



# Trees



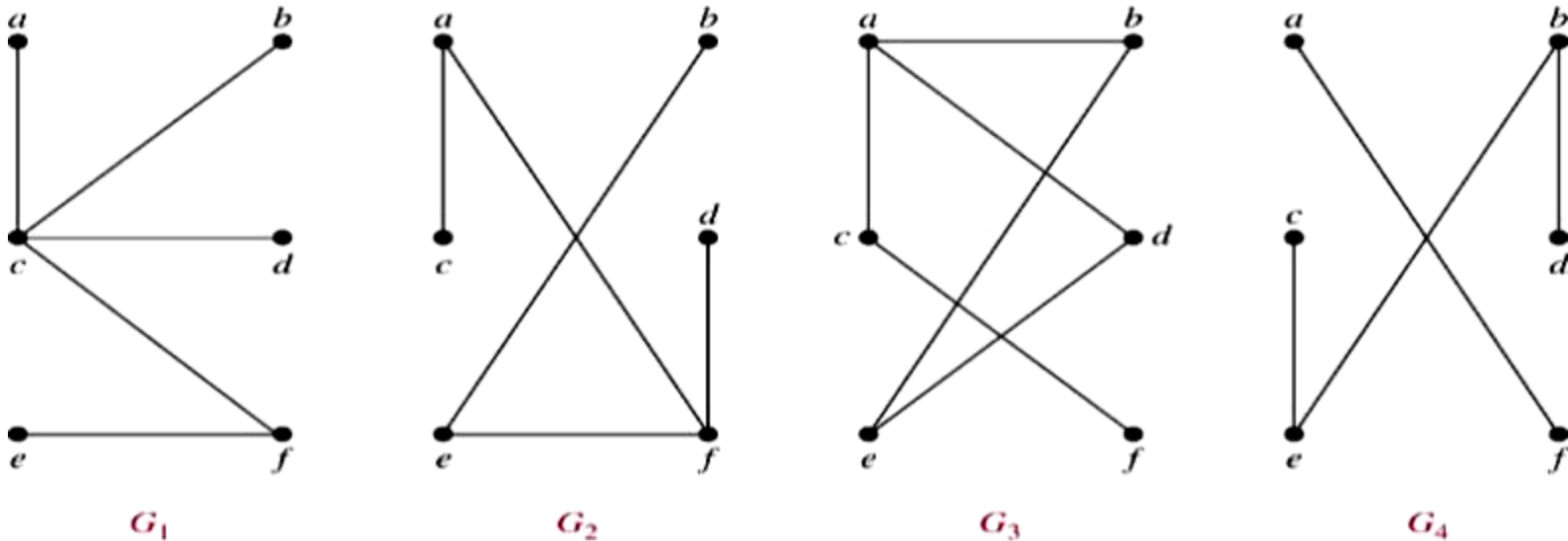
# Outline

- 10.1 Introduction to Trees
- 10.2 Applications of Trees
- 10.3 Tree Traversal
- 10.4 Spanning Trees
- 10.5 Minimal Spanning Trees

# 10.1 Introduction to Trees

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

**Example 1.** Which of the graphs are trees?



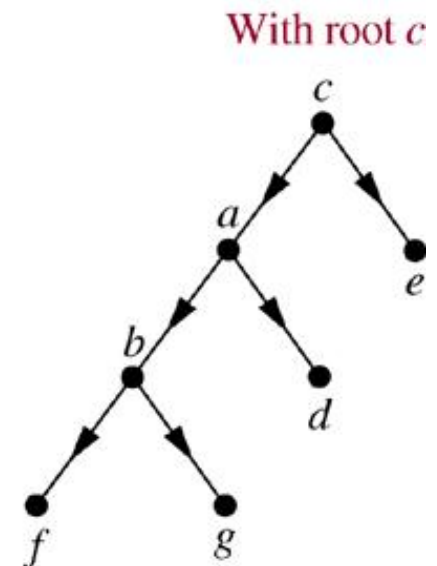
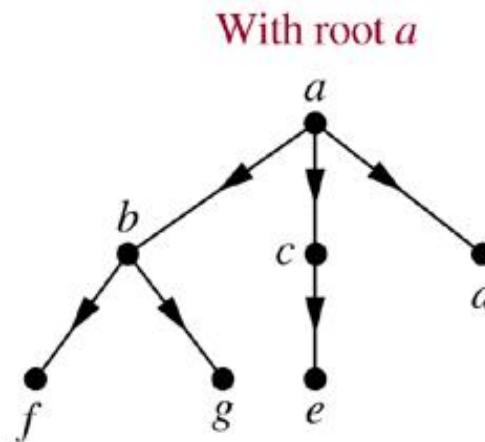
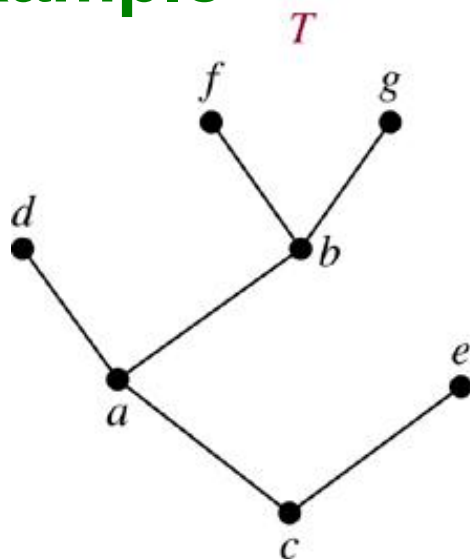
**Sol:**  $G_1, G_2$

**Note.** If connected condition is removed, it becomes forest

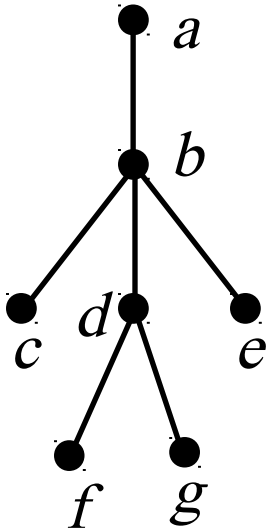
**Thm 1.** Any undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

**Def 2.** A **rooted tree** is a tree in which one vertex has been designed as the root and every edge is directed away from the root. (Arrow to disappear)

## Example

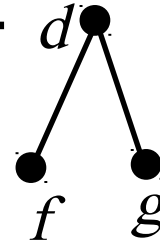


## Def:



$a$  is the **parent** of  $b$ ,  $b$  is the **child** of  $a$ ,  
 $c, d, e$  are **siblings**,  
 $a, b, d$  are **ancestors** of  $f$   
 $c, d, e, f, g$  are **descendants** of  $b$   
 $c, e, f, g$  are **leaves** of the tree (deg=1)  
 $a, b, d$  are **internal vertices** of the tree  
(at least one child)

**subtree** with  $d$  as its root:

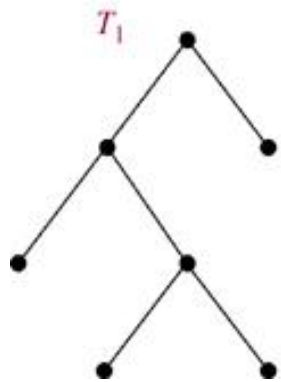


## Def:

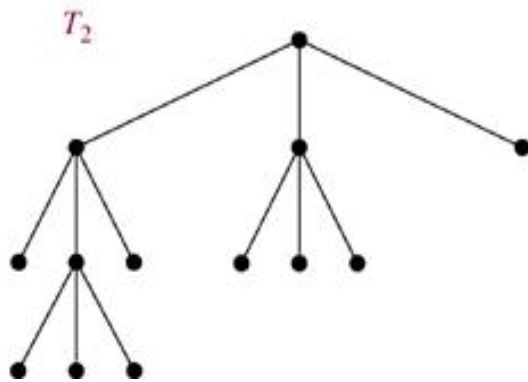
Vertices that have children are called **internal vertices**

