Trees

Chapter 11

Chapter Summary

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

Introduction to Trees

Section 11.1

Section Summary

- Introduction to Trees
- Rooted Trees
- Trees as Models
- Properties of Trees

Trees

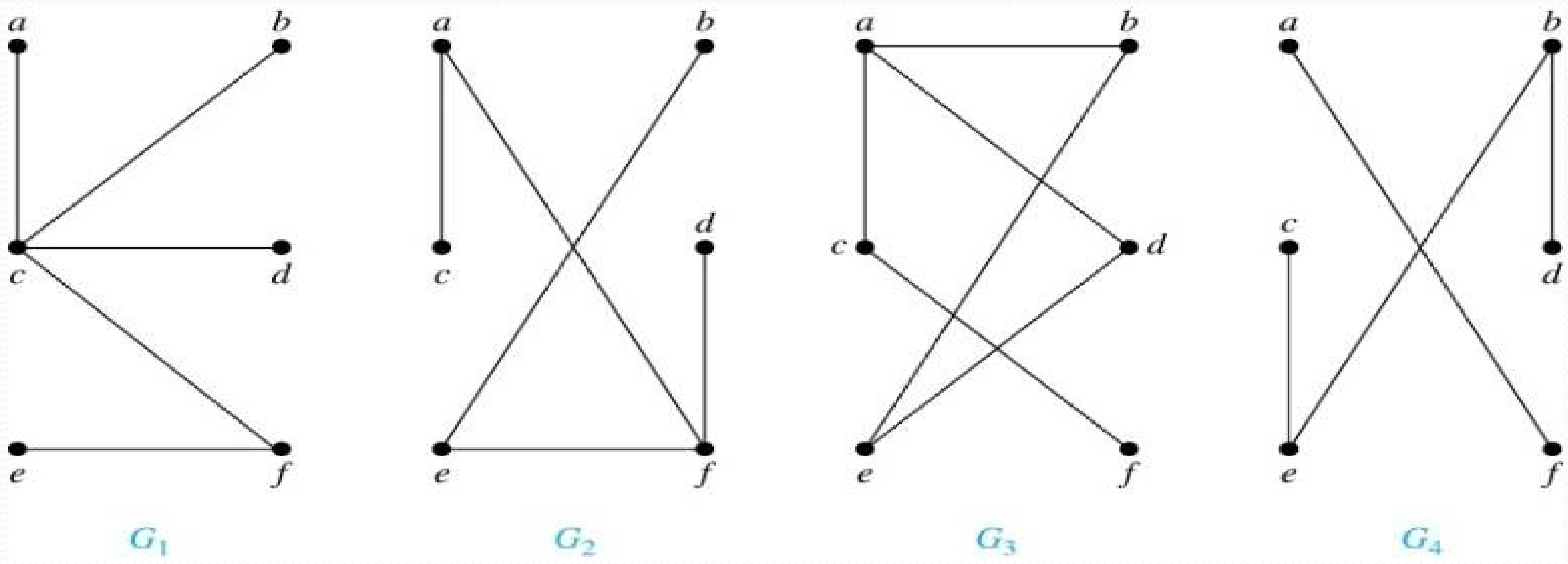
Definition: A *tree* is a connected undirected graph with no simple circuits.

Definition: An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices. A tree cannot contain multiple edges or loops.

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

Trees

Example: Which of these graphs are trees?

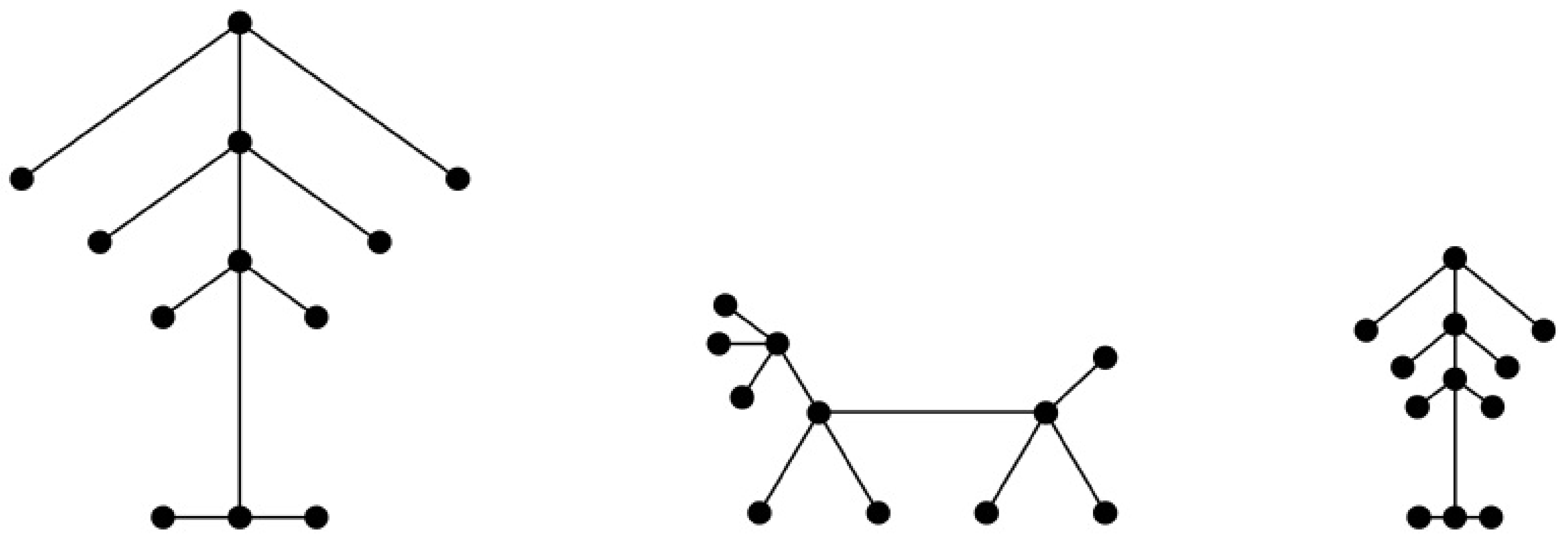


Solution: G_1 and G_2 are trees - both are connected and have no simple circuits. G_3 is not a tree because *e*, *b*, *a*, *d*, *e* is a simple circuit,. G_4 is not a tree because it is not connected.

FOREST

Definition: A *forest* is a graph that has no simple circuit, but is not connected. Each of the connected components in a forest is a tree.

This is one graph with three connected components.

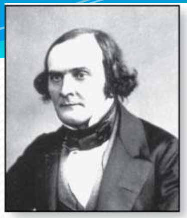


Applications of Trees

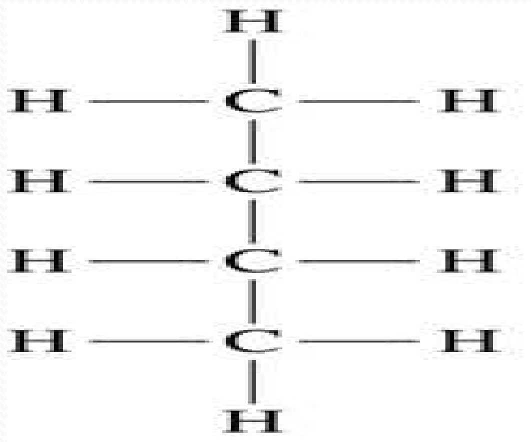
Section 11.2

Trees as Models

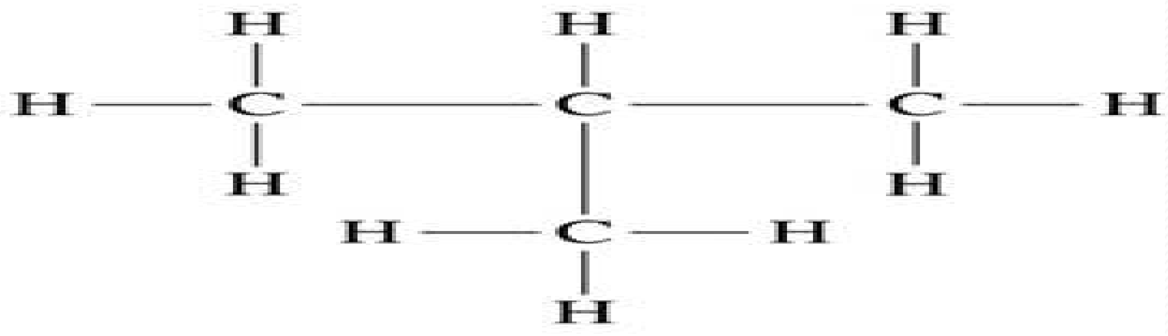
Arthur Cayley
(1821-1895)



- Trees are used as models in computer science, chemistry, geology, botany, psychology, and many other areas.
- Trees were introduced by the mathematician Cayley in 1857 in his work counting the number of isomers of saturated hydrocarbons. The two isomers of butane are:.



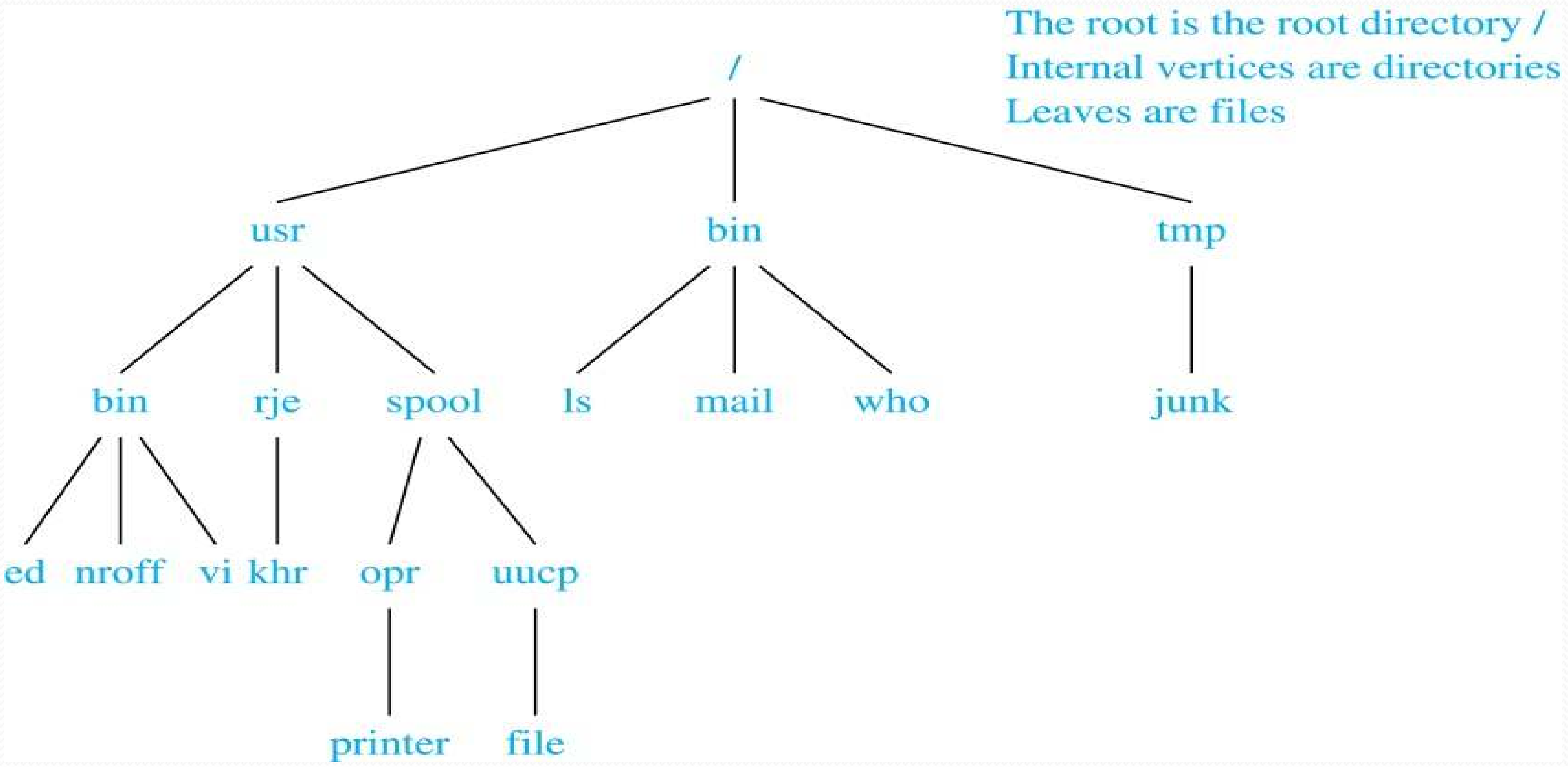
Butane



Isobutane

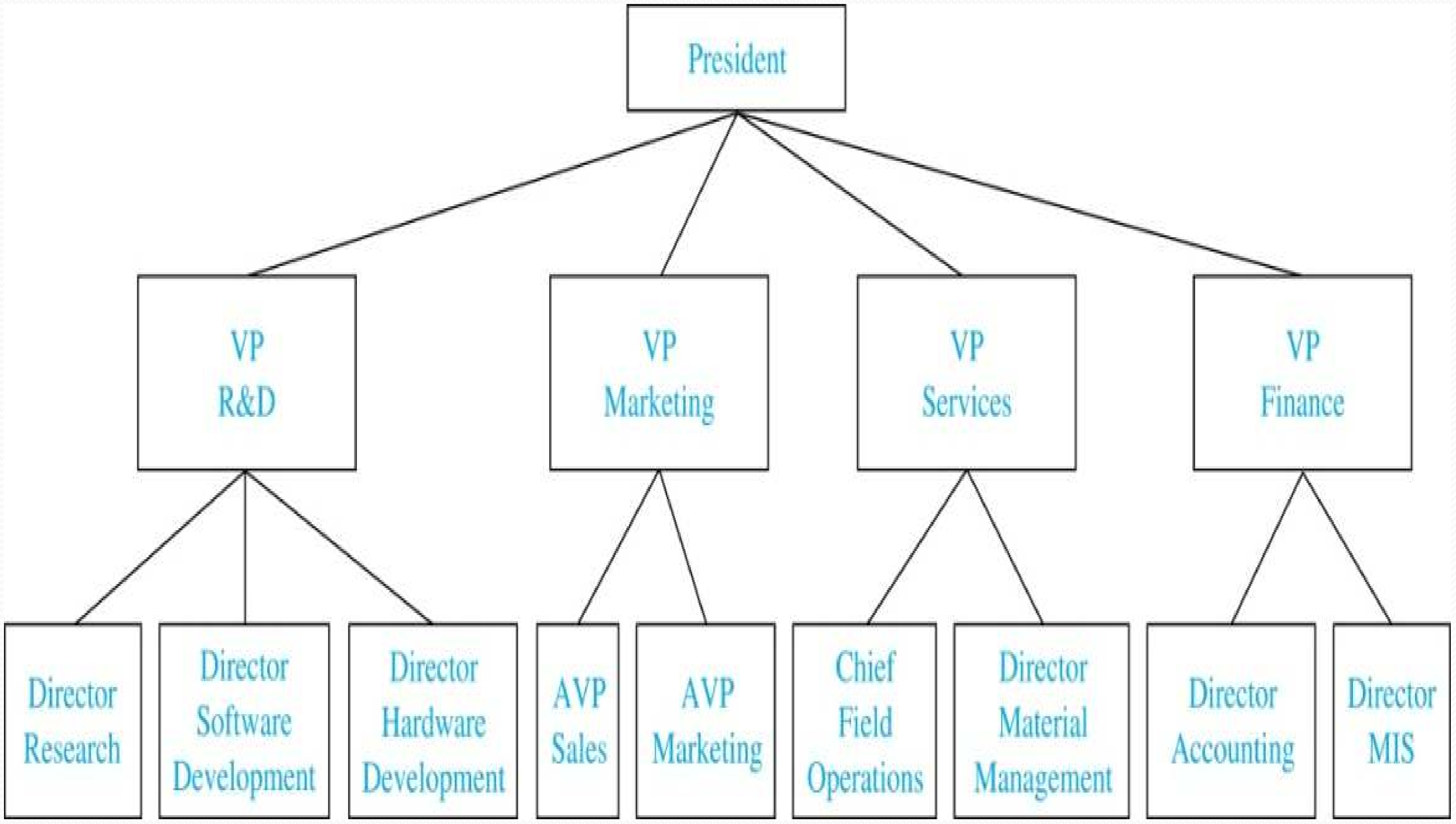
Trees as Models

- The organization of a computer file system into directories, subdirectories, and files is naturally represented as a tree.



