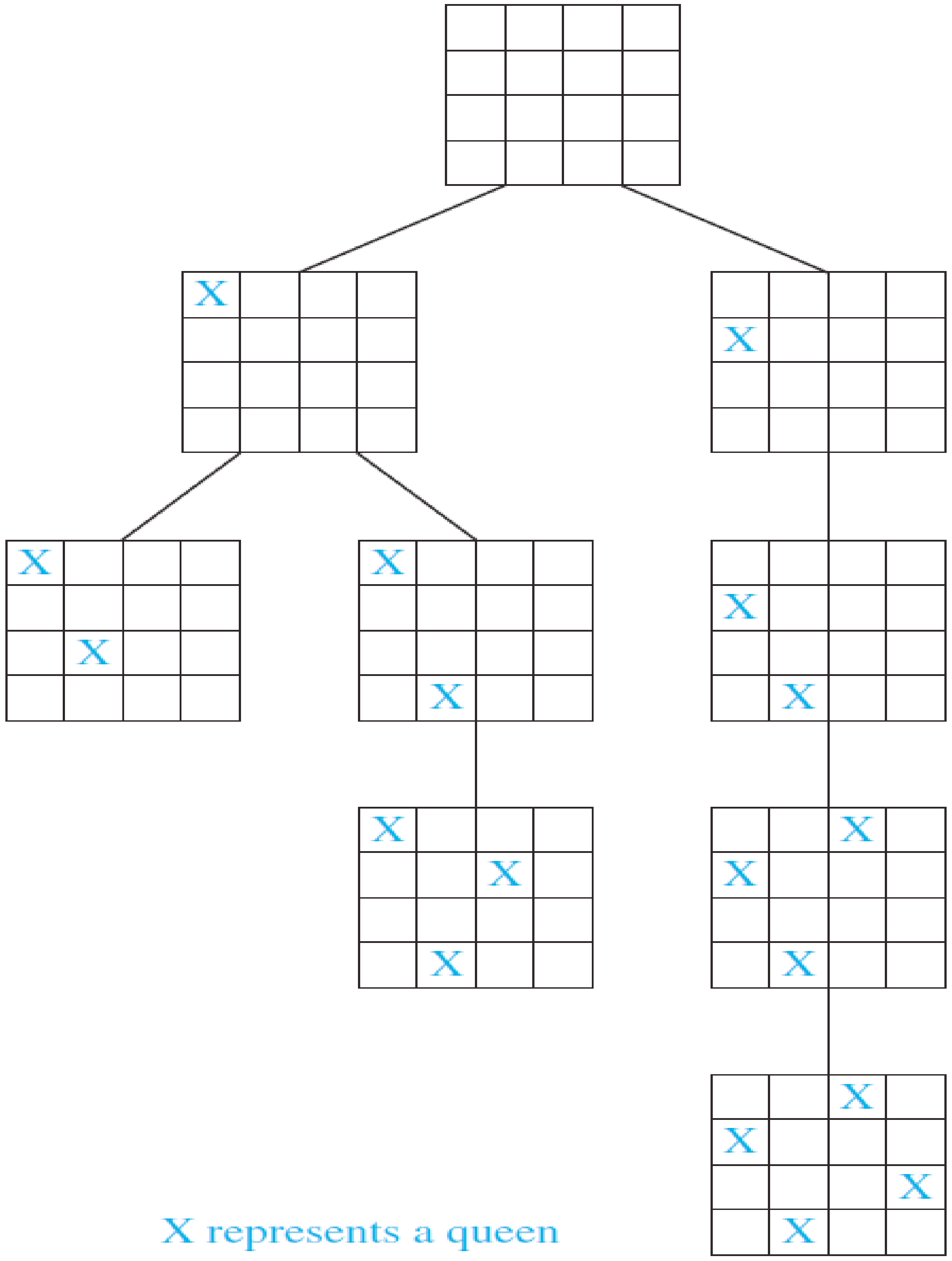
Backtracking Applications

- *Solution (cont.):* We will use backtracking to solve the n-queens problem. We start with an empty chessboard. At stage $k+1$ we attempt putting an additional queen on the board in the $(k+1)$st column, where there are already queens in the first k columns. We examine squares in the $(k+1)$st column starting with the square in the first row, looking for a position to place this queen so that it is not in the same row or on the same diagonal as a queen already on the board.

Backtracking Applications

- *Solution (cont.):* If it is impossible to find a position to place the queen in the *(k* + 1*)*st column, backtrack to the placement of the queen in the *k*th column, and place this queen in the next allowable row in this column, if such a row exists. If no such row exists, backtrack further.
- In particular, Figure 12 displays a backtracking solution to the four-queens problem. In this solution, we place a queen in the first row and column. Then we put a queen in the third row of the second column.

FIGURE 12



Backtracking Applications

- *Solution (cont.):* However, this makes it impossible to place a queen in the third column. So we backtrack and put a queen in the fourth row of the second column. When we do this, we can place a queen in the second row of the third column. But there is no way to add a queen to the fourth column. This shows that no solution results when a queen is placed in the first row and column. We backtrack to the empty chessboard, and place a queen in the second row of the first column. This leads to a solution as shown in Figure 12.

HOMEWORK: Exercises 2, 4, 6, 12, 14, 16 on pp. 795-796