**Def 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*.

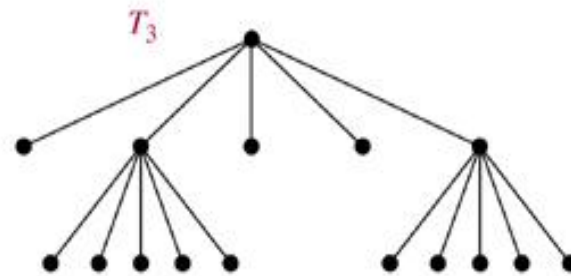
### Example 3



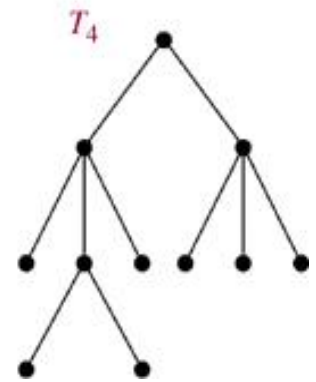
full binary tree



full 3-ary tree

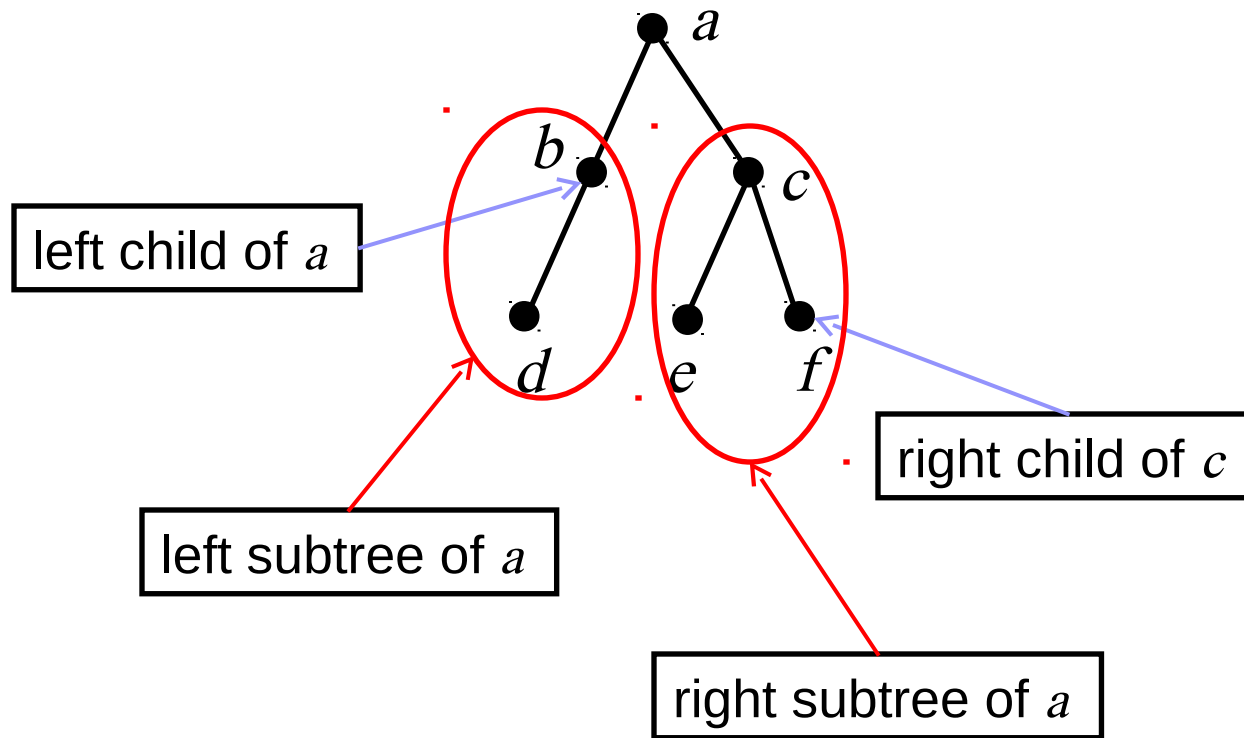


full 5-ary tree



not full 3-ary tree

Def:





## Properties of Trees

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



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

**Pf.** Every vertex, except the root, is the child of an internal vertex.

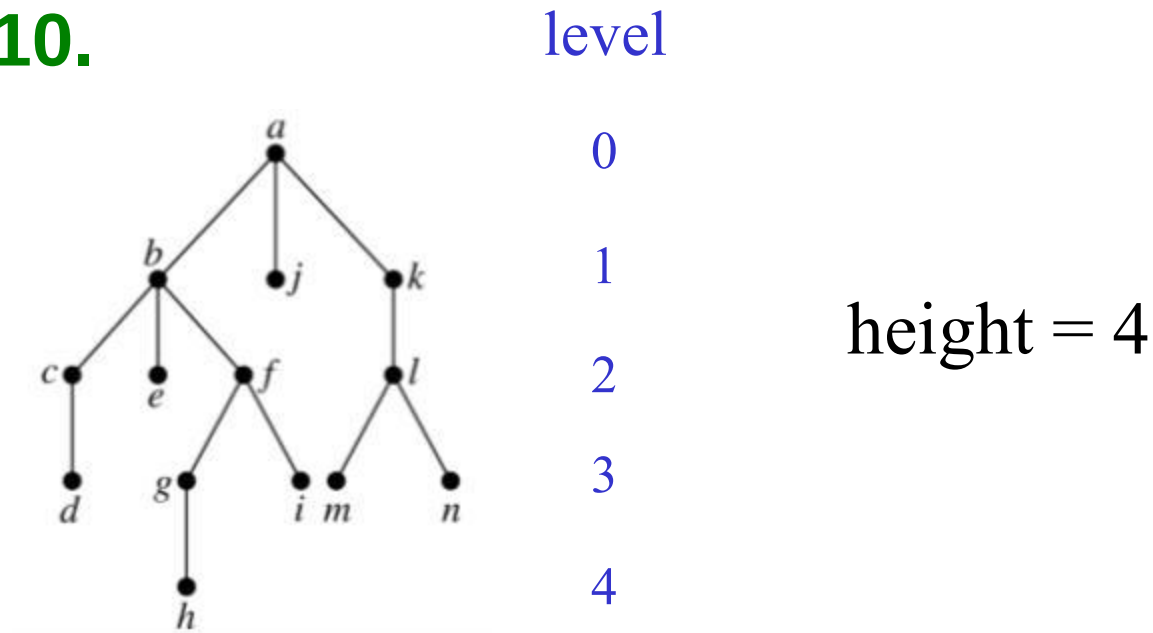
Each internal vertex has  $m$  children.

$\Rightarrow$  there are  $mi + 1$  vertices in the tree

**Cor.** A full  $m$ -ary tree with  $n$  vertices contains  $(n-1)/m$  internal vertices, and hence  $n - (n-1)/m = ((m-1)n+1)/m$  leaves.

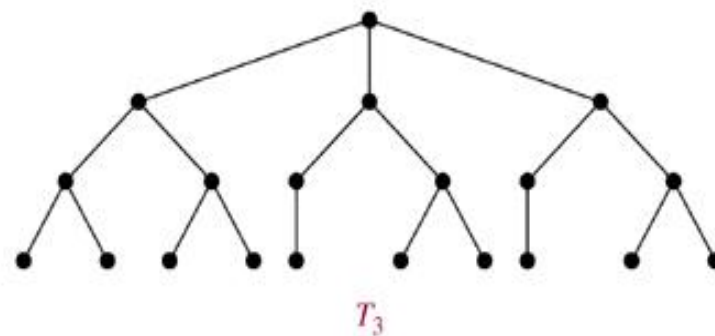
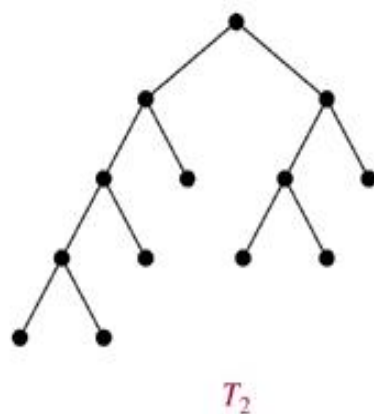
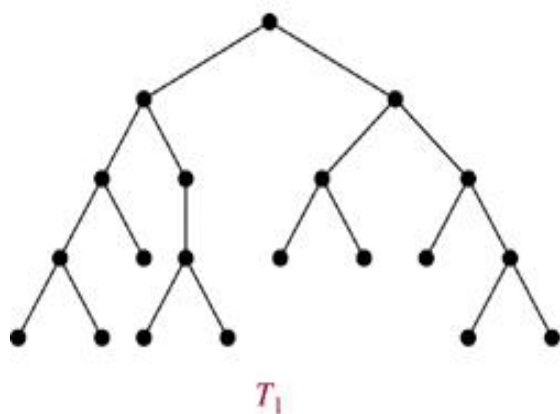
**Def:** The **level** of a vertex  $v$  in a rooted tree is the length of the unique path from the root to this vertex. The level of the root is defined to be zero. The **height** of a rooted tree is the maximum of the levels of vertices.

**Example 10.**




**Def:** 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 below are balanced?



**Sol.**  $T_1, T_3$

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



**Def:** A **complete  $m$ -ary tree** is a full  $m$ -ary tree, where every leaf is at the same level.

**Ex 28** How many vertices and how many leaves does a complete  $m$ -ary tree of height  $h$  have?

**Sol.**

$$\# \text{ of vertices} = 1 + m + m^2 + \dots + m^h = (m^{h+1} - 1) / (m - 1)$$

$$\# \text{ of leaves} = m^h$$



## 10.2 Applications of Trees

- ☐ **Binary Search Trees**
- ☐ **Decision Trees**
- ☐ **Prefix Codes**
- ☐ **Game Trees**

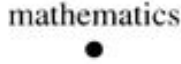
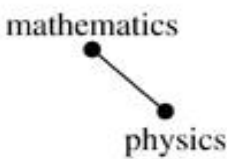
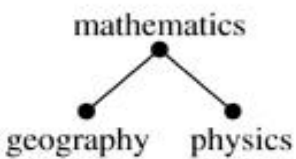
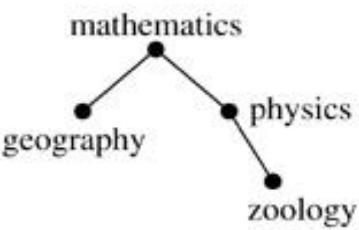
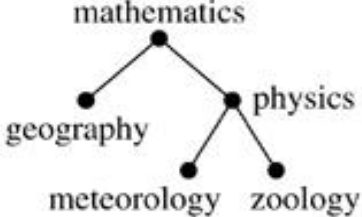
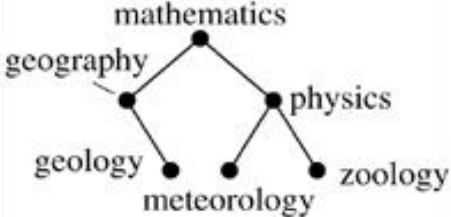
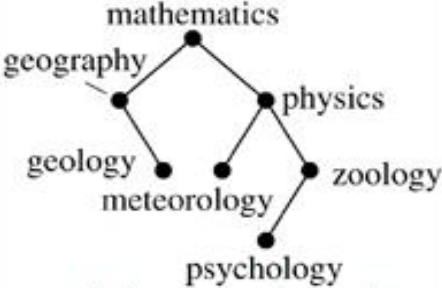
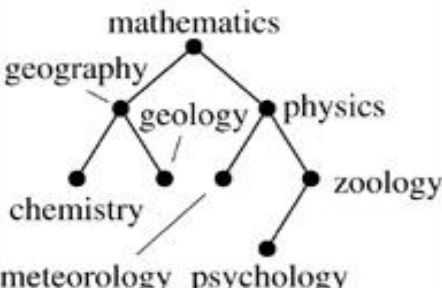
# Binary Search Trees

Goal: Implement a searching algorithm that finds items efficiently when the items are totally ordered.