Trees as Models

- Trees are used to represent the structure of organizations.

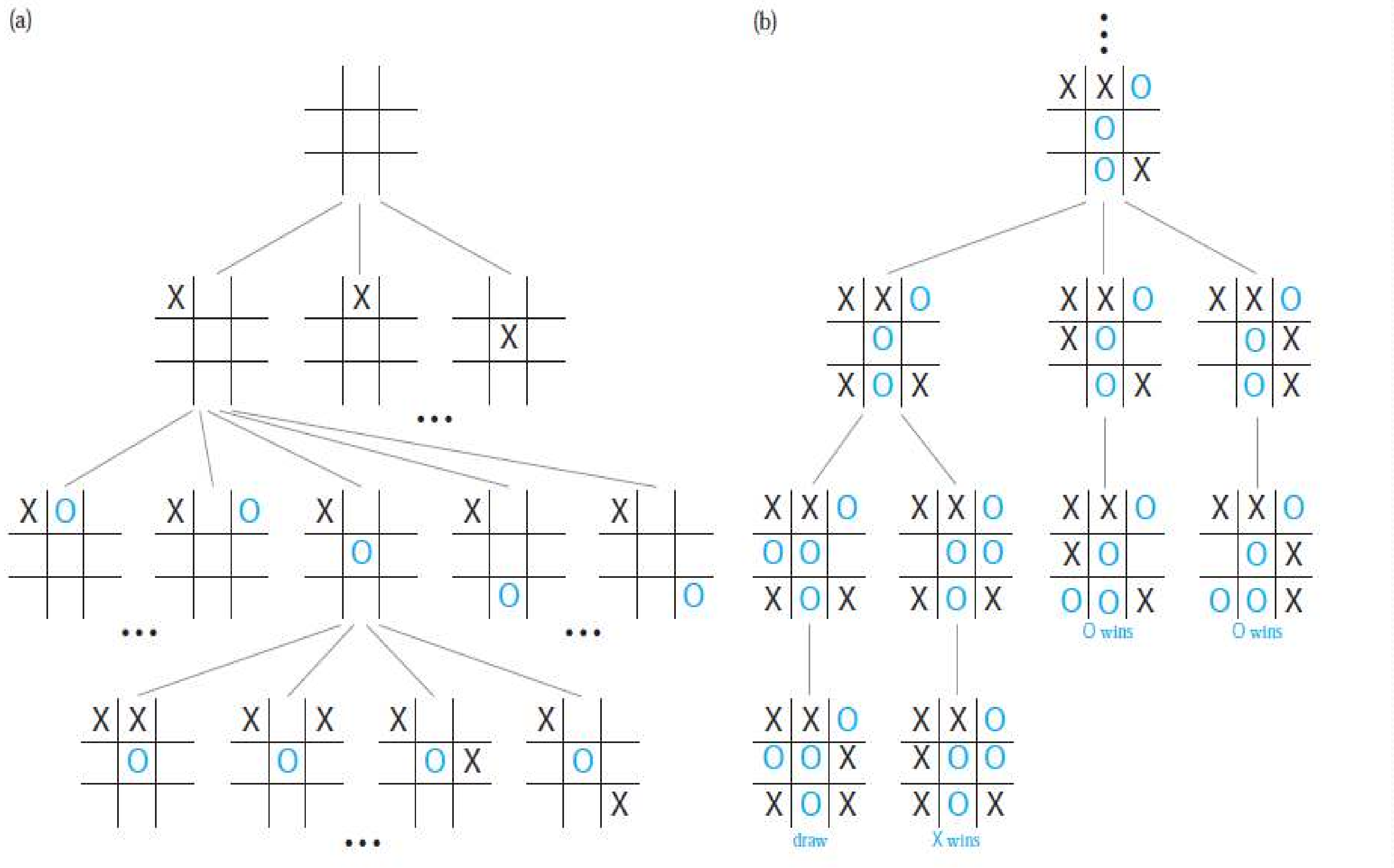


Applications of Trees

- **Game Trees**

Trees can be used to analyze certain types of games such as tic-tac-toe, nim, checkers, and chess.

Game Tree for Tic-Tac-Toe



Universal Address Systems

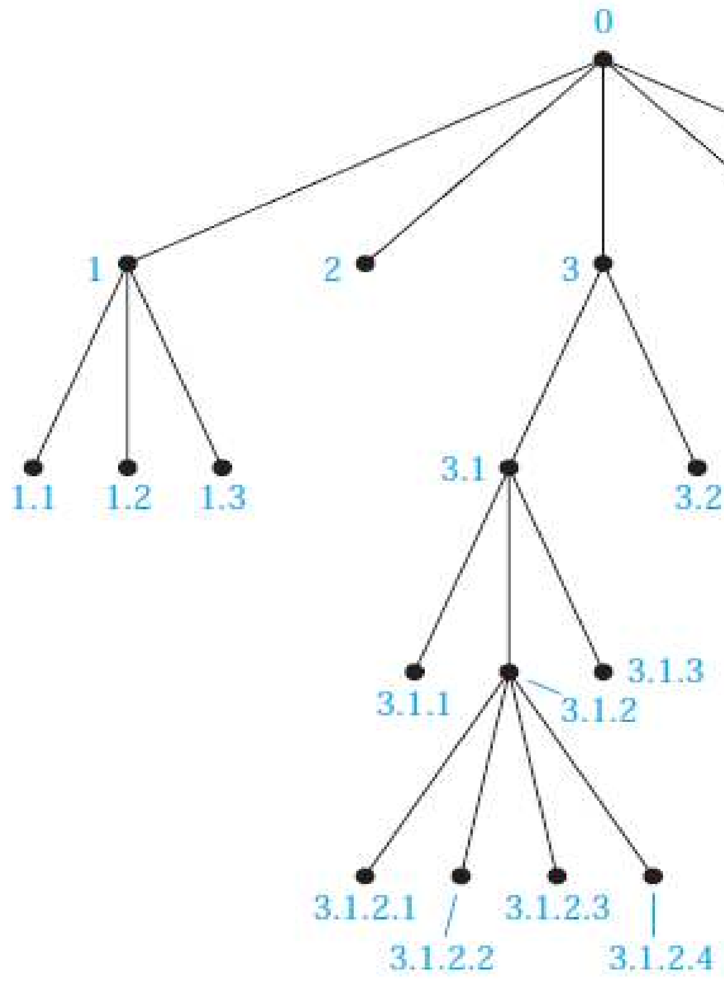


FIGURE 1 The Universal Address System of an Ordered Rooted Tree.

Prefix code

Definition: A code that has the property that the code of a character is never a prefix of the code of another character.

- A prefix code can be represented using a binary tree, where the characters are the labels of the leaves in the tree.
- The edges of the tree are labeled so that an edge leading to a left child is assigned a 0 and an edge leading to a right child is assigned a 1.
- The bit string used to encode a character is the sequence of labels of the edges in the unique path from the root to the leaf that has this character as its label.
- For instance, the tree in Figure 5 represents the encoding of *e* by 0, *a* by 10, *t* by 110, *n* by 1110, and *s* by 1111.

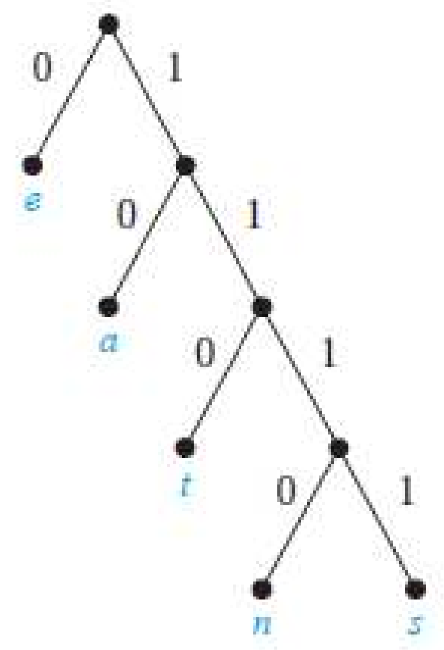


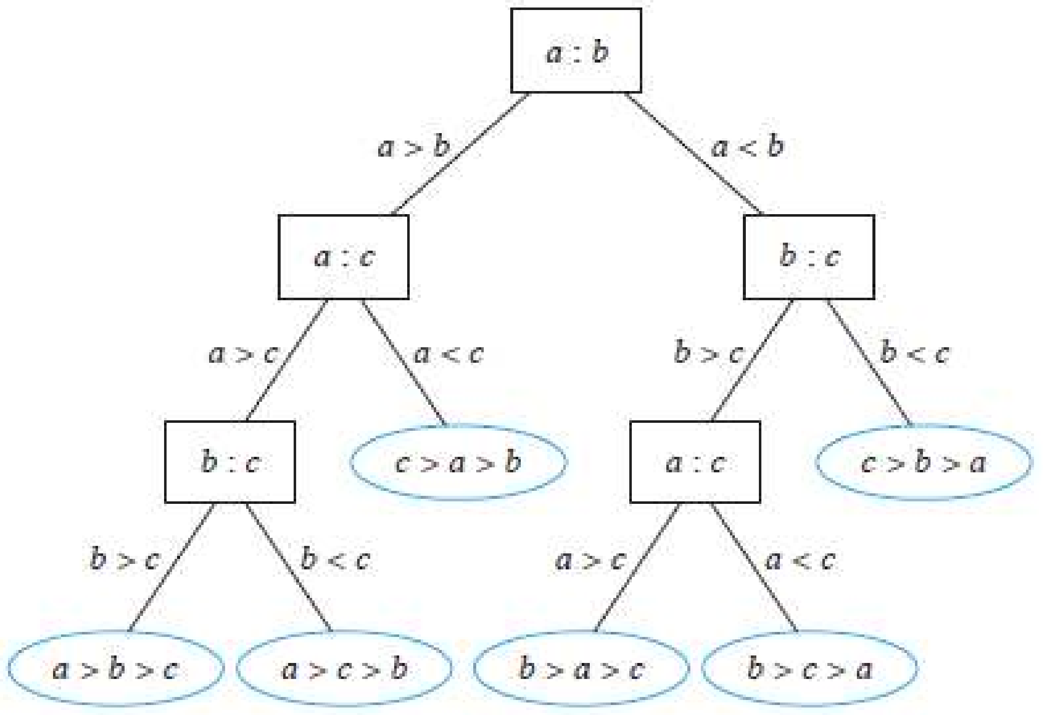
FIGURE 5 A Binary Tree with a Prefix Code.

Decision Trees

Definition: A rooted tree where each vertex represents a possible outcome of a decision and the leaves represent the possible solutions of a problem.

- Rooted trees can be used to model problems in which a series of decisions leads to a solution.
- The possible solutions of the problem correspond to the paths to the leaves of this rooted tree.

Example : A decision tree that orders the elements of the list *a*, *b*, *c*.

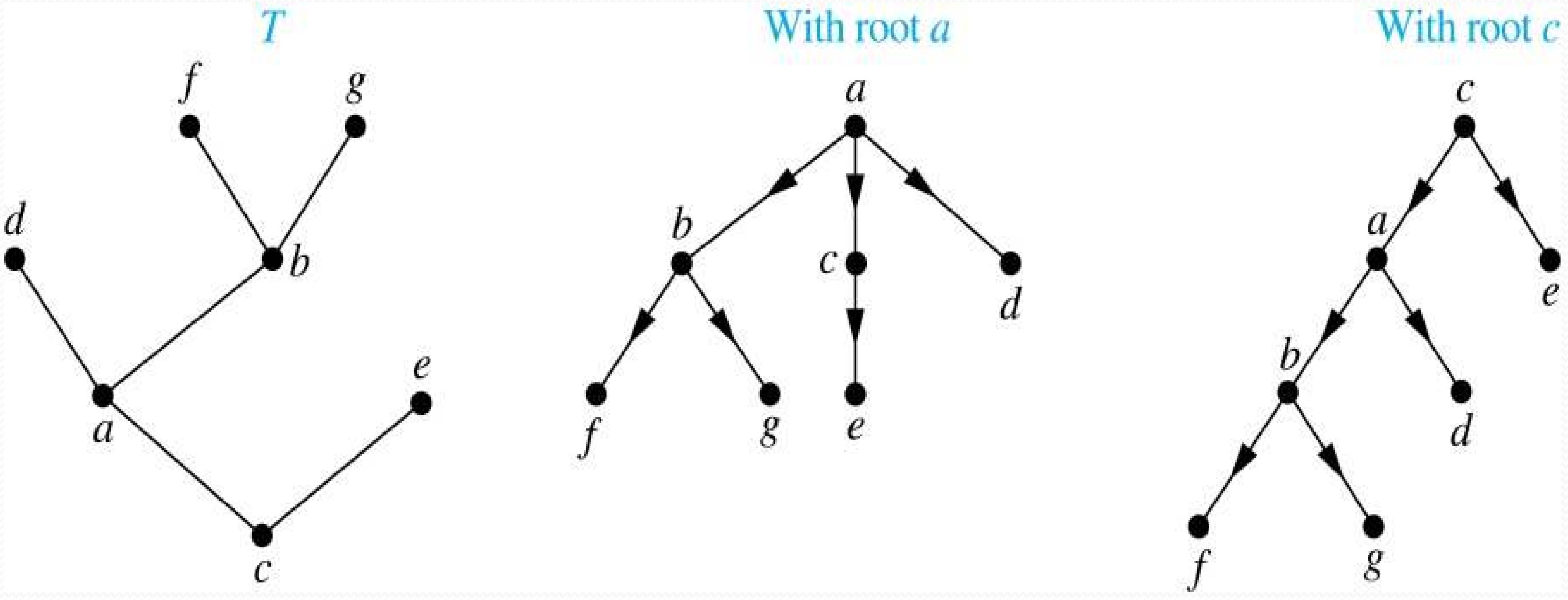


A Decision Tree for Sorting Three Distinct Elements.

Rooted Trees

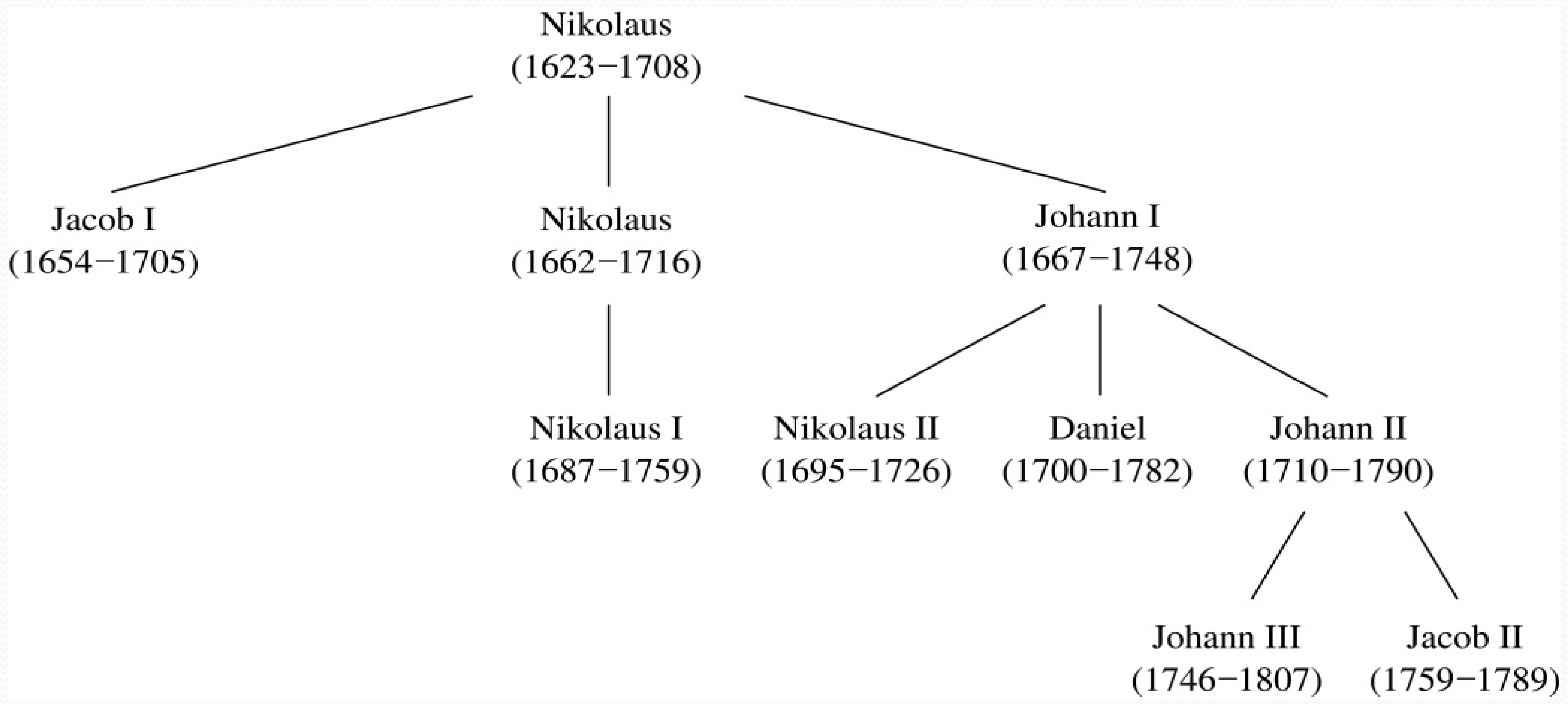
Definition: A *rooted tree* is a tree in which one vertex has been designated as the *root* and every edge is directed away from the root.

- An unrooted tree is converted into different rooted trees when different vertices are chosen as the root.



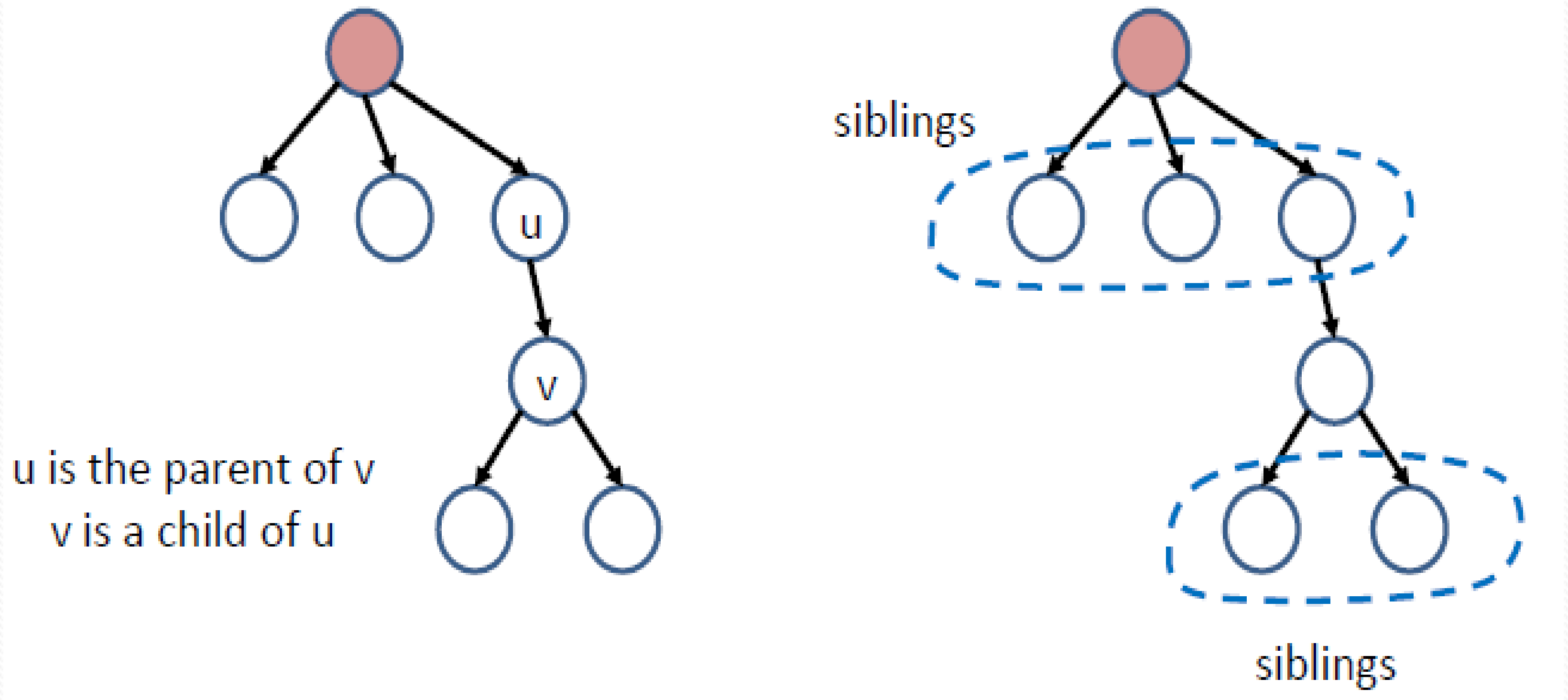
Rooted Tree Terminology

- Terminology for rooted trees is a mix from botany and genealogy (such as this family tree of the Bernoulli family of mathematicians).



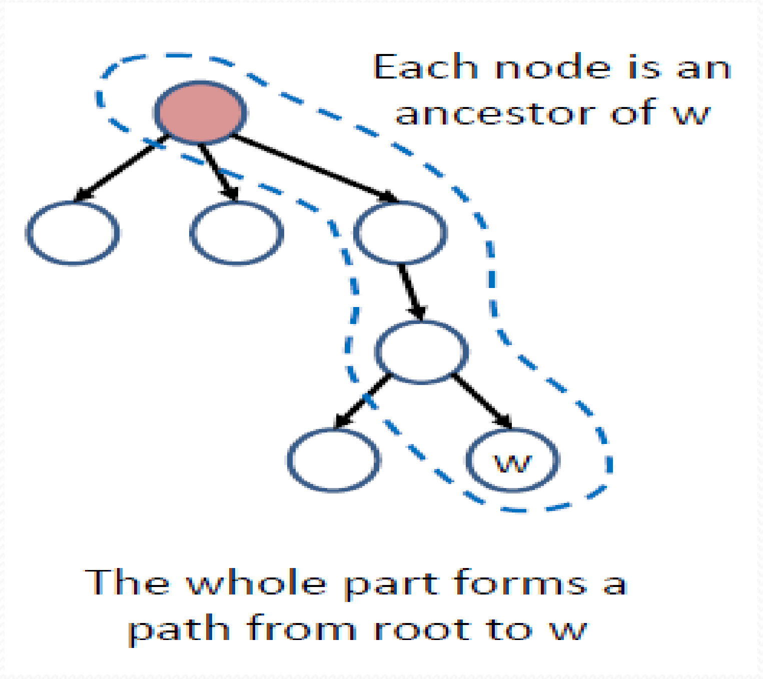
Rooted Tree Terminology

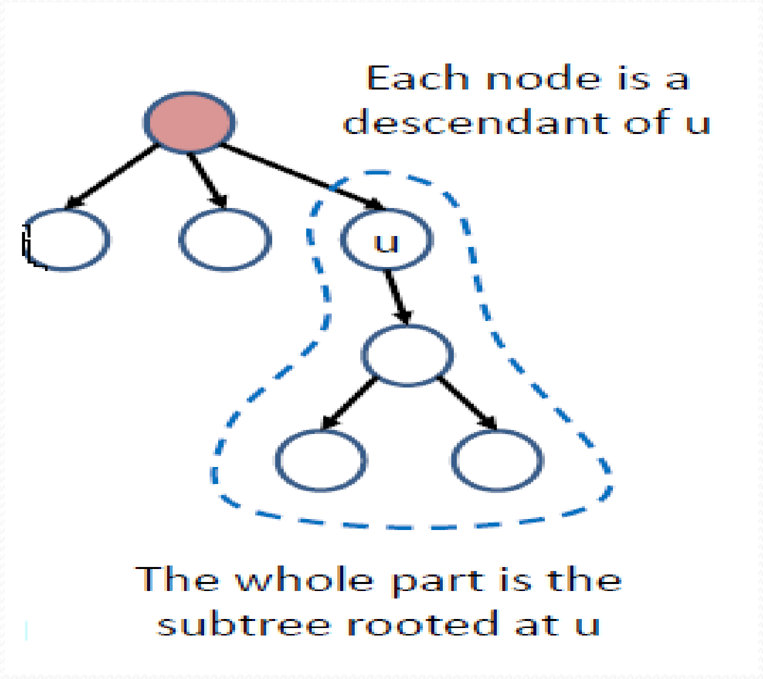
- If *v* is a vertex of a rooted tree other than the root, the *parent* of *v* is the unique vertex *u* such that there is a directed edge from *u* to *v*. When *u* is a parent of *v*, *v* is called a *child* of *u*. Vertices with the same parent are called *siblings*.



Rooted Tree Terminology

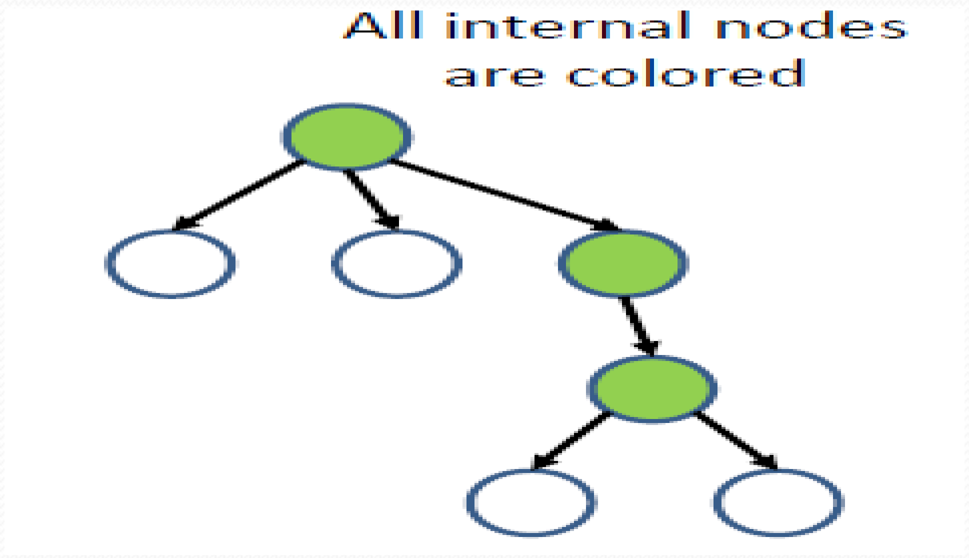
- The *ancestors* of a vertex are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root.
- The *descendants* of a vertex *v* are those vertices that have *v* as an ancestor. The subtree rooted at u includes all the descendants of u, and all edges that connect between them.


Each node is an ancestor of w
w
The whole part forms a path from root to w


Each node is a descendant of u
u
The whole part is the subtree rooted at u

Rooted Tree Terminology

- A vertex of a rooted tree with no children is called a *leaf*. Vertices that have children are called *internal vertices*.



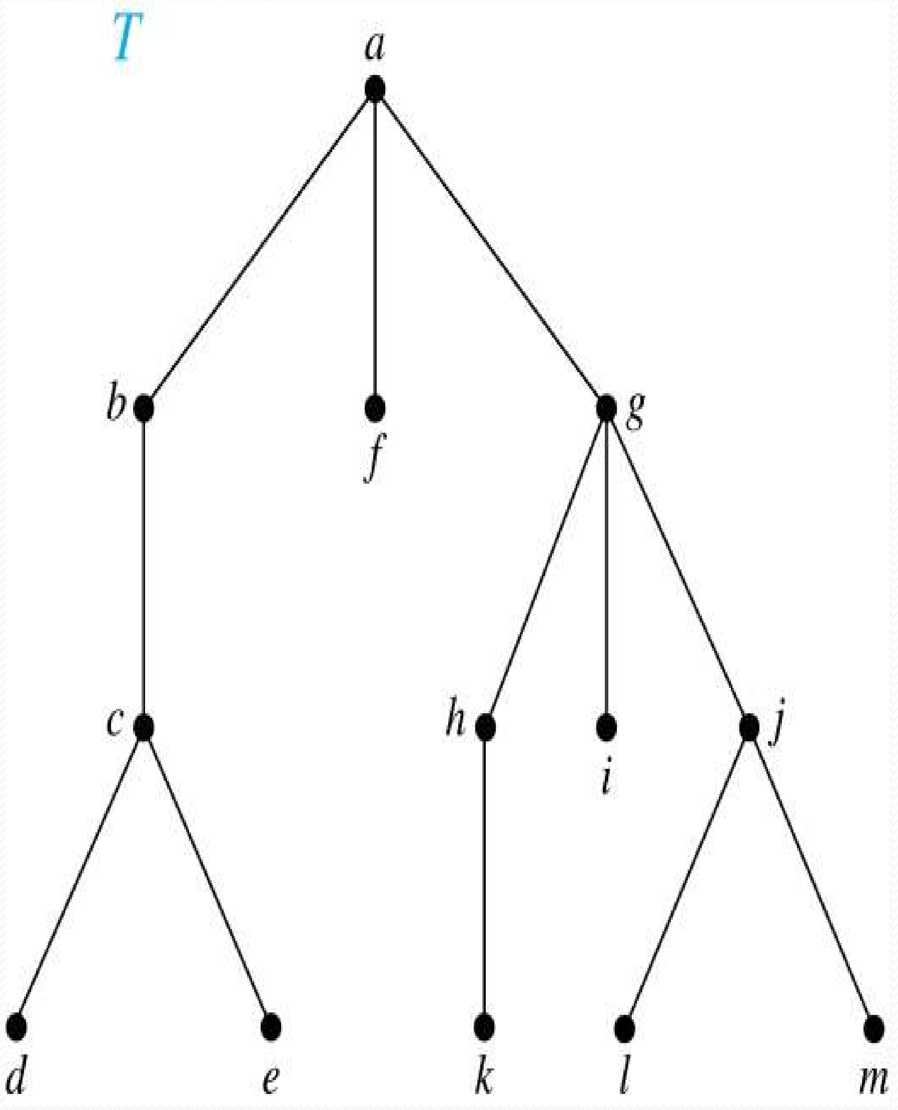
Terminology for Rooted Trees

Example: In the rooted tree *T* (with root *a*):

(i) Find the parent of *c*, the children of *g*, the siblings of *h*, the ancestors of *e*, and the descendants of *b*.

Solution:

(i) 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*.



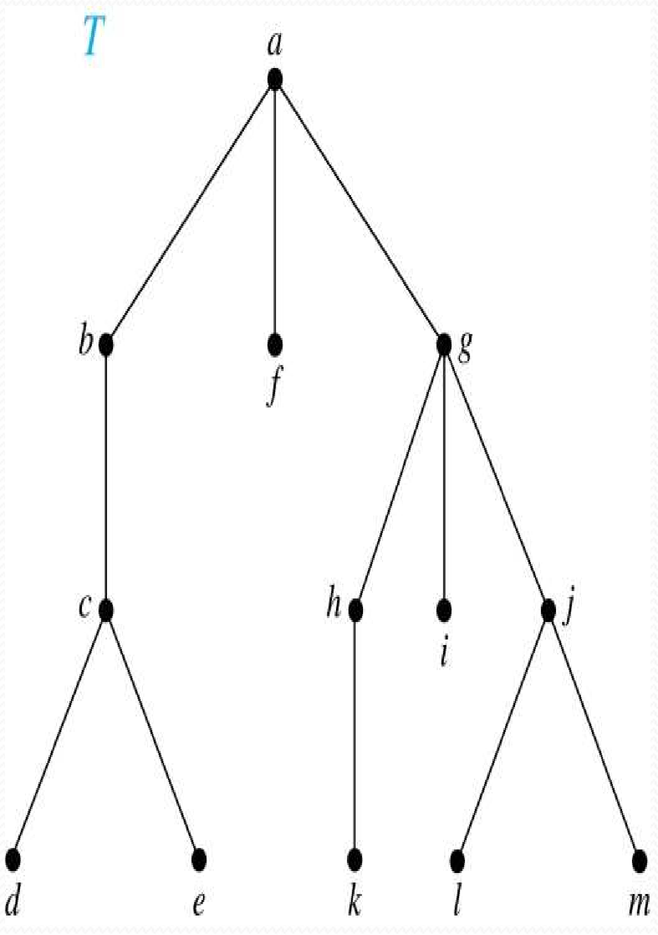
Terminology for Rooted Trees

Example: In the rooted tree *T* (with root *a*):

(i) Find all internal vertices and all leaves.

Solution:

(i) The internal vertices are *a*, *b*, *c*, *g*, *h*, and *j*. The leaves are *d*, *e*, *f*, *i*, *k*, *l*, and *m*.