**Binary Search Tree:** Binary tree + each child of a vertex is designed as a right or left child, and each vertex  $v$  is labeled with a key  $label(v)$ , which is one of the items.

**Note:**  $label(v) > label(w)$  if  $w$  is in the left subtree of  $v$   
and  $label(v) < label(w)$  if  $w$  is in the right subtree of  $v$

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

	 <p>physics &gt; mathematics</p>	 <p>geography &lt; mathematics</p>	 <p>zoology &gt; mathematics zoology &gt; physics</p>
 <p>meteorology &gt; mathematics meteorology &lt; physics</p>	 <p>geology &lt; mathematics geology &gt; geography</p>	 <p>psychology &gt; mathematics psychology &gt; physics psychology &lt; zoology</p>	 <p>chemistry &lt; mathematics chemistry &lt; geography</p>

# Algorithm 1 (Locating and Adding Items to a Binary Search

**Procedure** *insertion*( $T$ : binary search tree,  $x$ : item)

$v := \text{root of } T$

{a vertex not present in  $T$  has the value *null*}

**while**  $v \neq \text{null}$  and  $\text{label}(v) \neq x$

**begin**

**if**  $x < \text{label}(v)$  **then**

**if** left child of  $v \neq \text{null}$  **then**  $v := \text{left child of } v$

**else** add *new vertex* as a left child of  $v$  and set  $v := \text{null}$

**else**

**if** right child of  $v \neq \text{null}$  **then**  $v := \text{right child of } v$

**else** add *new vertex* as a right child of  $v$  and set  $v := \text{null}$

**end**

**if** root of  $T = \text{null}$  **then** add a vertex  $v$  to the tree and label it with  $x$

**else if**  $v$  is null or  $\text{label}(v) \neq x$  **then** label new vertex with  $x$  and

    let  $v$  be this new vertex

{  $v = \text{location of } x$ }



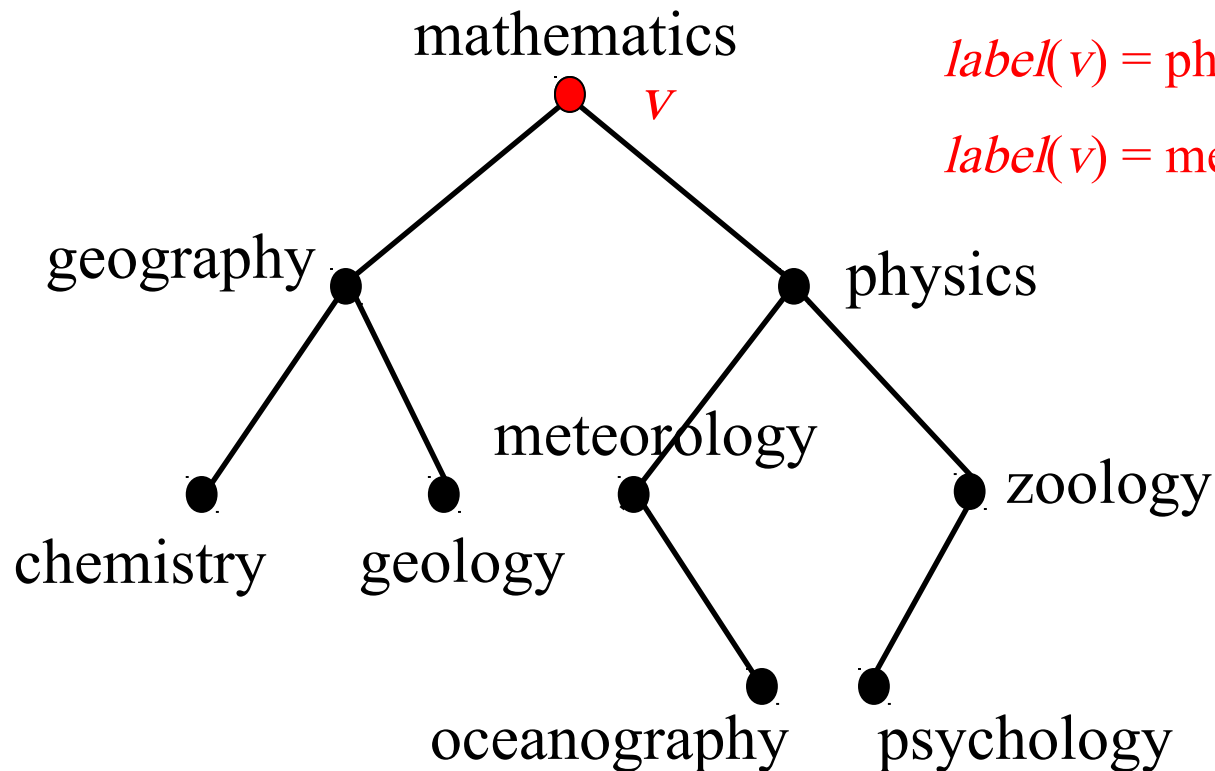
**Example 2** Use Algorithm 1 to insert the word *oceanography* into the binary search tree in Example 1.

**Sol.**

$label(v) = \text{mathematics} < \text{oceanography}$

$label(v) = \text{physics} > \text{oceanography}$

$label(v) = \text{meteorology} < \text{oceanography}$





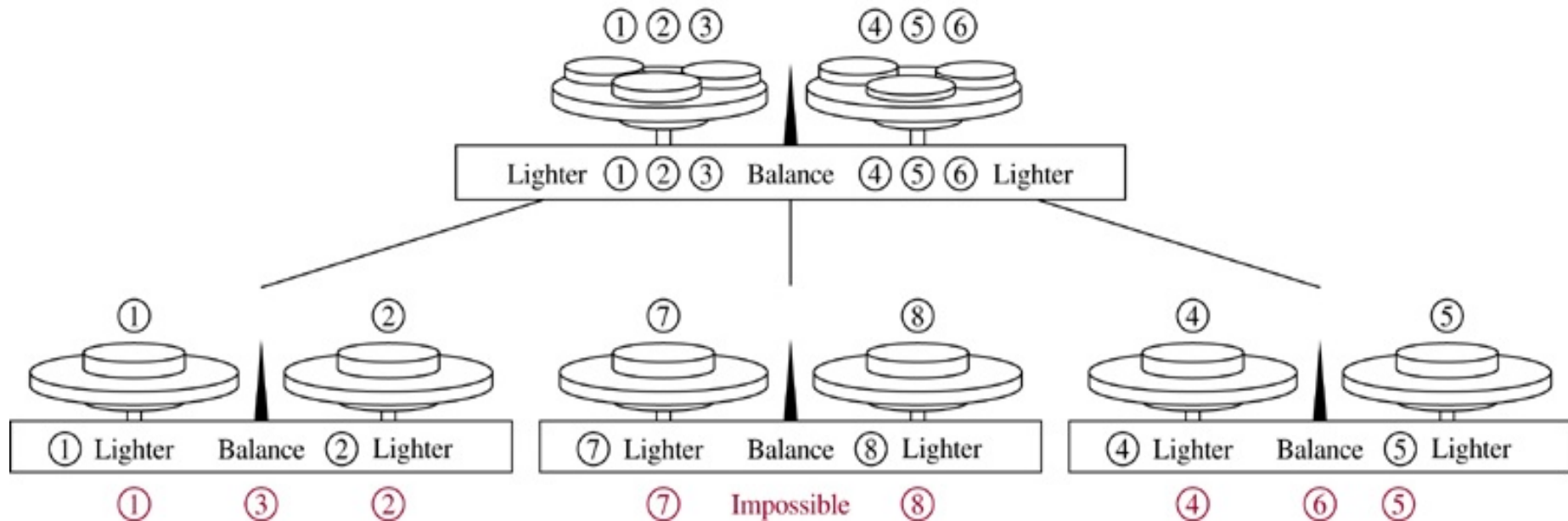
## Decision Trees

A rooted tree in which each internal vertex corresponds to a decision, with a subtree at these vertices for each possible outcome of the decision, is called **a decision tree**.

**Example 3** Suppose there are seven coins, all with the same weight, and a counterfeit coin that weights less than the others. How many weighings are necessary using a balance scale to determine which of the eight coins is the counterfeit one? Give an algorithm for finding this counterfeit coin.

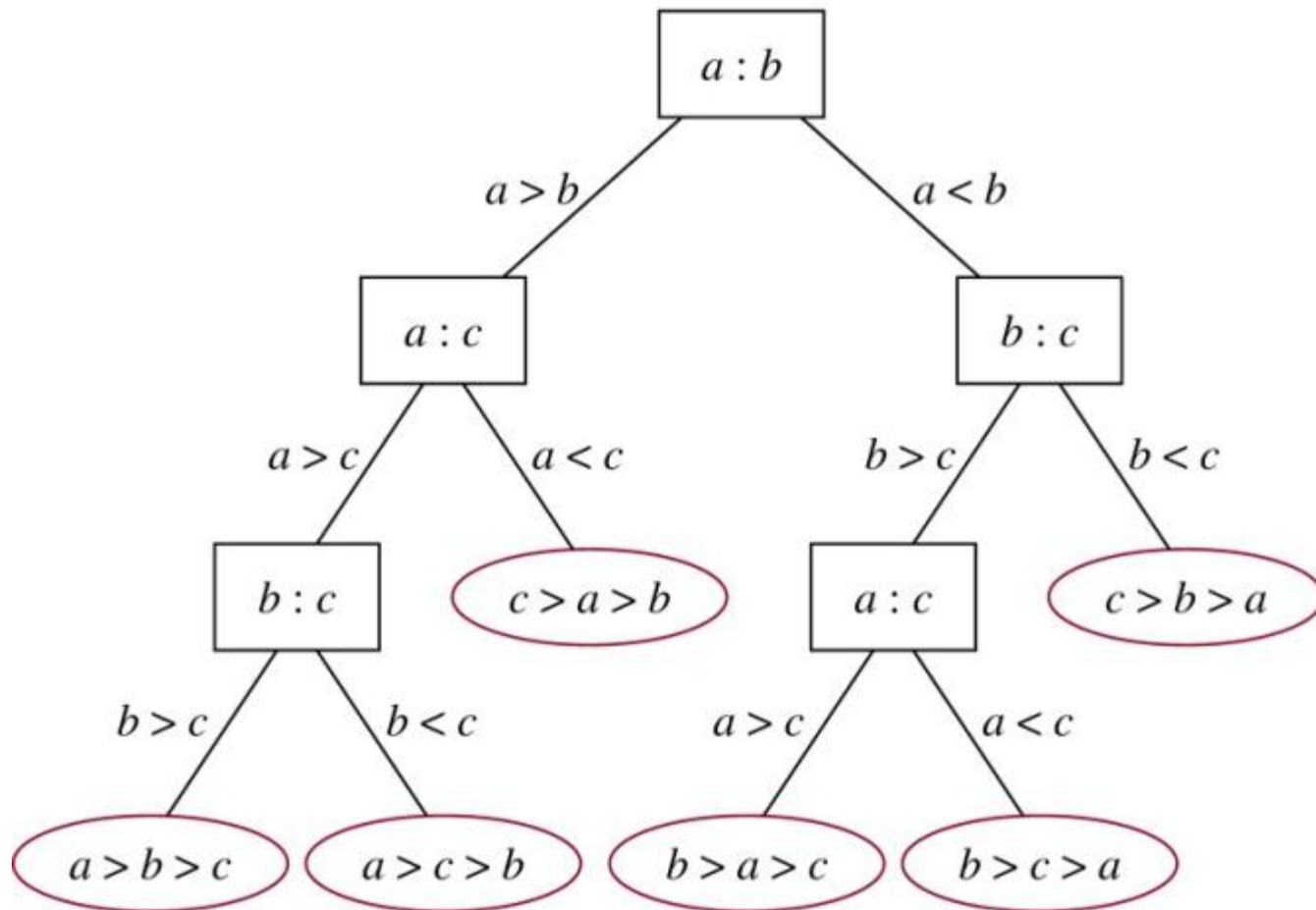
**Sol.** When weighing, it is either the two pans can have equal weight or the first pan can be heavier, or the second pan can be heavier  $\Rightarrow$  3-ary tree

Need 8 leaves  $\Rightarrow$  Need to weigh at least twice



**Example 4** A decision tree that orders the elements of the list  $a, b, c$ .

**Sol.**



# Prefix Codes

**Problem:** Using bit strings to encode the letter of the English alphabet (Not case sensitive)

- ⇒ each letter needs a bit string of length 5 (Because  $2^4 < 26 < 2^5$ )
- ⇒ Is it possible to find a coding scheme of these letter such that when data are coded, fewer bits are used?
- ⇒ Encode letters using varying numbers of bits.
- ⇒ Some methods must be used to determine where the bits for each character start and end.
- ⇒ **Prefix codes:** Codes with the property that the bit string for a letter never occurs as the first part of the bit string for another letter.

## Example: (not prefix code)

$e : 0, \quad a : 1, \quad t : 01$

The string 0101 could correspond to *eat*, *tea*, *eaea*, or *tt*.

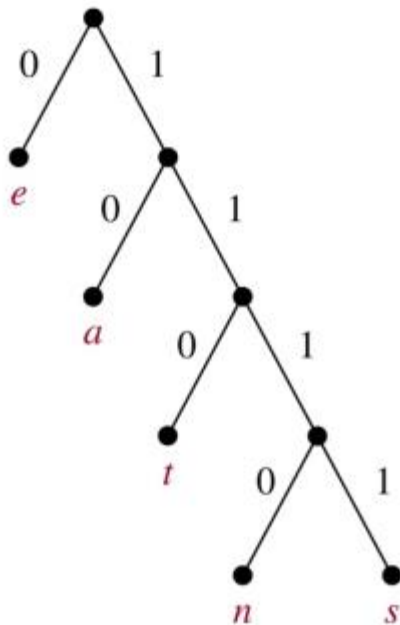
## Example: (prefix code)

$e : 0, \quad a : 10, \quad t : 11$

The string 10110 is the encoding of *ate*.

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.

## Walking from the root to the leaf repeatedly


$$\begin{array}{l} e : 0 \\ a : 10 \\ t : 110 \\ n : 1110 \\ s : 1111 \end{array}$$

11111011100  
 s a n e

$\Rightarrow$  *sane*



## Huffman Coding (data compression)

Input the frequencies of symbols in a string and output a prefix code that encodes the string using the fewest possible bits, among all possible binary prefix codes for these symbols.

Start with a lot of isolated points and label is the symbol,  
Use at least two symbol into a subtree,  
Repeat this concept  
Using at least two of the subtrees into a subtree, ...



## Algorithm 2 (Huffman Coding)

**Procedure** *Huffman*( $C$ : symbols  $a_i$  with frequencies  $w_i$ ,  $i = 1, \dots, n$ )

$F :=$  forest of  $n$  rooted trees, each consisting of the single vertex  $a_i$   
and assigned weighted  $w_i$

**while**  $F$  is not a tree

**begin**

Replace the rooted trees  $T$  and  $T'$  of least weights from  $F$  with  
 $w(T) \geq w(T')$  with a tree having a new root that has  $T$  as its  
left subtree and  $T'$  as its right subtree. Label the new edge to  $T$   
with 0 and the new edge to  $T'$  with 1.

Assign  $w(T) + w(T')$  as the weight of the new tree.

**end**

**Example 5** Use Huffman coding to encode the following symbols with the frequencies listed:

A: 0.08, B: 0.10, C: 0.12, D: 0.15, E: 0.20, F: 0.35.

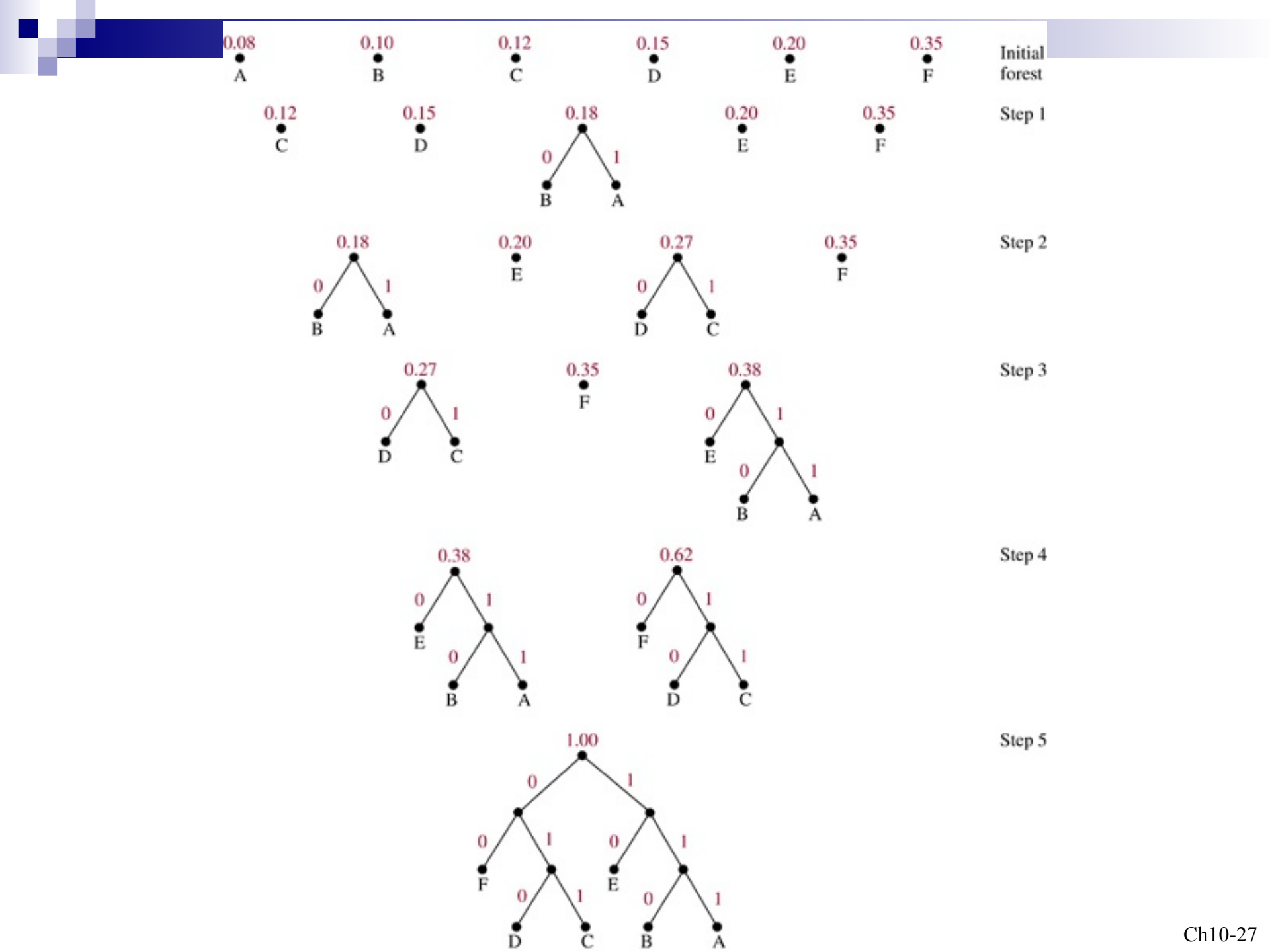
What is the average number of bits used to encode a character?