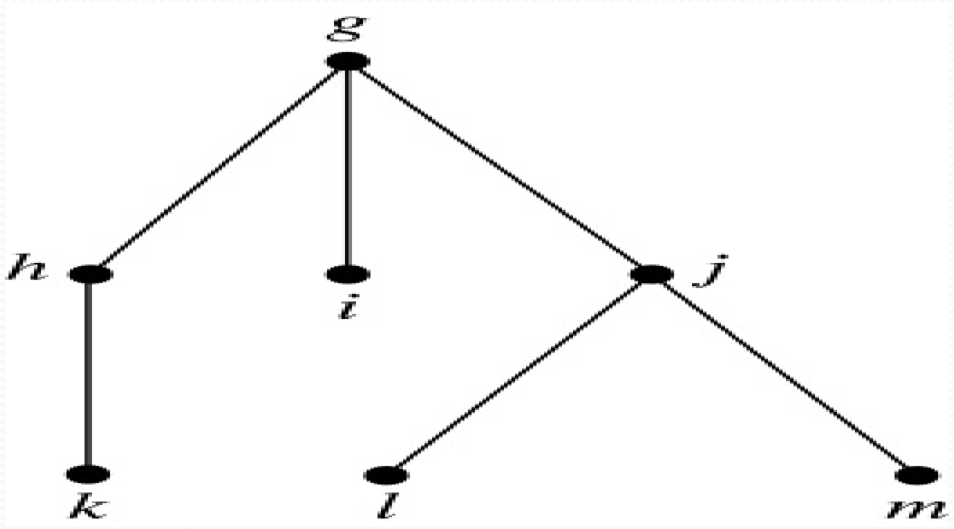


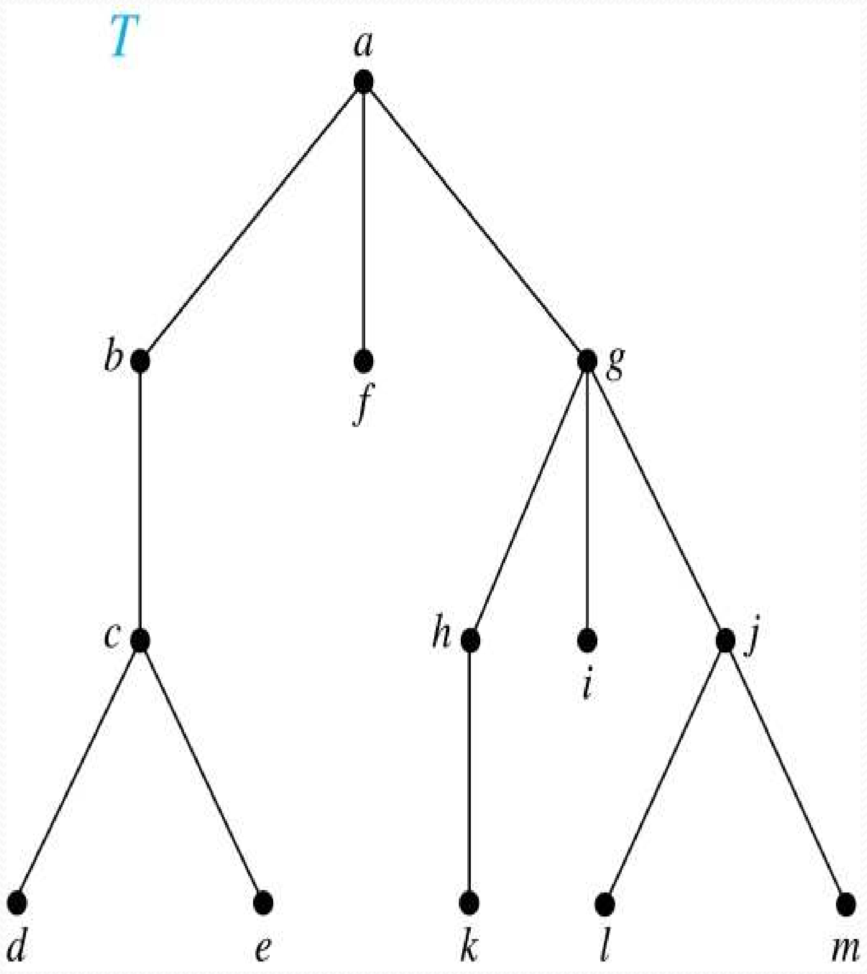
Terminology for Rooted Trees

(i) What is the subtree rooted at *g*?

Solution:

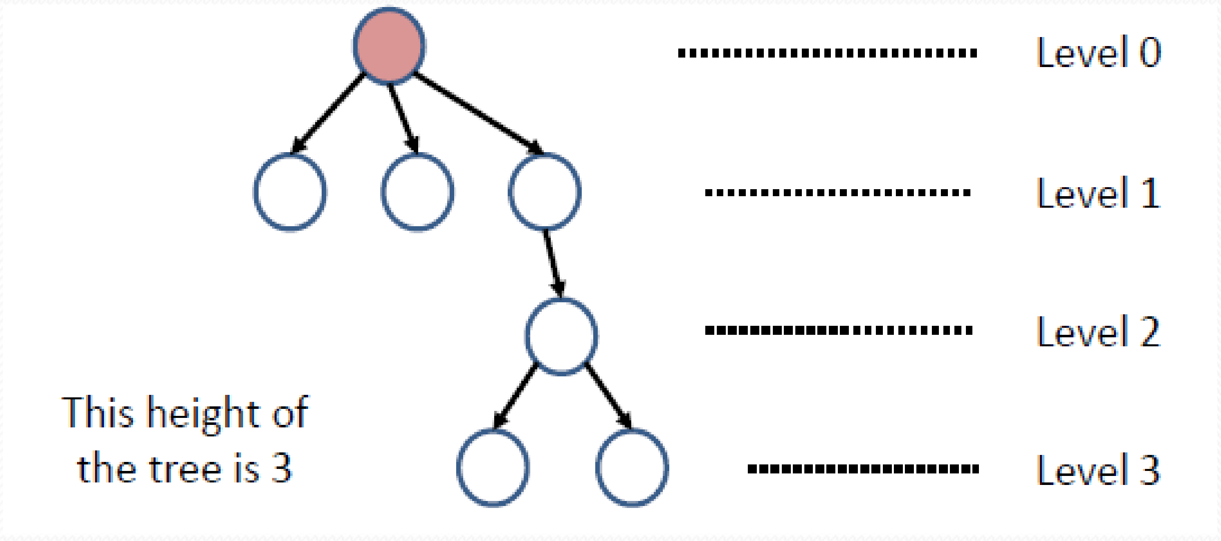
(i) We display the subtree rooted at *g*.





Level of vertices and height of trees

- When working with trees, we often want to have rooted trees where the sub trees at each vertex contain paths of approximately the same length.
- To make this idea precise we need some definitions:
  - The *level* of a vertex *v* in a rooted tree is the length of the unique path from the root to this vertex.
  - The *height* of a rooted tree is the maximum of the levels of the vertices.



Level of vertices and height of trees

Example:

(i) Find the level of each vertex in the tree to the right.

(ii) What is the height of the tree?

Solution:

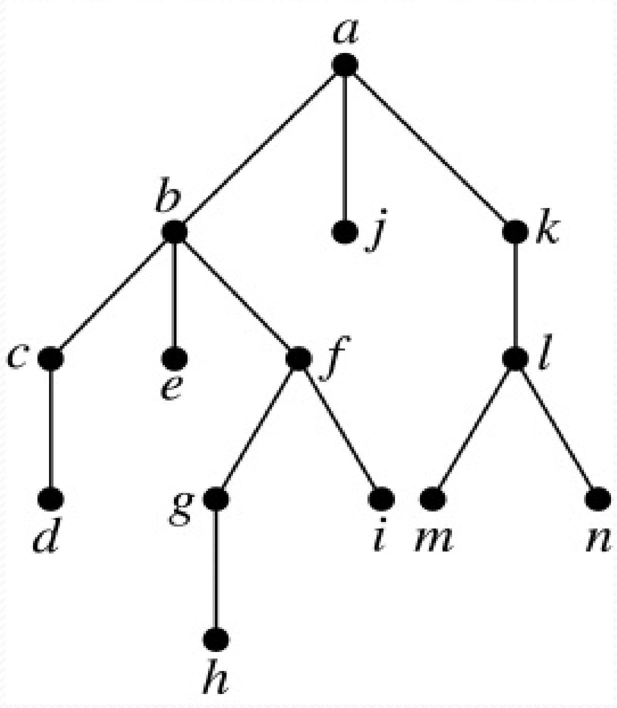
(i) The root a is at level 0.

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.

Vertex h is at level 4.

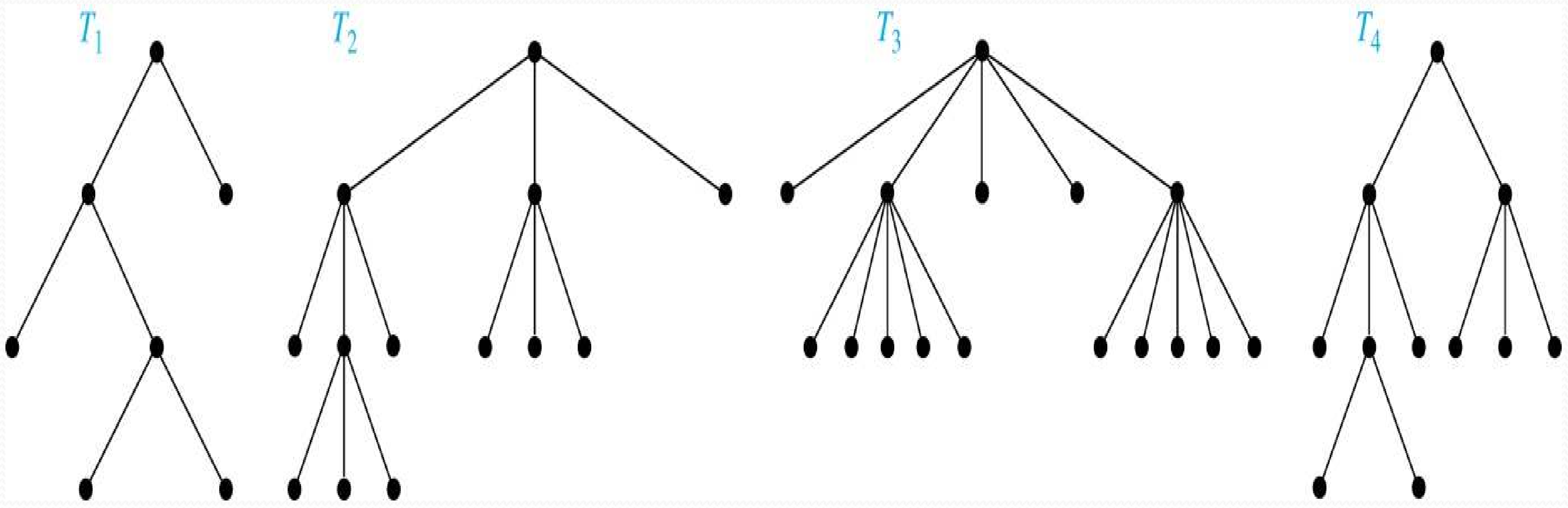


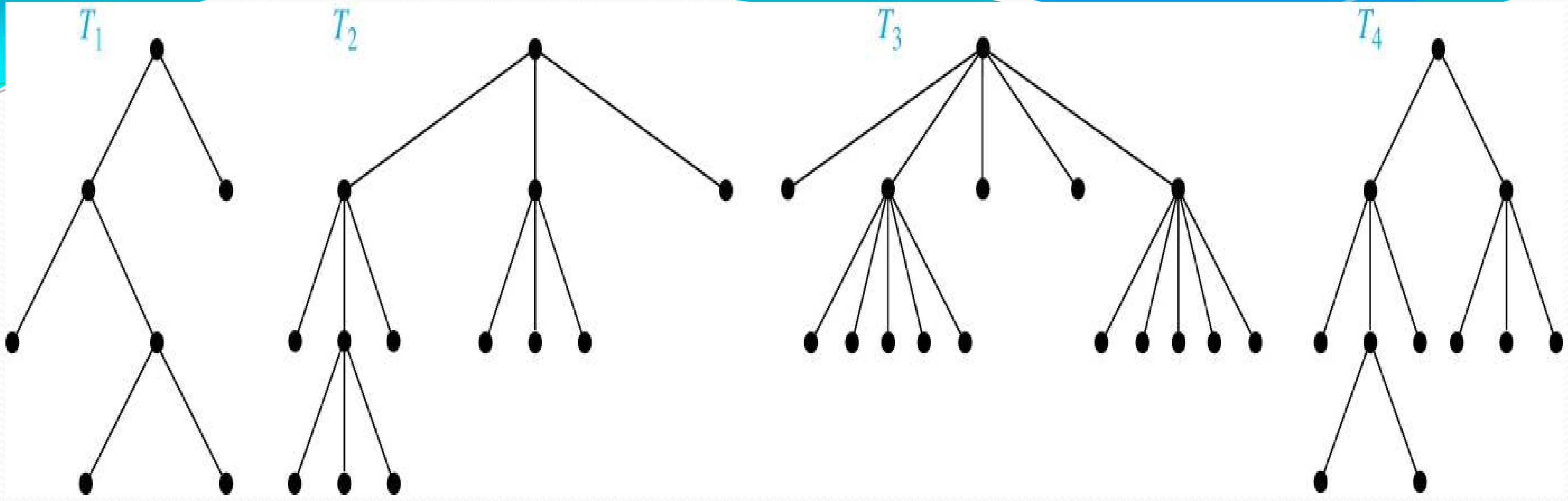
(ii) The height is 4, since 4 is the largest level of any vertex.

m-ary Rooted Trees

Definition: 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.

Example: Are the following rooted trees full *m*-ary trees for some positive integer *m*?





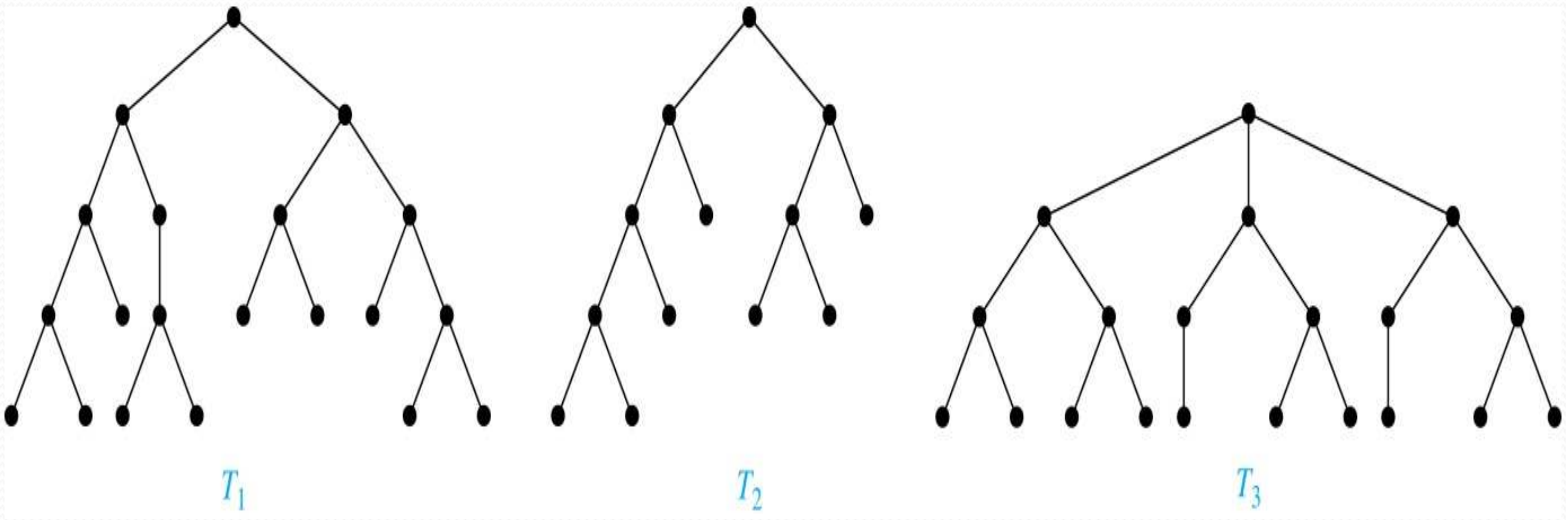
Solution:

- 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.