**Sol:**

1. Next map

2. The average number of bits is:

$$\begin{aligned} & \text{Each symbol total length} \times \text{frequency} \\ &= 3 \times 0.08 + 3 \times 0.10 + 3 \times 0.12 + 3 \times 0.15 + 2 \times 0.20 + 2 \times 0.35 \\ &= 2.45 \end{aligned}$$



## 10.3 Tree Traversal

We need procedures for visiting each vertex of an ordered rooted tree to access data.

### Universal Address Systems

Label vertices:

1. root  $\rightarrow 0$ , its  $k$  children  $\rightarrow 1, 2, \dots, k$  (from left to right)
2. For each vertex  $v$  at level  $n$  with label  $A$ , its  $r$  children  $\rightarrow A.1, A.2, \dots, A.r$  (from left to right).

We can **totally order** the vertices using the lexicographic ordering of their labels in the universal address system.

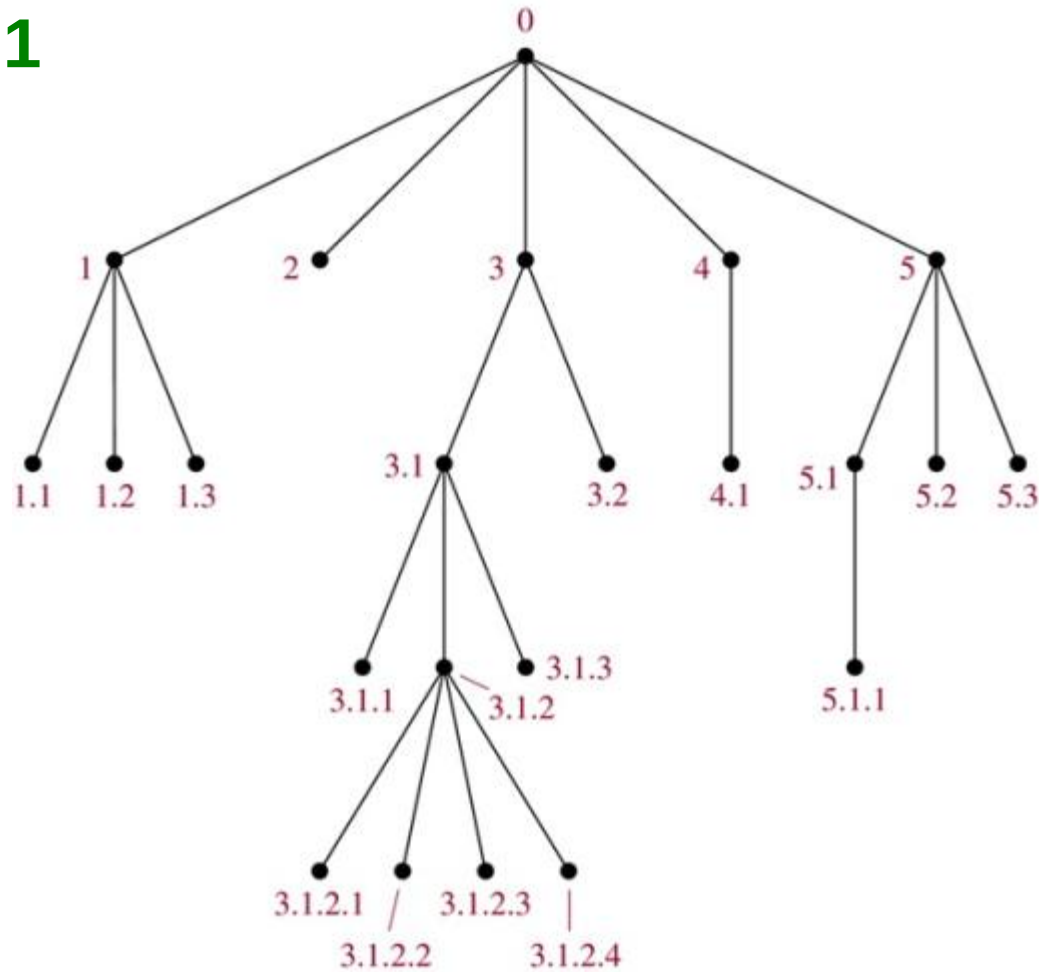
$$x_1.x_2.\dots.x_n < y_1.y_2.\dots.y_m$$

if there is an  $i$ ,  $0 \leq i \leq n$ , with  $x_1=y_1, x_2=y_2, \dots, x_{i-1}=y_{i-1}$ , and

$$x_i < y_i;$$

or if  $n < m$  and  $x_i=y_i$  for  $i=1, 2, \dots, n$ .

## Example 1

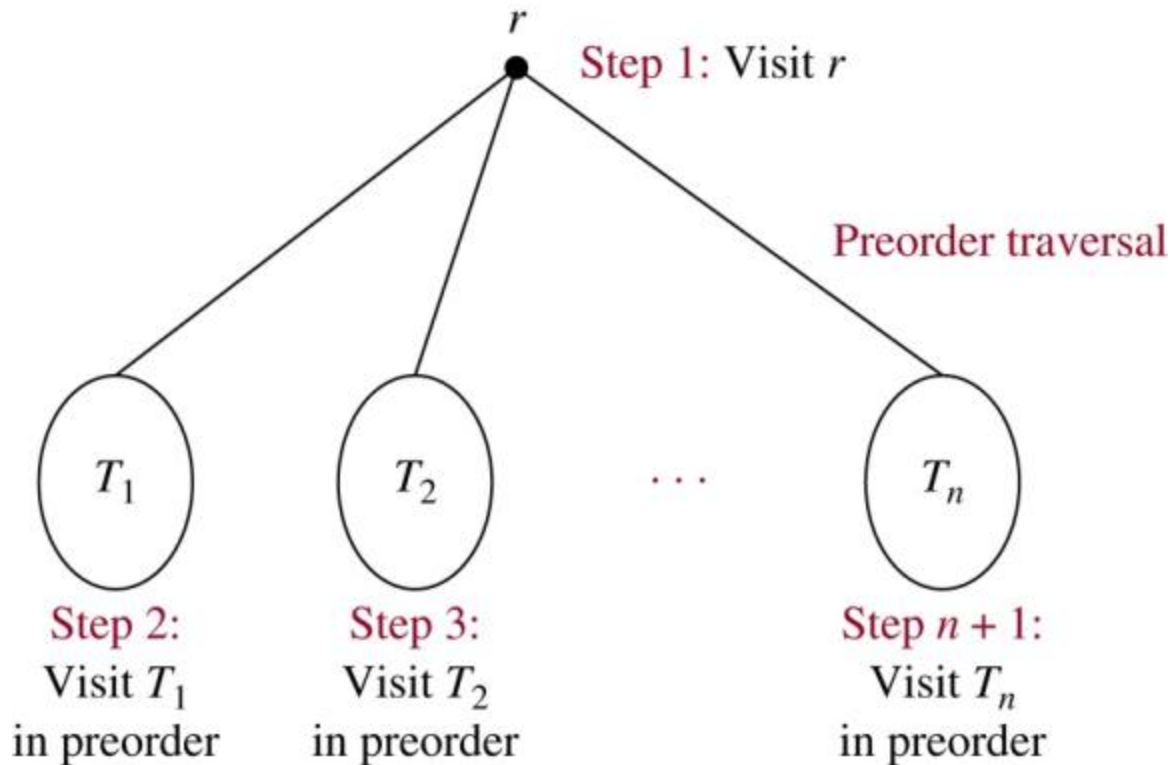


The lexicographic ordering is:

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

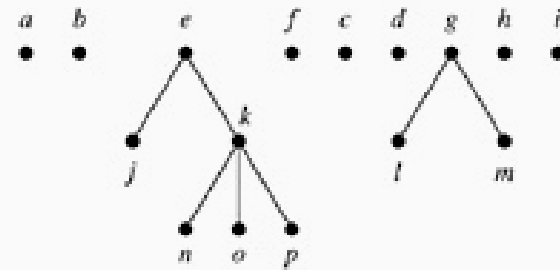
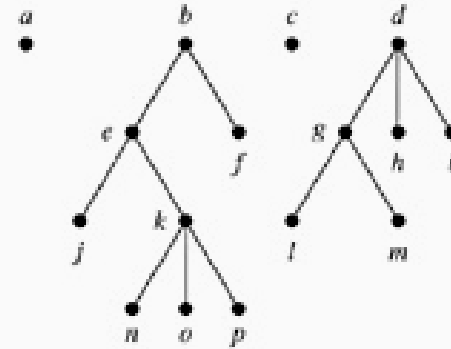
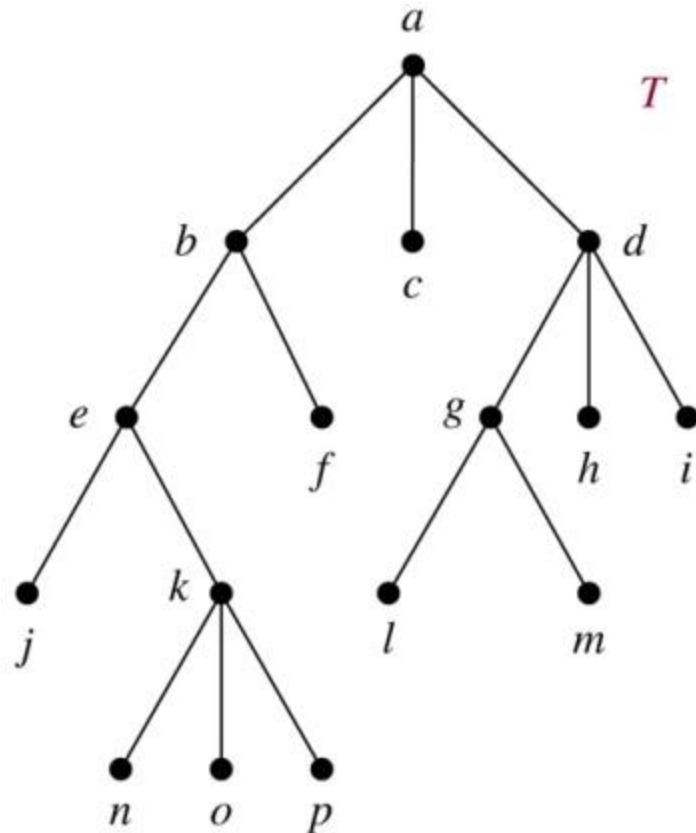
# Traversal Algorithms

## Preorder traversal (Preorder)



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

**Sol:**

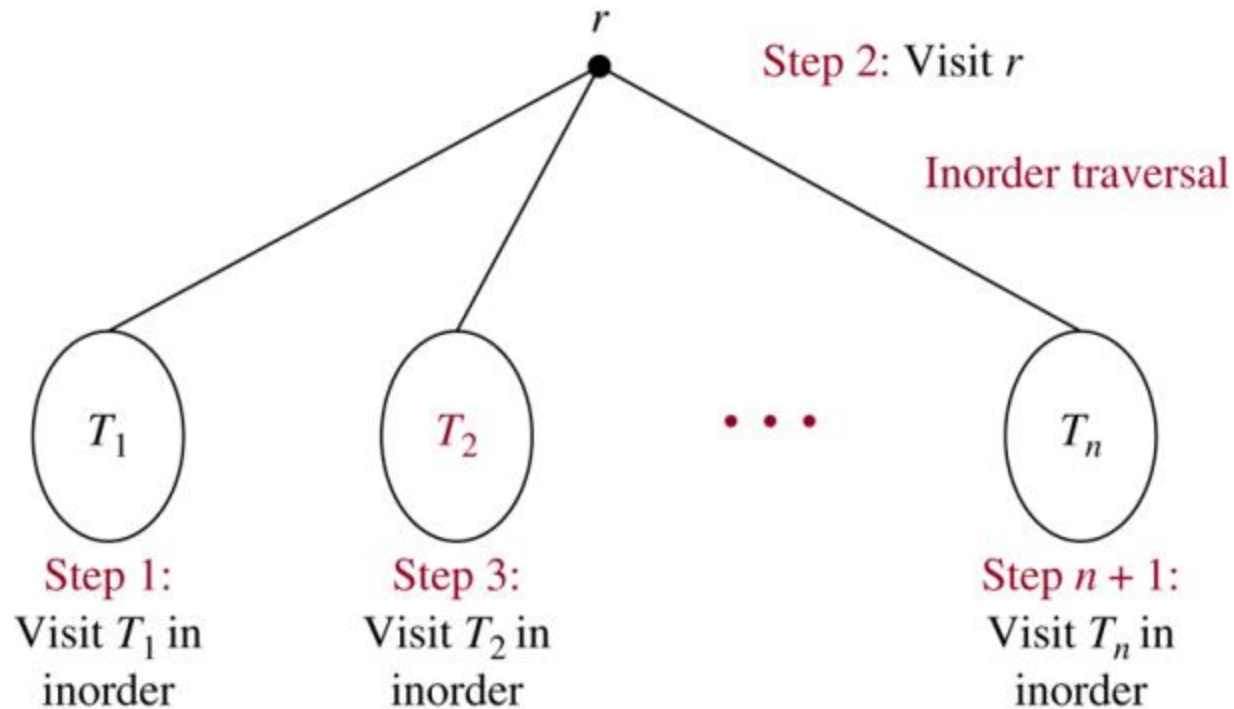


# Algorithm 1 (Preorder Traversal)

```
Procedure preorder( $T$ : ordered rooted tree)
 $r := \text{root of } T$ 
list  $r$ 
for each child  $c$  of  $r$  from left to right
begin
     $T(c) := \text{subtree with } c \text{ as its root}$ 
    preorder( $T(c)$ )
end
```

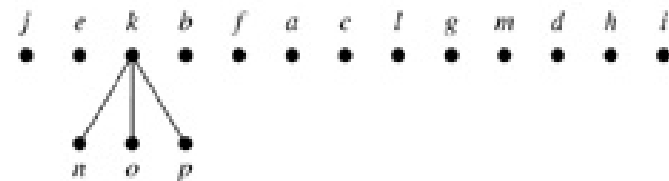
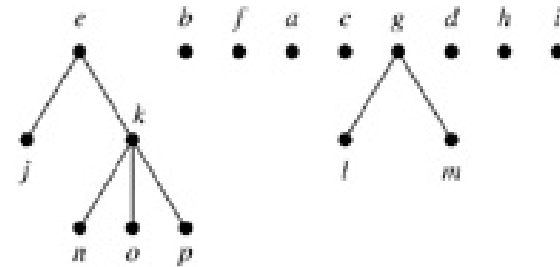
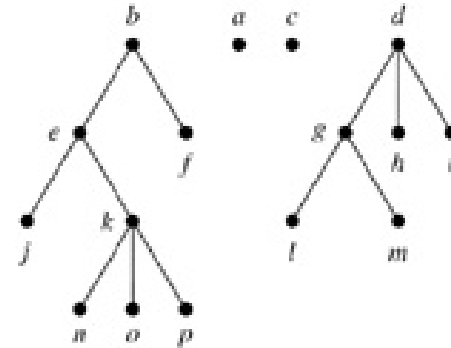
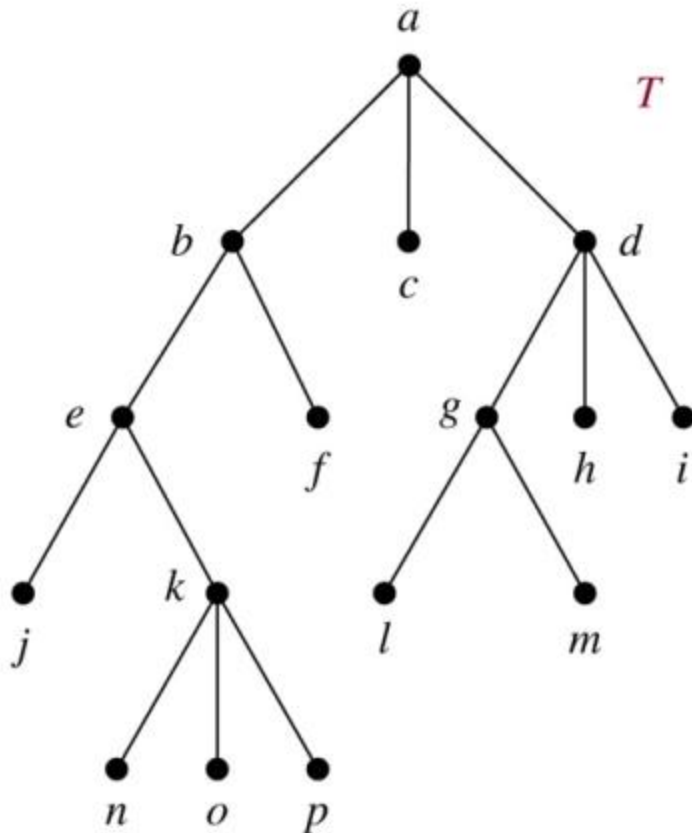