Balanced *m*-Ary Trees

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

Example: Which of the rooted trees shown below is balanced?

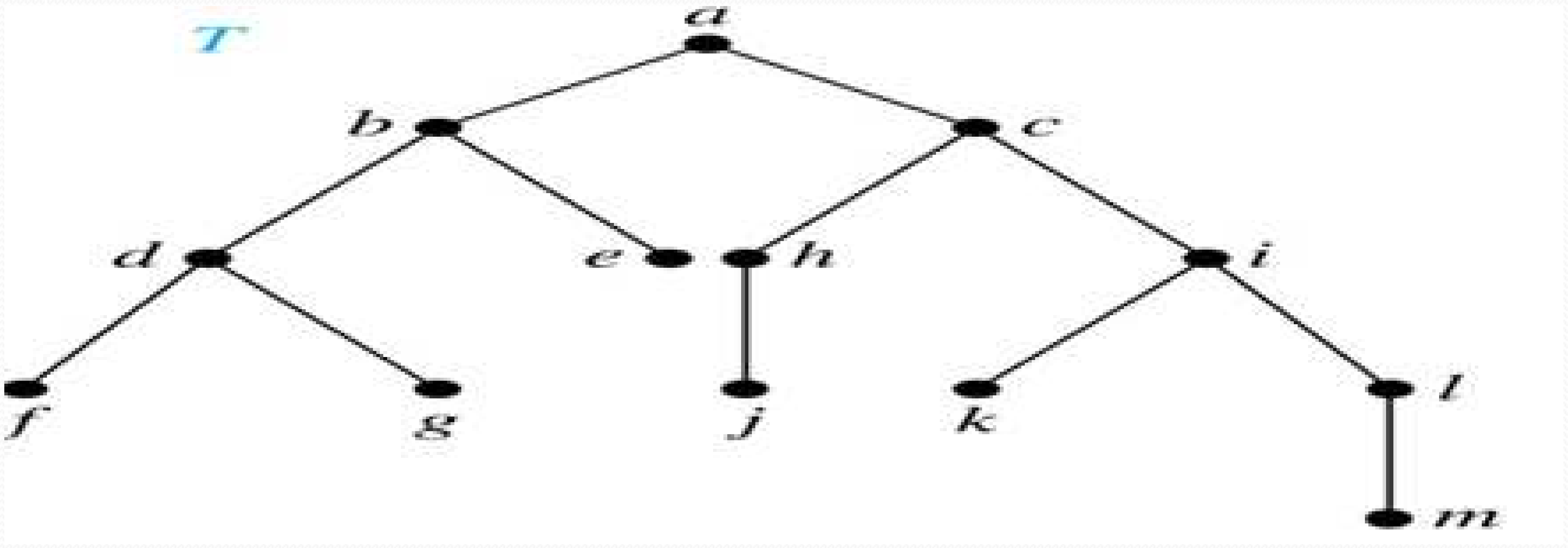


Solution: T_1 and T_3 are balanced, but T_2 is not because it has leaves at levels 2, 3, and 4.

Ordered Rooted Trees

Definition: An *ordered rooted tree* is a rooted tree where the children of each internal vertex are ordered.

- We draw ordered rooted trees so that the children of each internal vertex are shown in order from left to right.



Binary Trees

Definition: A *binary tree* is an ordered rooted where each internal vertex has at most two children. If an internal vertex of a binary tree has two children, the first is called the *left child* and the second 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 this vertex.

Example:

Consider the binary tree T.

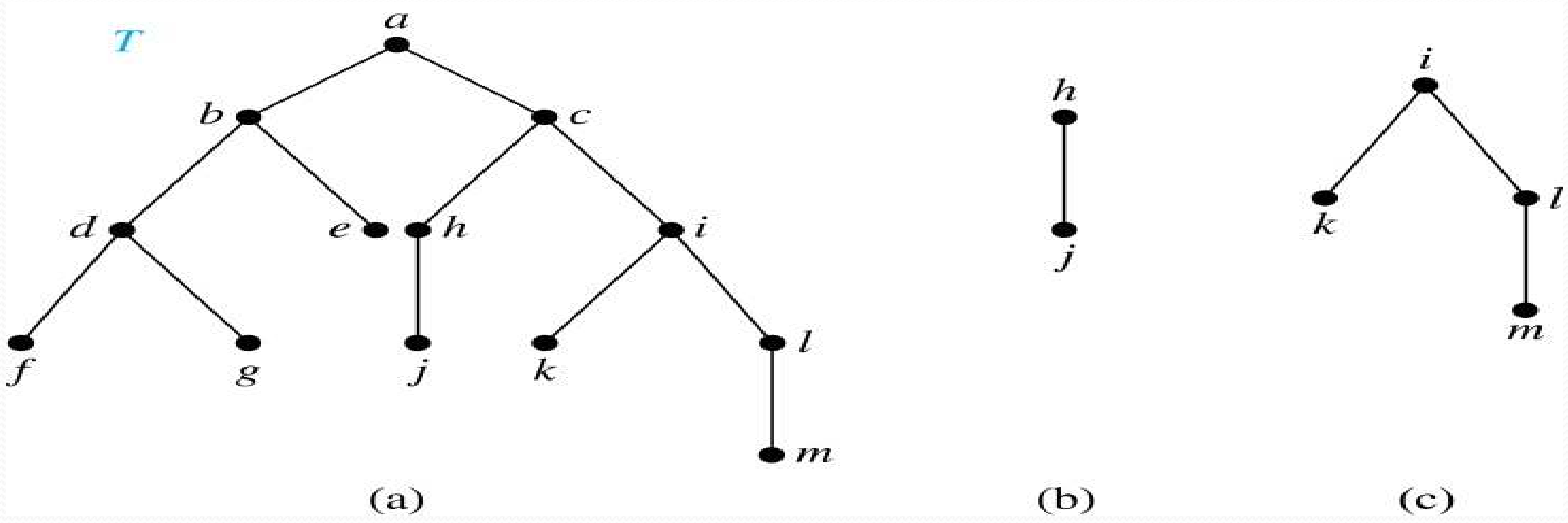
(i) What are the left and right children of d?

(ii) What are the left and right sub trees of c?

Solution:

(i) The left child of d is f and the right child is g.

(ii) The left and right subtrees of c are displayed in (b) and (c).



(a) (b) (c)

Properties of Trees

- A tree with n vertices has $n - 1$ edges.
- A full m-ary tree with i internal vertices has $n = mi + 1$ vertices.
- 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,

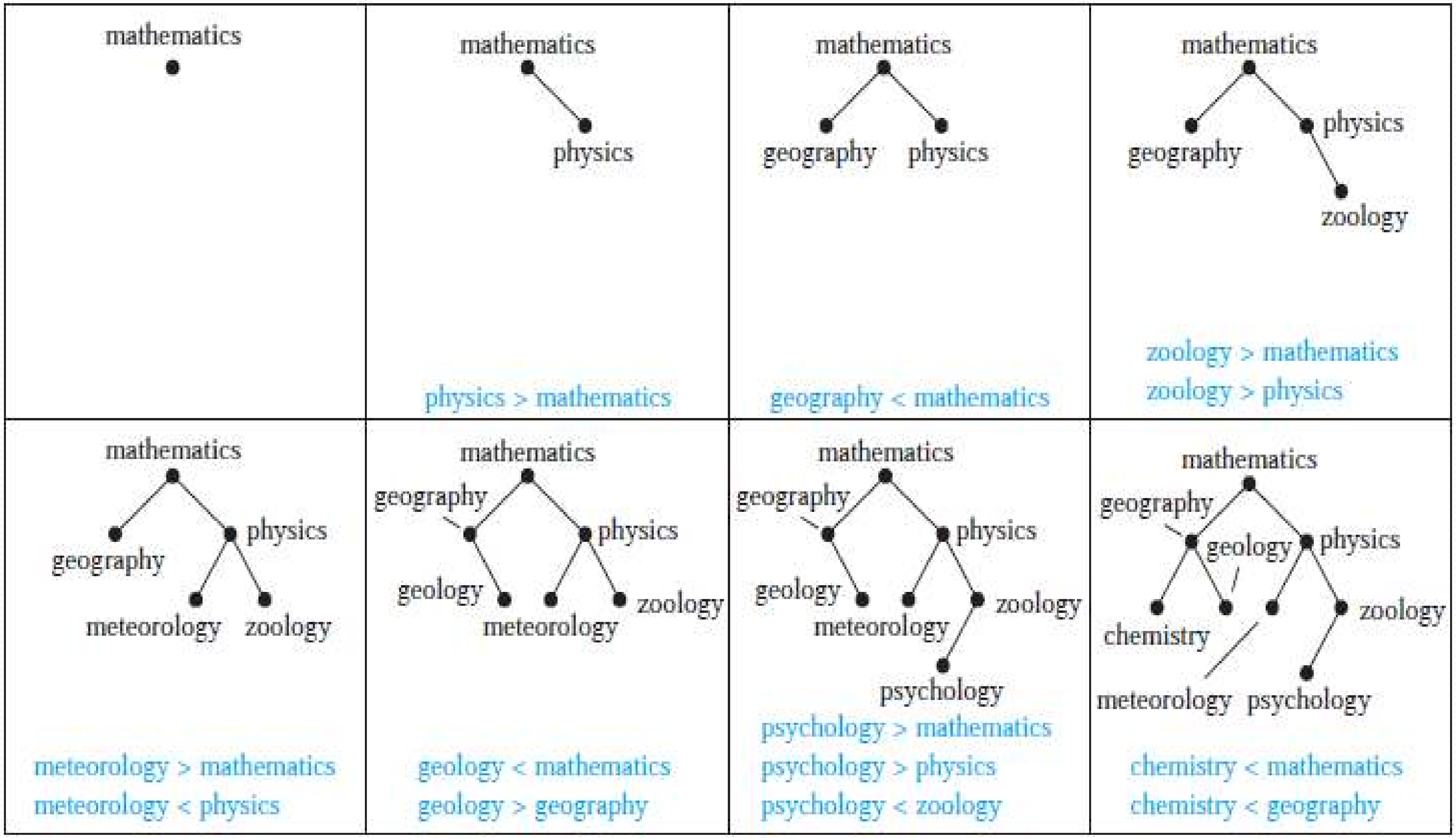
  (iii) l leaves has $n = (ml - 1)/(m - 1)$ vertices and $i = (l - 1)/(m - 1)$ internal vertices.
- There are at most m^h leaves in an m-ary tree of height h.

Binary Search Tree

Definition: A binary tree in which the vertices are labeled with items so that a label of a vertex is greater than the labels of all vertices in the left subtree of this vertex and is less than the labels of all vertices in the right subtree of this vertex.

- 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.

Example : Form a binary search tree for the words mathematics, physics, geography, zoology, meteorology, geology, psychology, and chemistry (using alphabetical order).



Tree Traversal

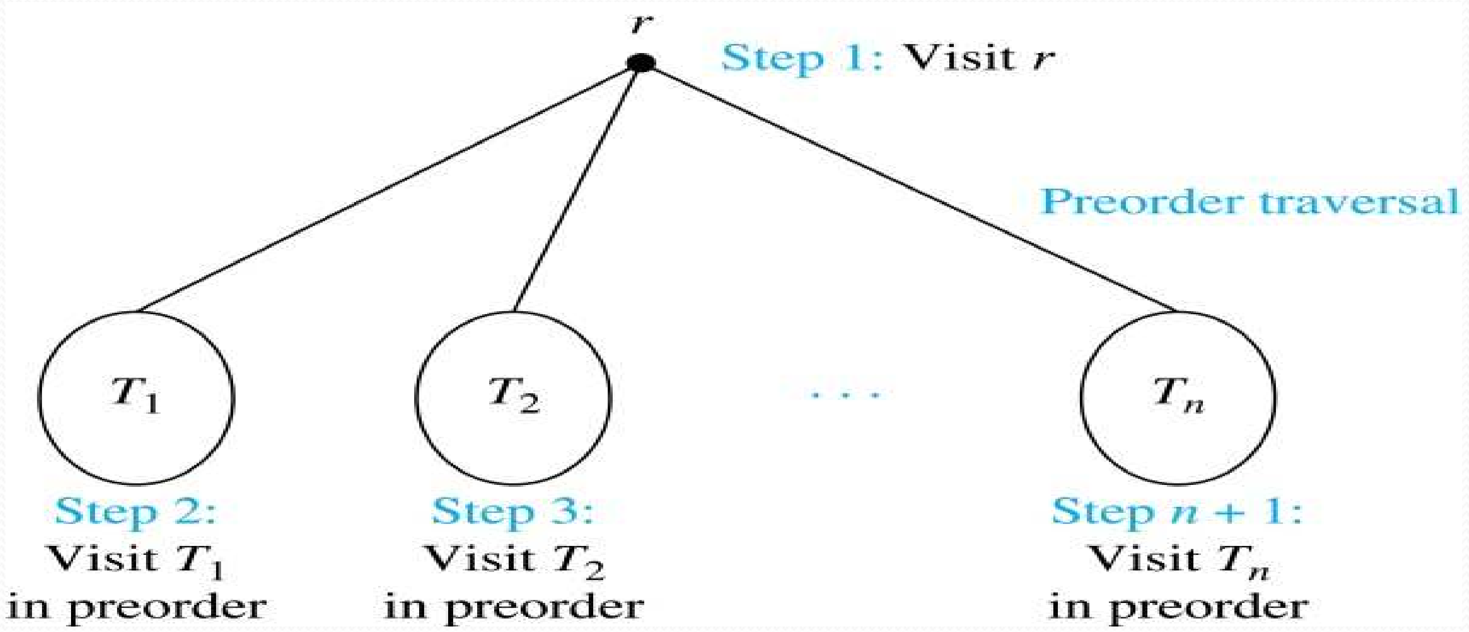
Section 11.3

Tree Traversal

- Procedures for systematically visiting every vertex of an ordered tree are called *traversals*.
- The three most commonly used *traversals* are *preorder traversal*, *inorder traversal*, and *postorder traversal*.

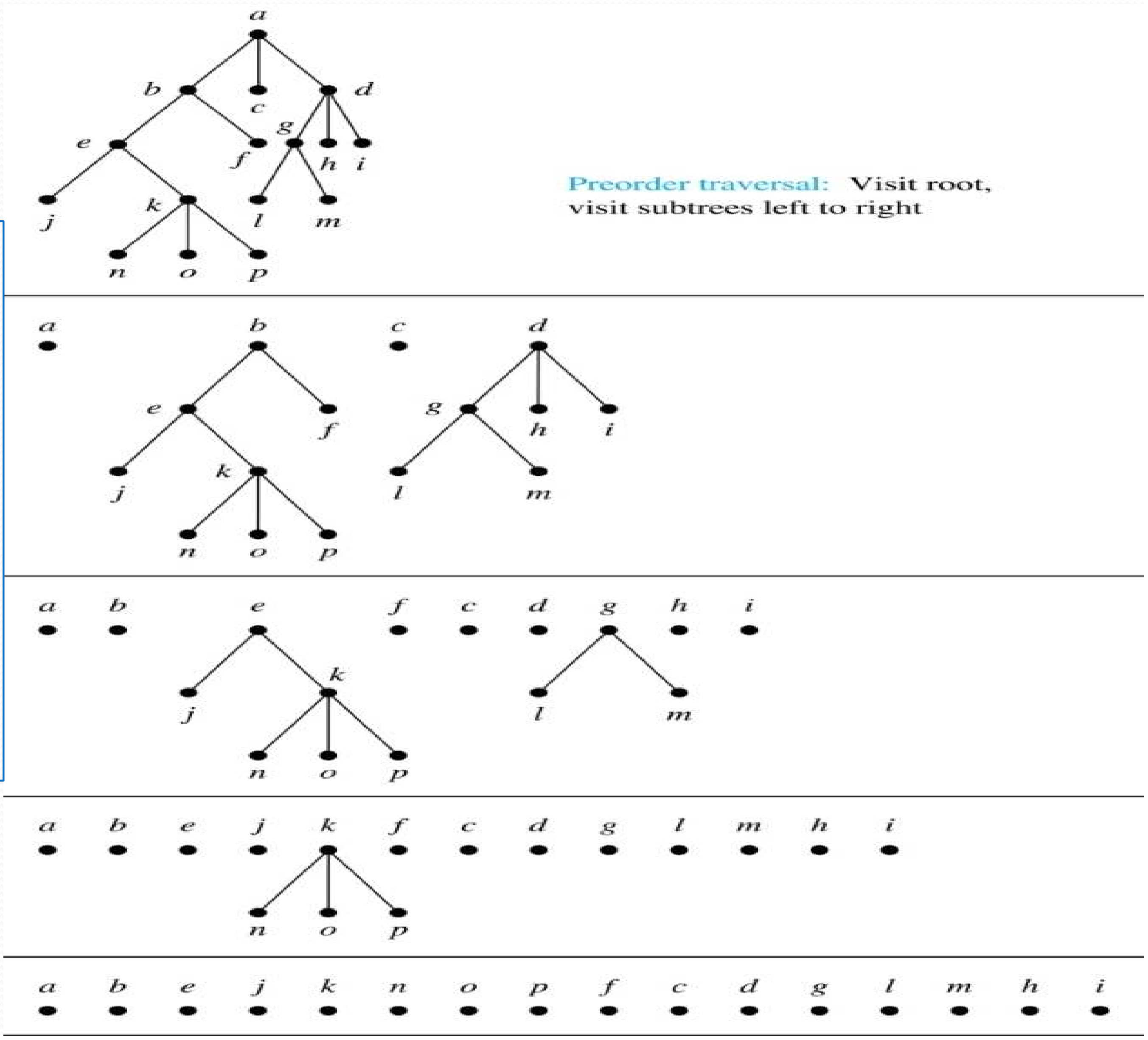
Preorder Traversal

Definition: 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, …, T_n are the subtrees of r from left to right in T. The preorder traversal begins by visiting r, and continues by traversing T_1 in preorder, then T_2 in preorder, and so on, until T_n is traversed in preorder.



r
Step 1: Visit r
Preorder traversal
T_1
T_2
. . .
T_n
Step 2: Visit T_1 in preorder
Step 3: Visit T_2 in preorder
Step $n + 1$: Visit T_n in preorder

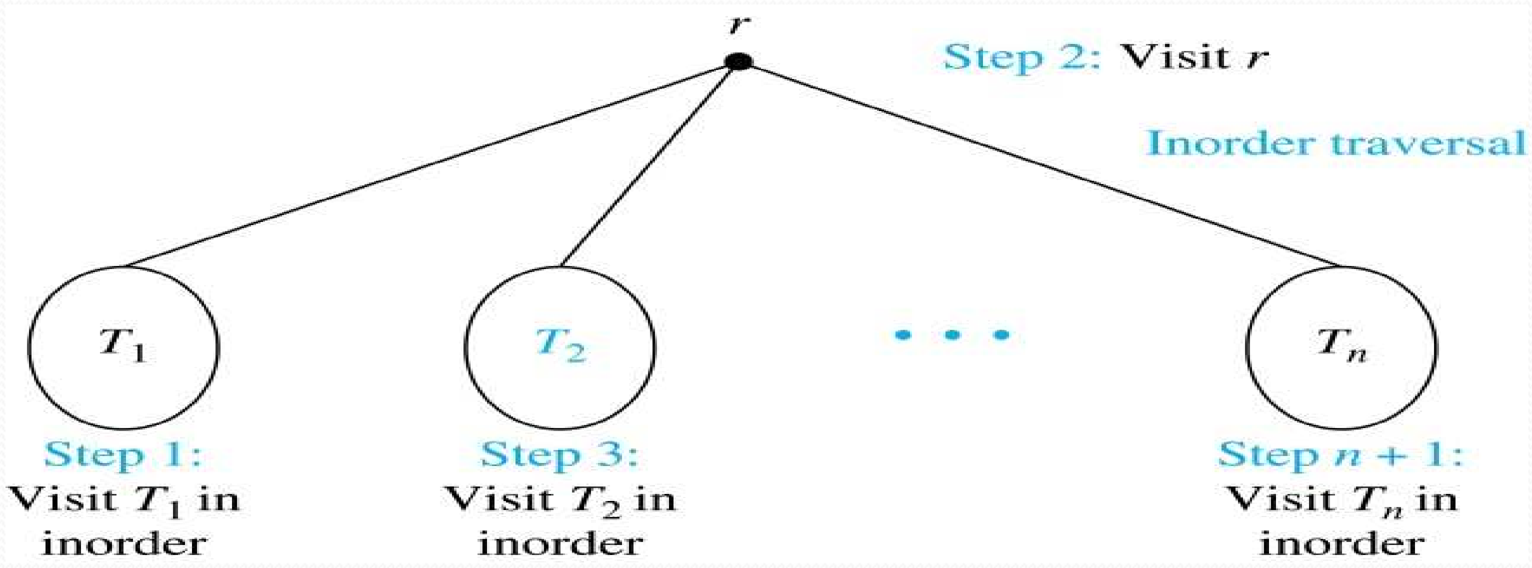
Preorder Traversal (*continued*)


Preorder traversal: Visit root, visit subtrees left to right

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

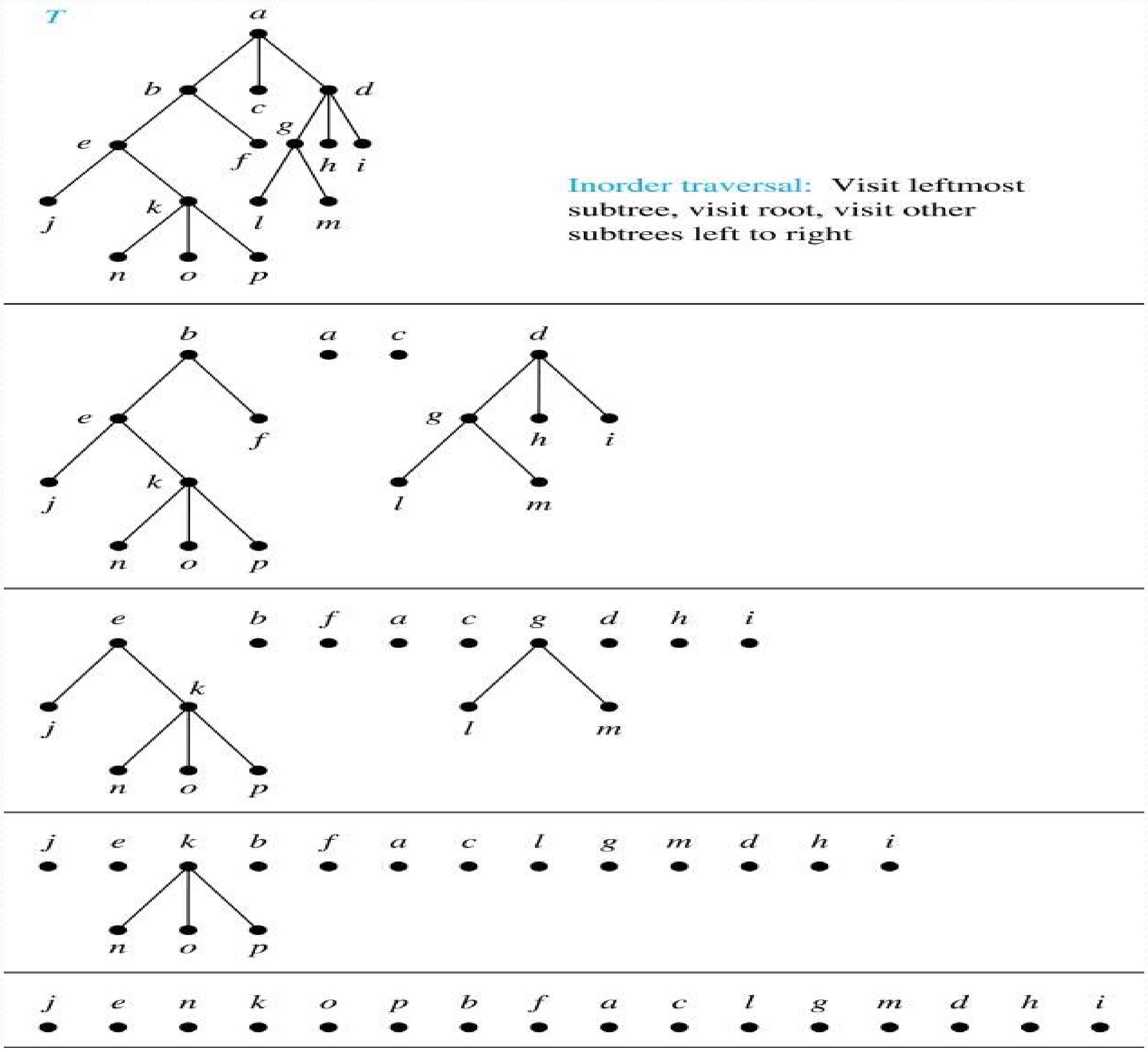
Inorder Traversal

Definition: 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, …, T_n are the subtrees of r from left to right in T. The inorder traversal begins by traversing T_1 in inorder, then visiting r, and continues by traversing T_2 in inorder, and so on, until T_n is traversed in inorder.


r
Step 2: Visit r
Inorder traversal
T_1
T_2
T_n
Step 1: Visit T_1 in inorder
Step 3: Visit T_2 in inorder
Step $n + 1$: Visit T_n in inorder

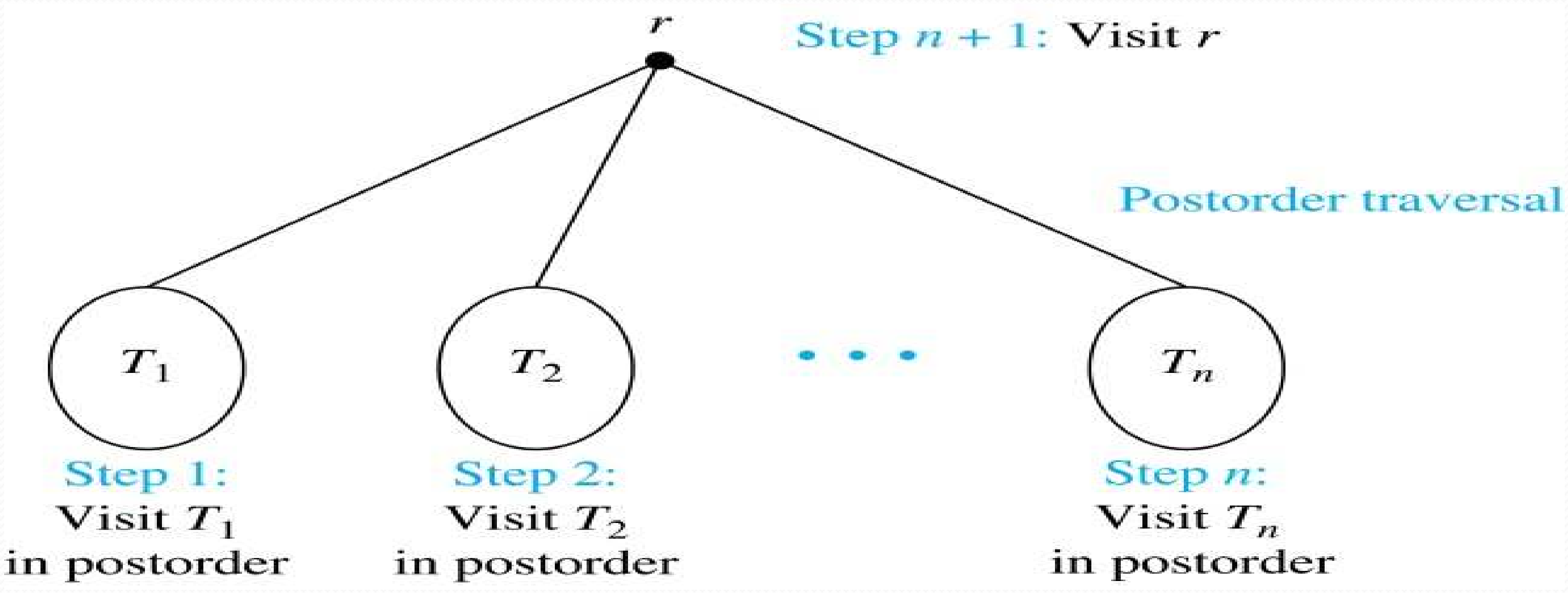
Inorder Traversal (*continued*)

```
procedure
 inorder (T: ordered rooted tree)
r := root of T
if r is a leaf then list r
else
    l := first child of r from left to right
    T(l) := subtree with l as its root
    inorder(T(l))
    list(r)
    for each child c of r from left to right
        T(c) := subtree with c as root
        inorder(T(c))
```

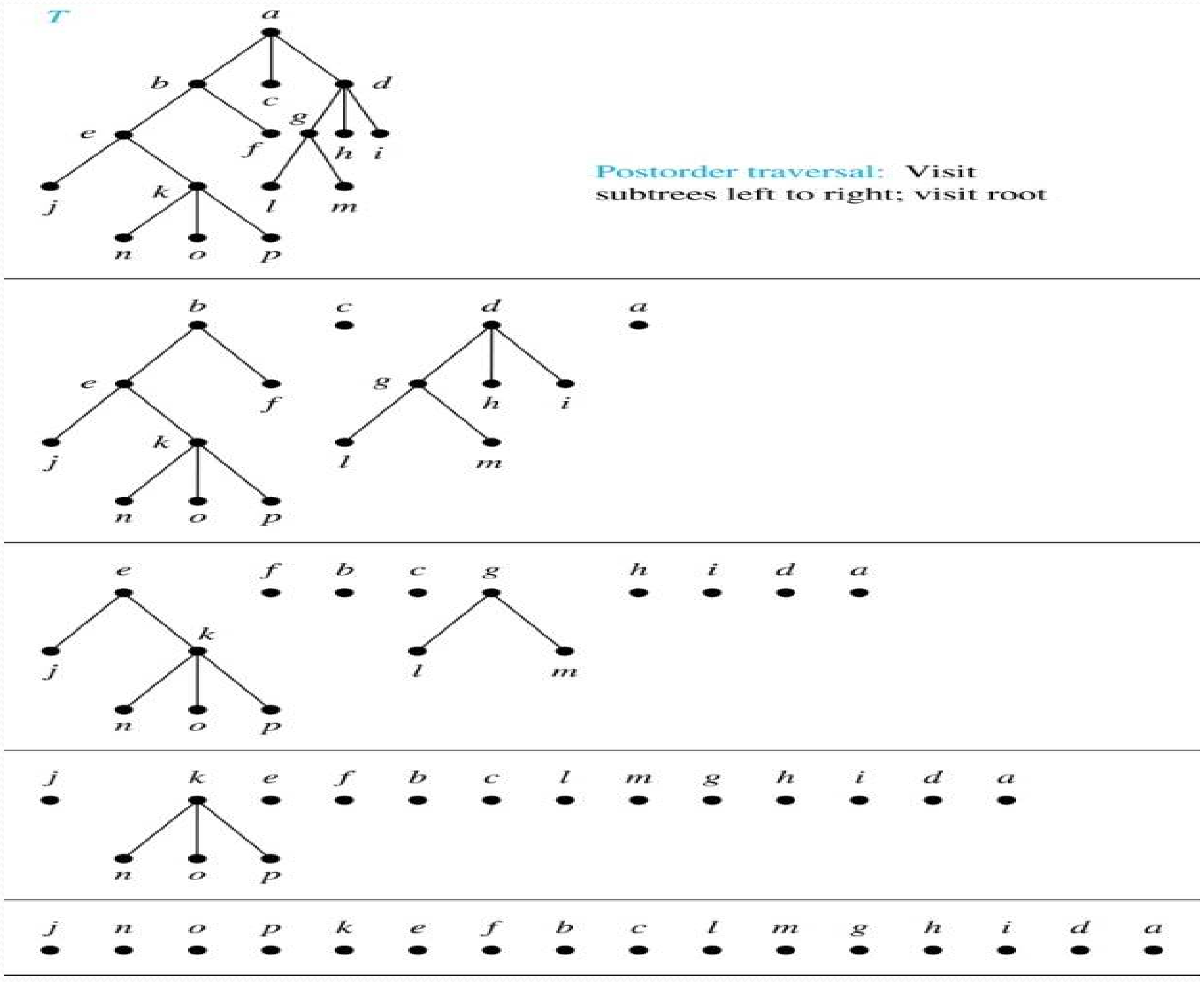

T
a b c d e f g h i j k l m n o p
Inorder traversal: Visit leftmost subtree, visit root, visit other subtrees left to right
j e n k o p b f a c l g m d h i

Postorder Traversal

Definition: 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, …, T_n are the subtrees of r from left to right in T. The postorder traversal begins by traversing T_1 in postorder, then T_2 in postorder, and so on, after T_n is traversed in postorder, r is visited.