## Inorder traversal(In the sequence)



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

**Sol:**



## Algorithm 2 (Inorder Traversal)

**Procedure** *inorder*( $T$ : ordered rooted tree)

$r := \text{root of } T$

**If**  $r$  is a leaf **then** list  $r$

**else**

**begin**

$l := \text{first child of } r \text{ from left to right}$

$T(l) := \text{subtree with } l \text{ as its root}$

*inorder*( $T(l)$ )

    list  $r$

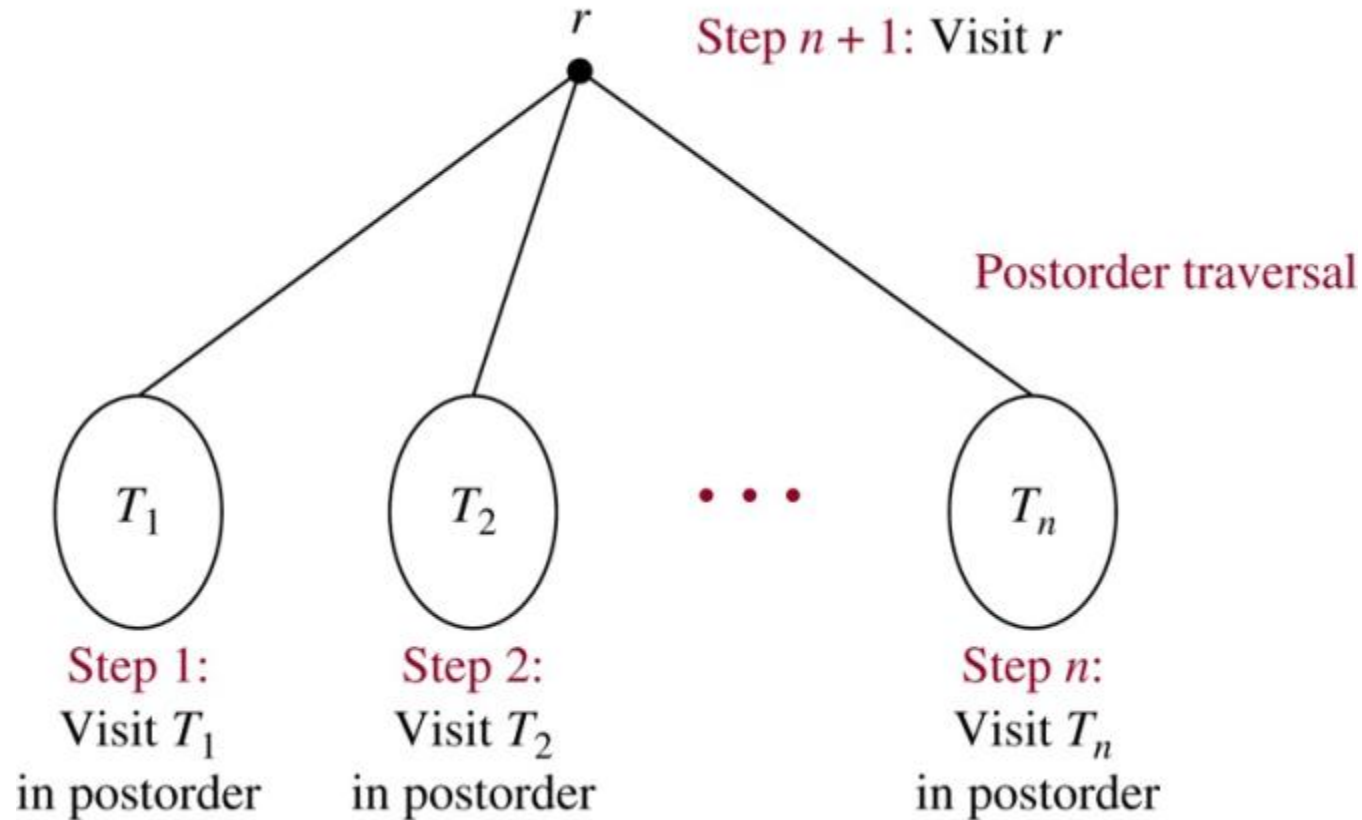
**for** each child  $c$  of  $r$  except for  $l$  from left to right

$T(c) := \text{subtree with } c \text{ as its root}$

*inorder*( $T(c)$ )

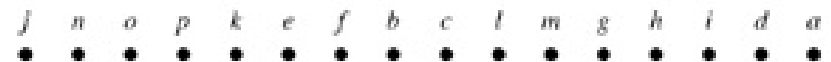
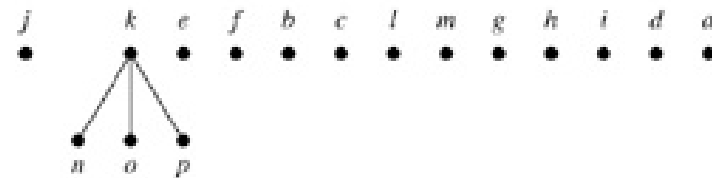
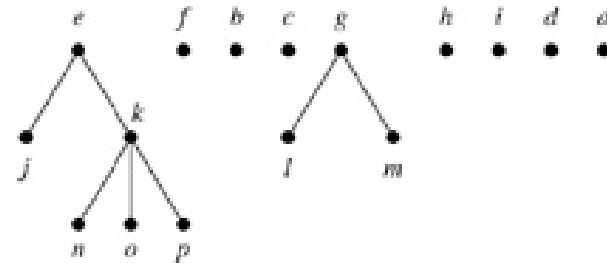
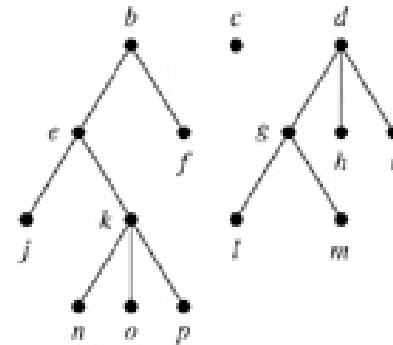
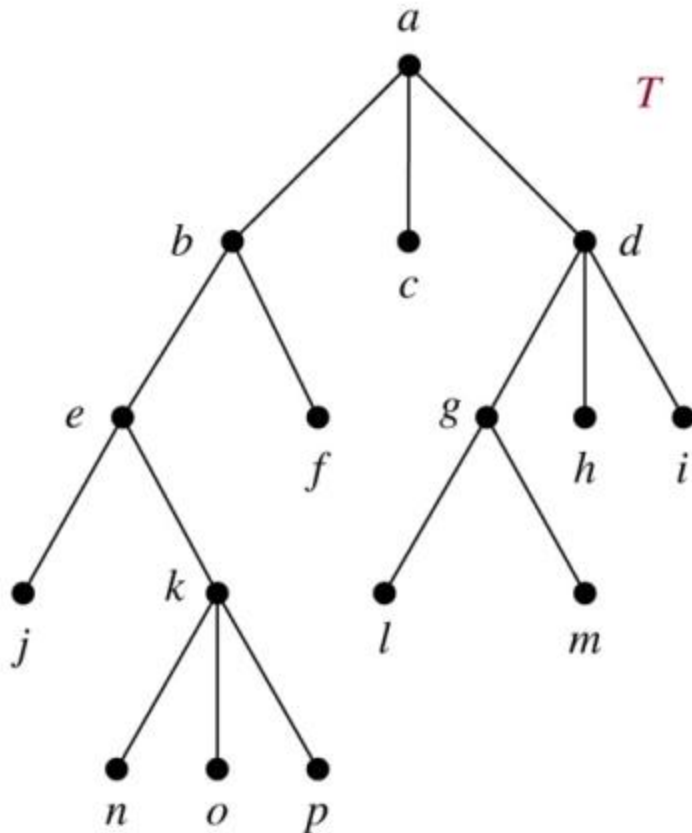
**end**

# Postorder traversal



**Example 4.** In which order does a preorder traversal visit the vertices in the ordered rooted tree  $T$  shown below?

**Sol:**



## Algorithm 3 (Postorder Traversal)

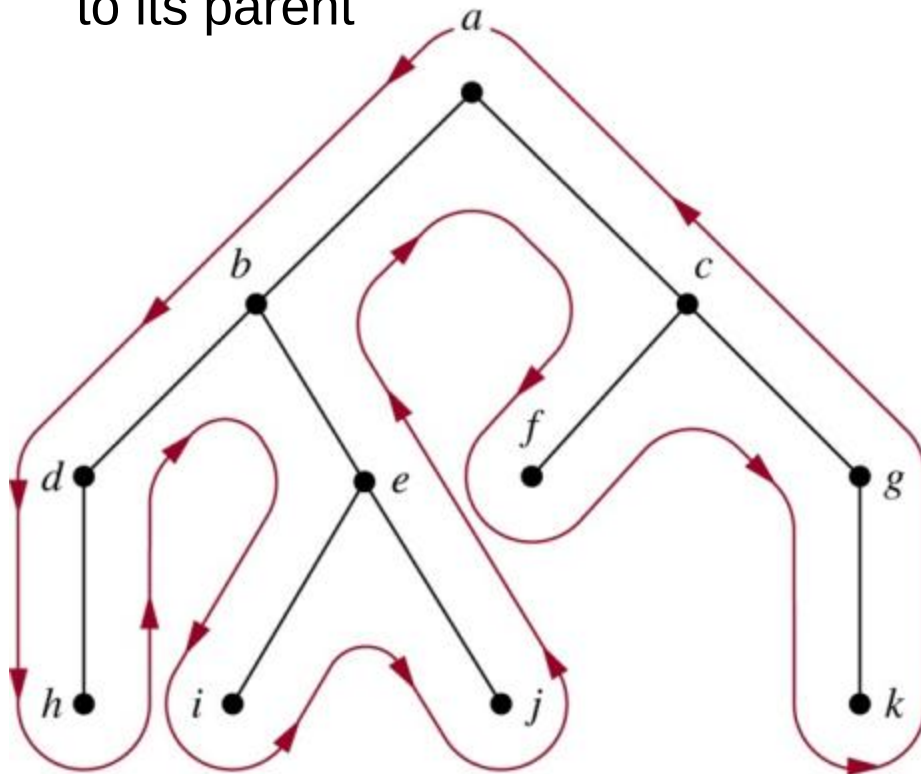
```
Procedure postorder( $T$ : ordered rooted tree)
 $r :=$  root of  $T$ 
for each child  $c$  of  $r$  from left to right
begin
     $T(c) :=$  subtree with  $c$  as its root
    postorder( $T(c)$ )
end
list  $r$ 
```

Easy representation: draw a red line around the ordered rooted tree starting at the root, moving along the edges

Preorder: listing each vertex the first time this line passes it

Inorder: listing a leaf the first time the line passes it and listing each internal vertex the second time the line passes it

Postorder: listing a vertex the last time it is passed on the way back up to its parent



Preorder:

*a, b, d, h, e, i, j, c, f, g, k*

Inorder:

*h, d, b, i, e, j, a, f, c, k, g*

Postorder:

*h, d, i, j, e, b, f, k, g, c, a*

# Infix, Prefix, and Postfix Notation

We can represent complicated expressions, such as compound propositions, combinations of sets, and arithmetic expressions using ordered rooted trees.