r
Step $n+1$: Visit r
Postorder traversal
T_1
T_2
T_n
Step 1: Visit T_1 in postorder
Step 2: Visit T_2 in postorder
Step n: Visit T_n in postorder

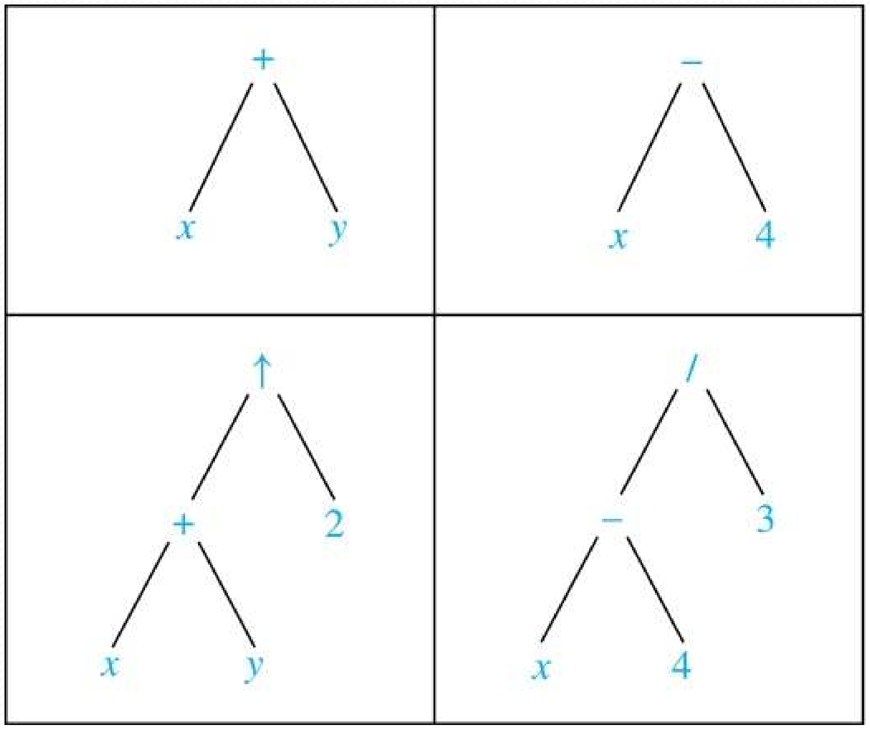
Post order Traversal *(continued)*

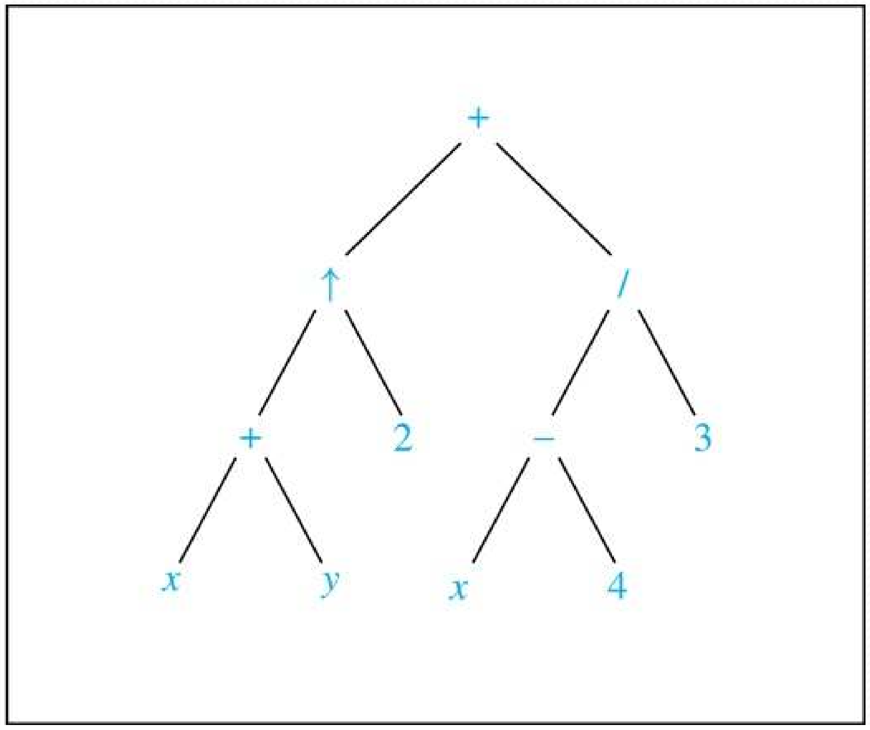

T
Postorder traversal: Visit subtrees left to right; visit root
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p

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

Expression Trees

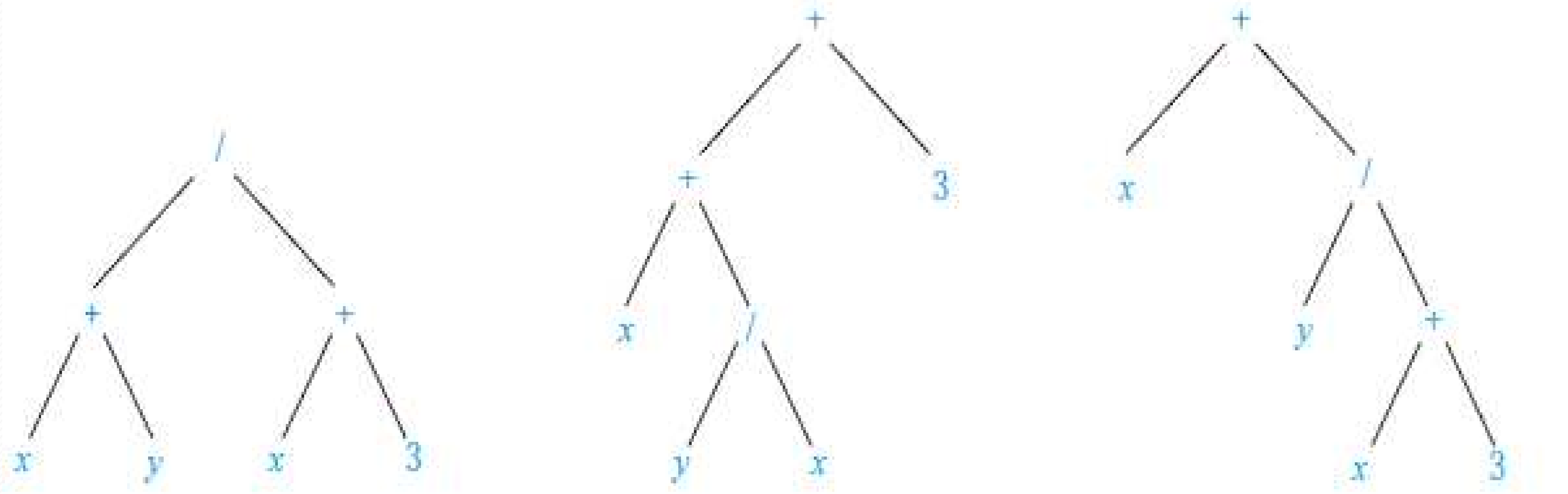
- Complex expressions can be represented using ordered rooted trees.
- Consider the expression $((x + y) \uparrow 2\ ) + ((x - 4)/3)$.
- A binary tree for the expression can be built from the bottom up, as is illustrated here.



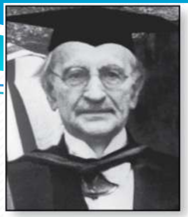


Infix Notation

- An inorder traversal of the tree representing an expression produces the original expression when parentheses are included except for unary operations, which now immediately follow their operands.
- We illustrate why parentheses are needed with an example that displays three trees all yield the same infix representation.



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


Jan Łukasiewicz
(1878-1956)

Prefix Notation

- When we traverse the rooted tree representation of an expression in preorder, we obtain the *prefix* form of the expression. Expressions in prefix form are said to be in *Polish notation*, named after the Polish logician Jan Łukasiewicz.
- Operators precede their operands in the prefix form of an expression. Parentheses are not needed as the representation is unambiguous.
- The prefix form of $((x + y) \uparrow 2) + ((x - 4)/3)$
  is $+ \uparrow + x\ y\ 2\ /\ - x\ 4\ 3$.
- Prefix expressions are evaluated by working from right to left. When we encounter an operator, we perform the corresponding operation with the two operations to the right.

Prefix Notation

- **Example**: We show the steps used to evaluate a particular prefix expression:

$$+ \quad - \quad * \quad 2 \quad 3 \quad 5 \quad / \quad \underbrace{\uparrow \quad 2 \quad 3}_{2 \uparrow 3 = 8} \quad 4$$

$$+ \quad - \quad * \quad 2 \quad 3 \quad 5 \quad \underbrace{/ \quad 8 \quad 4}_{8 / 4 = 2}$$

$$+ \quad - \quad \underbrace{* \quad 2 \quad 3}_{2 * 3 = 6} \quad 5 \quad 2$$

$$+ \quad \underbrace{- \quad 6 \quad 5}_{6 - 5 = 1} \quad 2$$

$$\underbrace{+ \quad 1 \quad 2}_{1 + 2 = 3}$$

Value of expression: 3

Postfix Notation

- We obtain the *postfix form* of an expression by traversing its binary trees in postorder. Expressions written in postfix form are said to be in *reverse Polish notation.*
- Parentheses are not needed as the postfix form is unambiguous.
- $x\ y + 2 \uparrow x\ 4 - 3\ / +$ is the postfix form of $((x + y) \uparrow 2\ ) + ((x - 4)/3)$.
- A binary operator follows its two operands. So, to evaluate an expression one works from left to right, carrying out an operation represented by an operator on its preceding operands.

Postfix Notation

- **Example**: We show the steps used to evaluate a particular postfix expression.

$$7\quad \underbrace{2\quad 3\quad *}_{2 * 3 = 6}\quad -\quad 4\quad \uparrow\quad 9\quad 3\quad /\quad +$$

$$\underbrace{7\quad 6\quad -}_{7 - 6 = 1}\quad 4\quad \uparrow\quad 9\quad 3\quad /\quad +$$

$$\underbrace{1\quad 4\quad \uparrow}_{1^4 = 1}\quad 9\quad 3\quad /\quad +$$

$$1\quad \underbrace{9\quad 3\quad /}_{9 / 3 = 3}\quad +$$

$$\underbrace{1\quad 3\quad +}_{1 + 3 = 4}$$

Value of expression: 4

Spanning Trees

Section 11.4

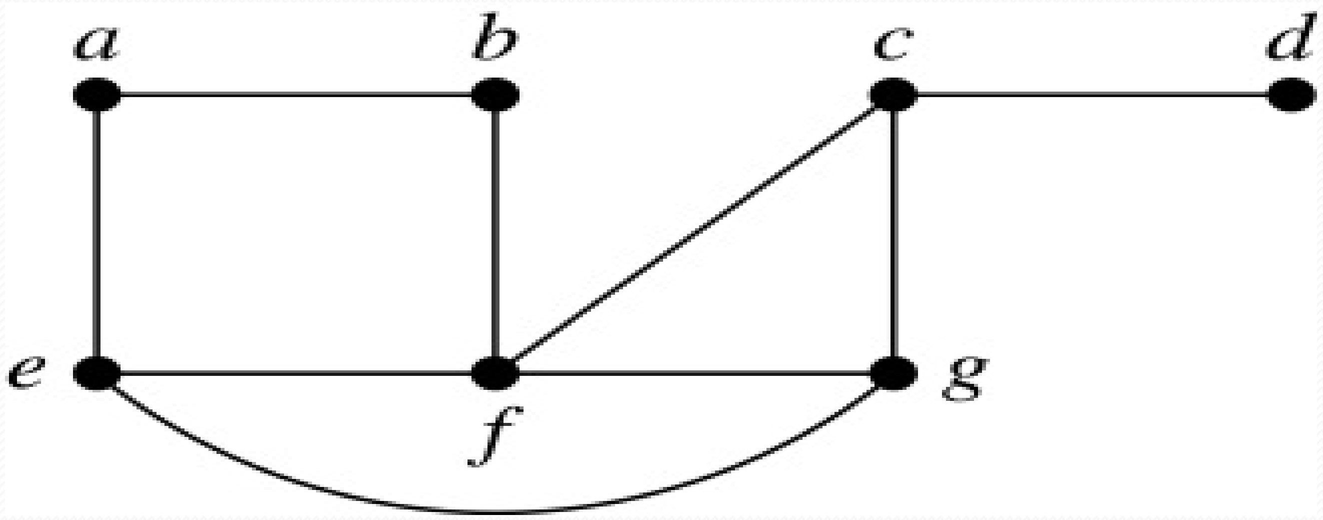
Section Summary

- Spanning Trees
- Prim's Algorithm
- Kruskal Algorithm

Spanning Trees

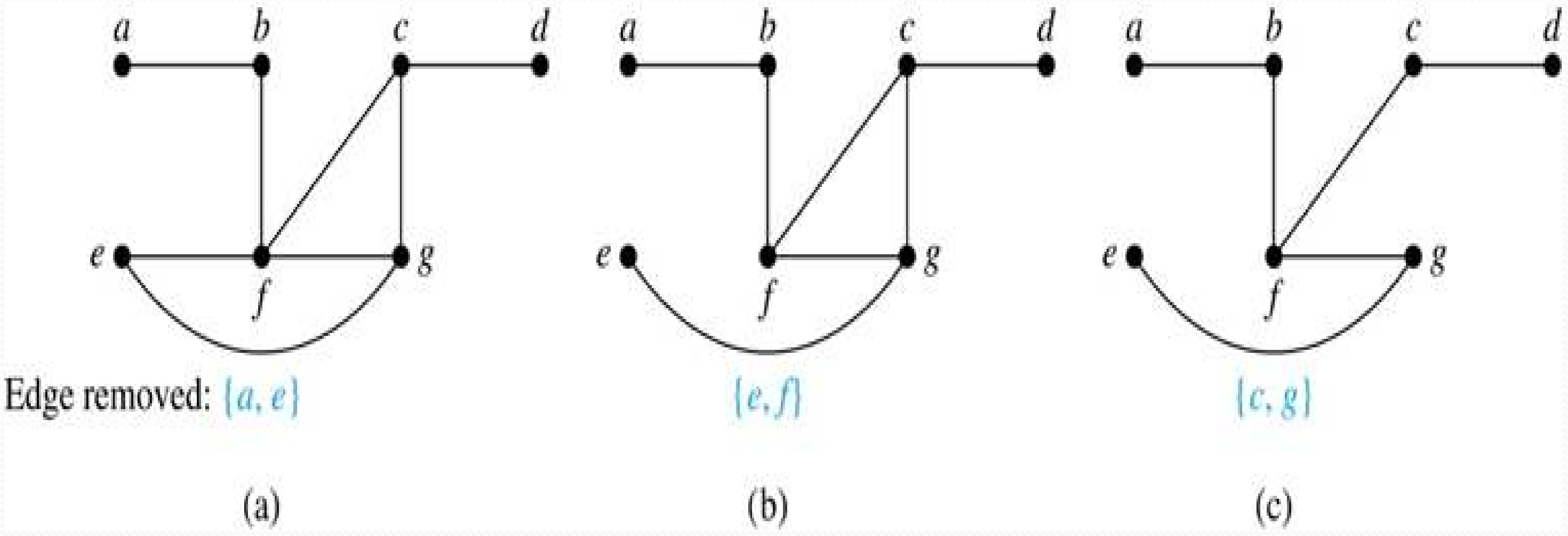
Definition: Let G be a simple graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G.

Example: Find the spanning tree of the simple graph:



Spanning Trees

Solution: The graph is connected, but is not a tree because it contains simple circuits. Remove the edge $\{a, e\}$. Now one simple circuit is gone, but the remaining subgraph still has a simple circuit. Remove the edge $\{e, f\}$ and then the edge $\{c, g\}$ to produce a simple graph with no simple circuits. It is a spanning tree, because it contains every vertex of the original graph.



Minimum Spanning

Section 11.5

Minimum spanning tree

- A *minimum spanning tree* in a connected weighted graph is a spanning tree that has the smallest possible sum of weights of its edges.
- **Example:** 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?

Minimum spanning tree

- **Solution:** 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**.

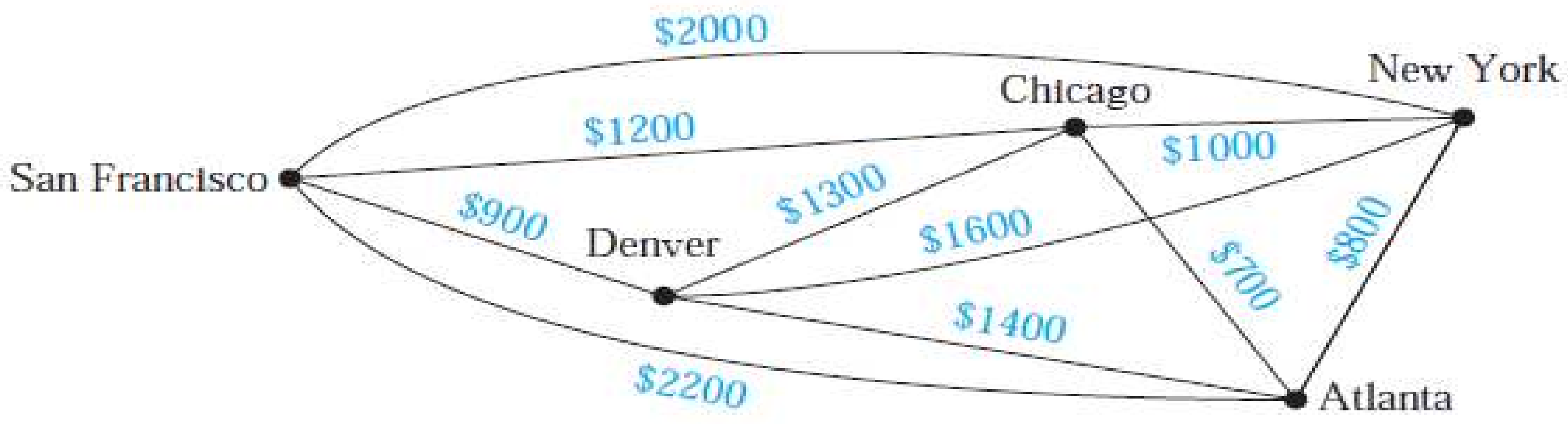
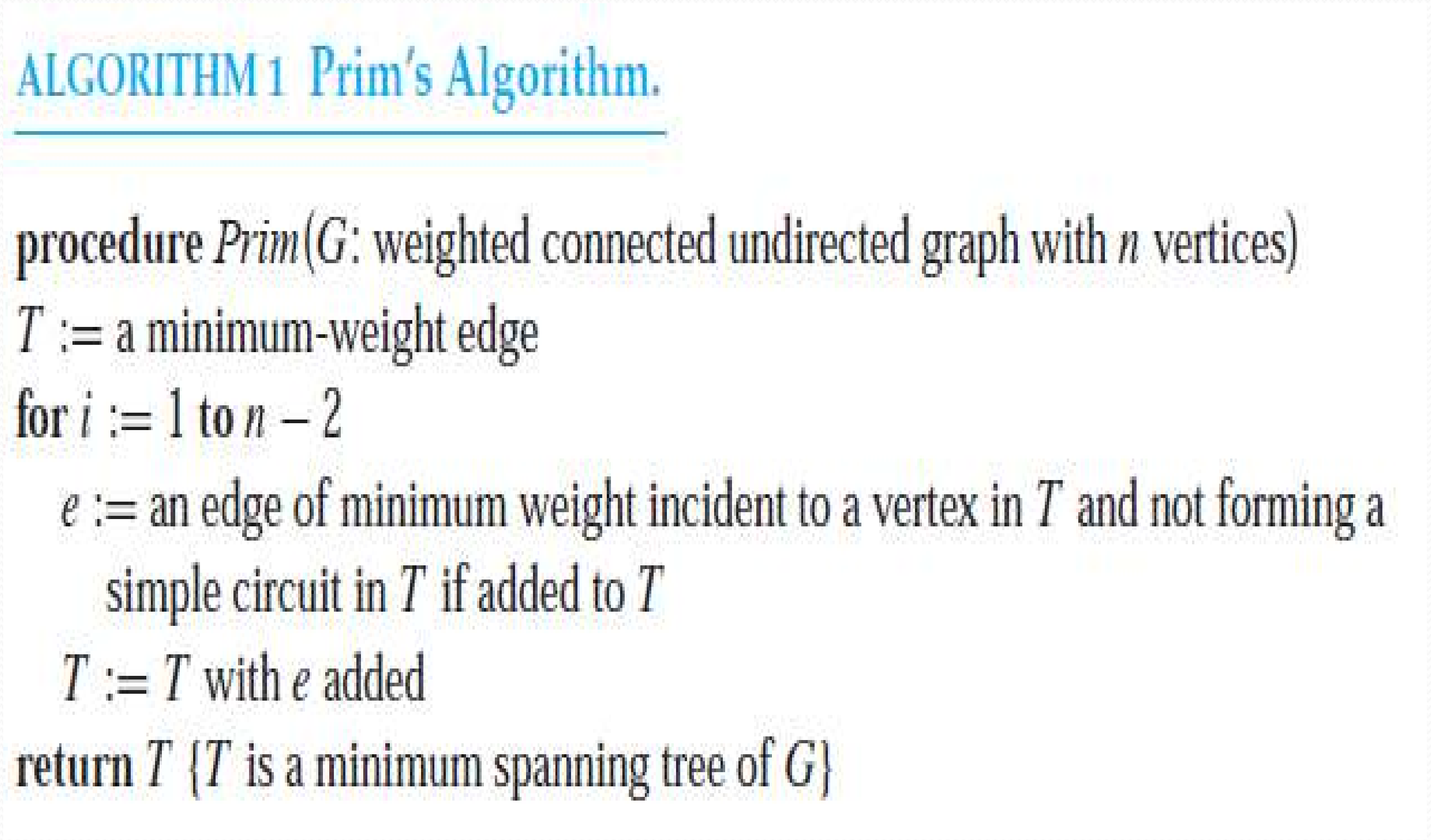


FIGURE 1 A Weighted Graph Showing Monthly Lease Costs for Lines in a Computer Network.

PRIM’S ALGORITHM

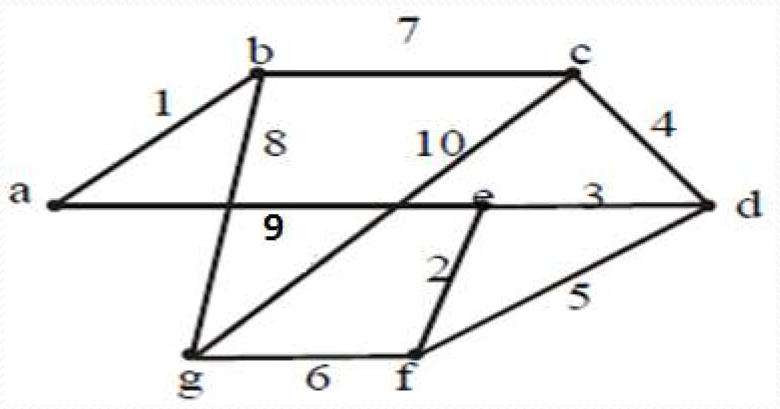


ALGORITHM 1 Prim's Algorithm.

```
procedure Prim($G$: weighted connected undirected graph with $n$ vertices)
$T$ := a minimum-weight edge
for $i$ := 1 to $n - 2$
    $e$ := 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
return $T$ {$T$ is a minimum spanning tree of $G$}
```

Minimal spanning tree (MST)

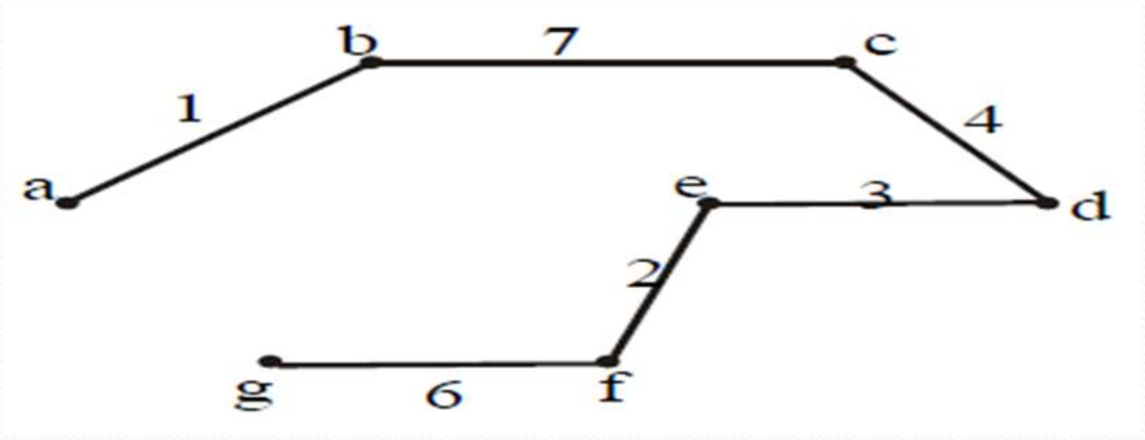
Example: Use Prims algorithm to find a minimal spanning tree for the graph below. Indicate the order in which edges are added to form the tree.



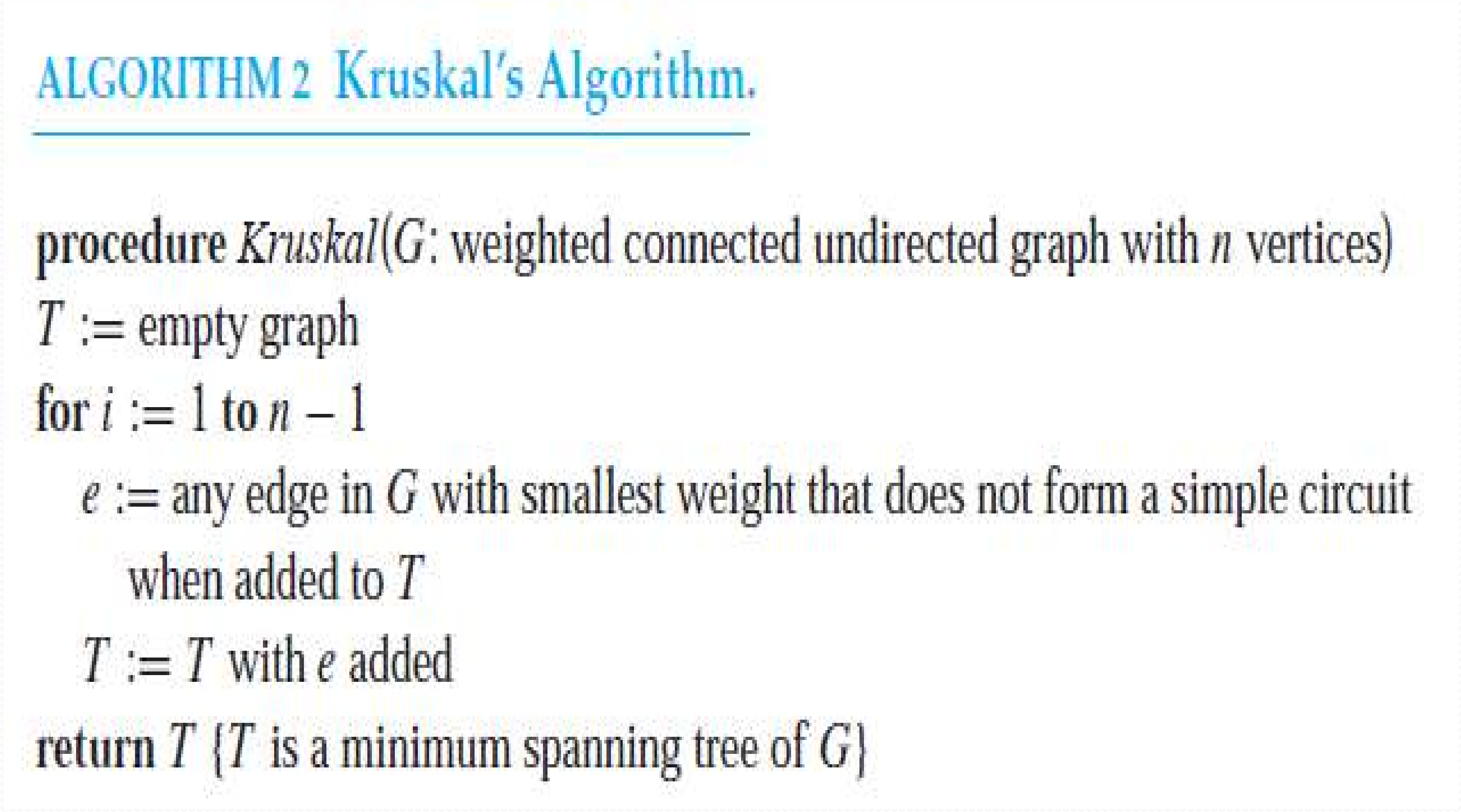
Order of adding the edges:

{a , b}, {b , c}, {c , d}, {d , e}, {e , f}, {f , g}

MST COST = 23



KRUSKAL’S ALGORITHM

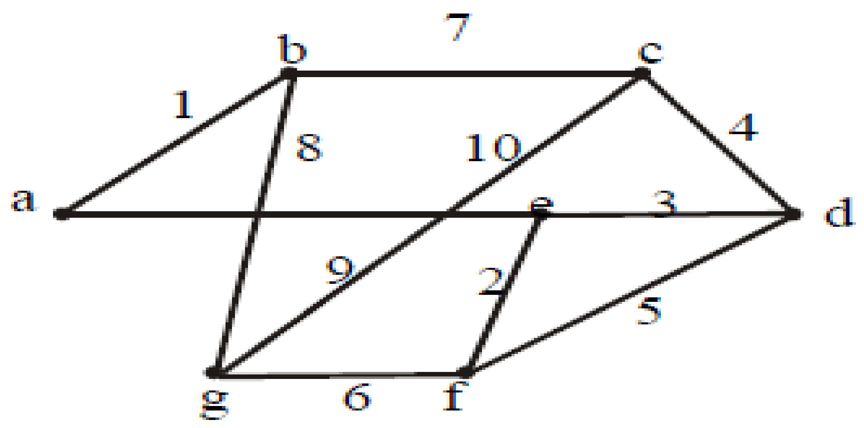


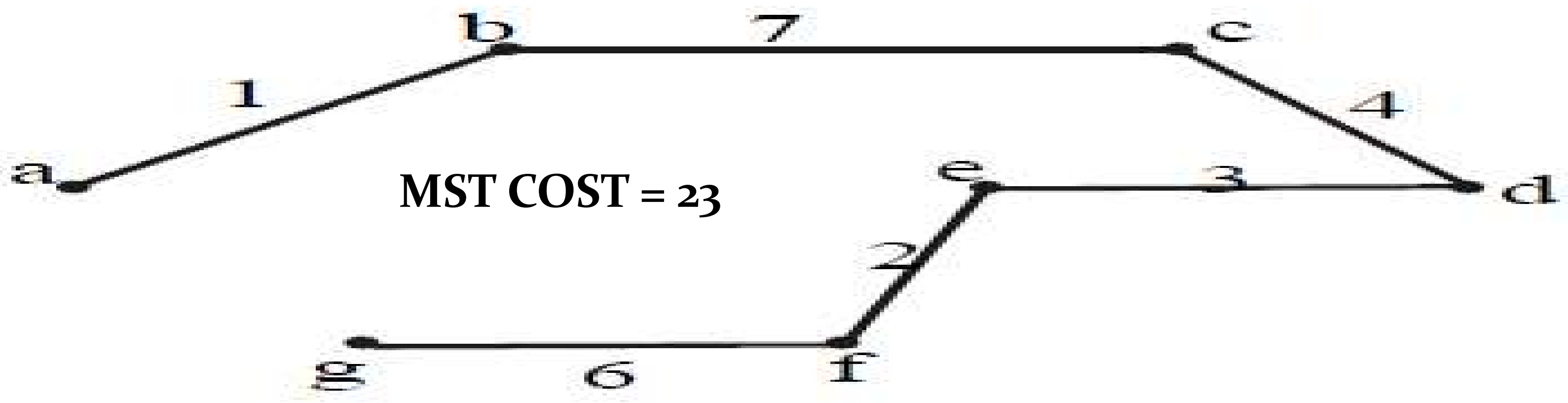
ALGORITHM 2 Kruskal's Algorithm.

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

Minimal spanning tree (MST)

Example: Use Kruskal's algorithm to find a minimal spanning tree for the graph below. Indicate the order in which edges are added to form the tree.





MST COST = 23

Order of adding the edges:
{a, b}, {e, f}, {e, d}, {c, d}, {g, f}, {b, c}