**Example 1** Find the ordered rooted tree for  $((x+y)^{\uparrow 2})+((x-4)/3)$ .

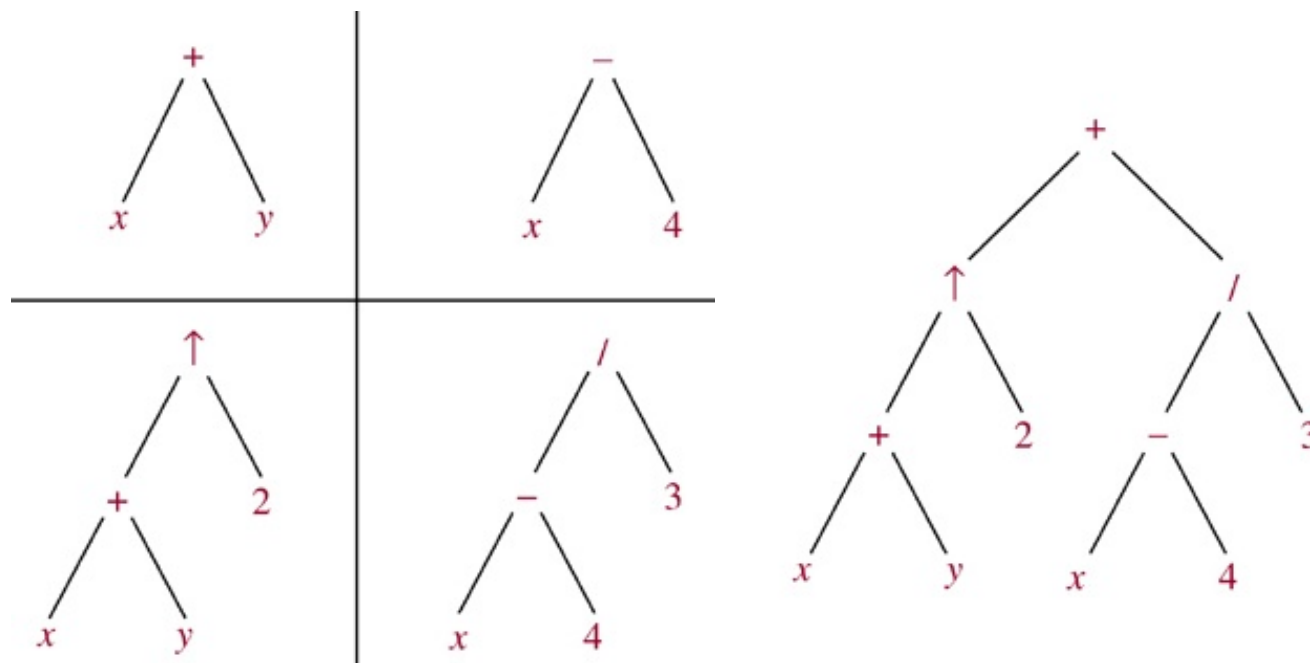
**Sol.**

leaf:

variable

internal vertex:

operation on  
its left and right  
subtrees

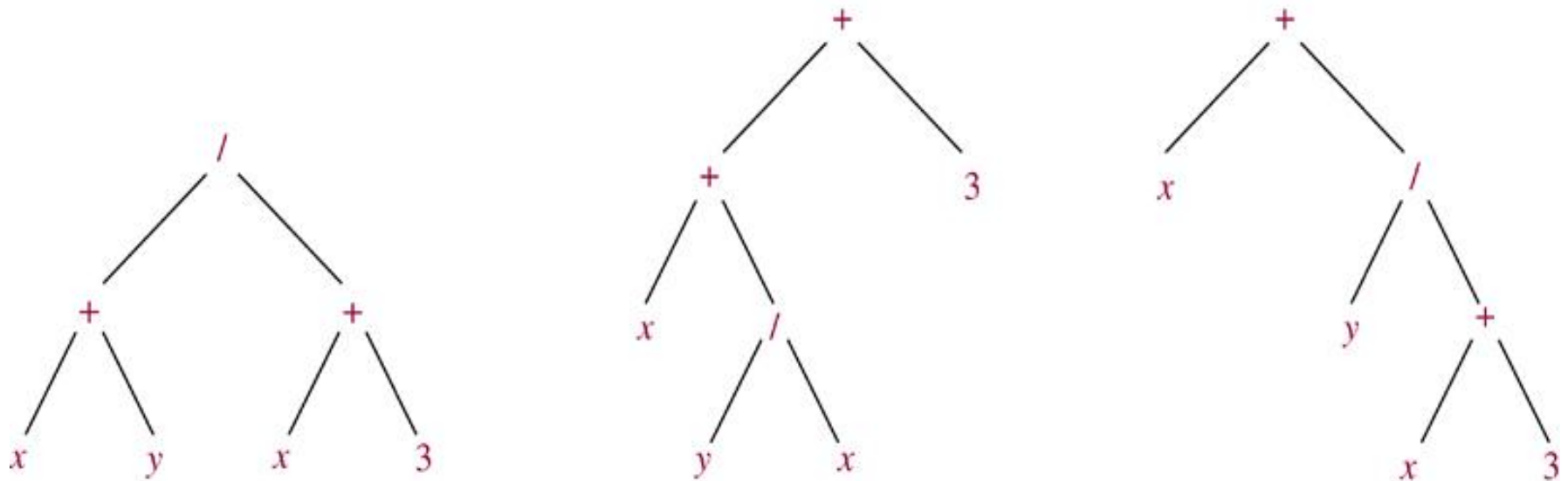




The following binary trees represent the expressions:

$(x+y)/(x+3)$ ,  $(x+(y/x))+3$ ,  $x+(y/(x+3))$ .

All their inorder traversals lead to  $x+y/x+3 \Rightarrow$  ambiguous  
 $\Rightarrow$  need parentheses



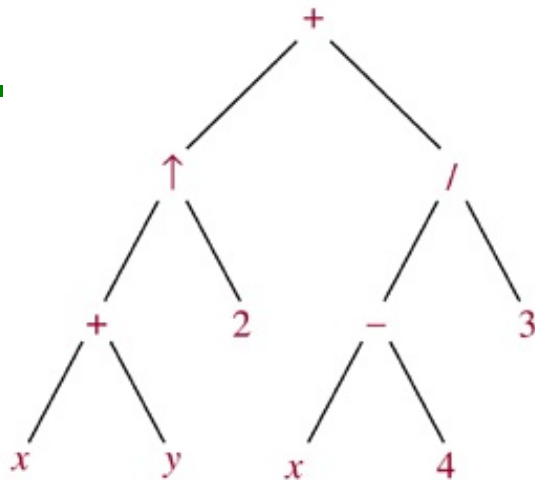
**Infix form:** An expression obtained when we traverse its rooted tree with inorder.

**Prefix form:** ... by preorder. (also named **Polish notation**)

**Postfix form:** ... by postorder (**reverse Polish notation**)

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

**Sol.**



$+ \uparrow + x y 2 / - x 4 3$

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

**Sol.**

$x y + 2 \uparrow x 4 - 3 / +$

**Note.** An expression in prefix form or postfix form is unambiguous, so no parentheses are needed.

**Example 7** What is the value of the prefix expression

$+ - * 2 3 5 / \uparrow 2 3 4$ ?

**Sol.**

Operation from right to left,  
First sign of operation (For  
example,  $\uparrow$ ) The two figures  
on the right doing this  
operation, Results in original  
place, And so on.

$$\begin{array}{ccccccccccc} + & - & * & 2 & 3 & 5 & / & \uparrow & 2 & 3 & 4 \\ & & & & & & & \underbrace{\phantom{2 \uparrow 3}} & & & \\ & & & & & & & 2 \uparrow 3 = 8 & & & \\ + & - & * & 2 & 3 & 5 & / & 8 & 4 & & \\ & & & & & & & \underbrace{\phantom{8 / 4}} & & & \\ & & & & & & & 8 / 4 = 2 & & & \\ + & - & * & 2 & 3 & 5 & 2 & & & & \\ & & & \underbrace{\phantom{2 * 3}} & & & & & & & \\ & & & 2 * 3 = 6 & & & & & & & \\ + & - & 6 & 5 & 2 & & & & & & \\ & & \underbrace{\phantom{6 - 5}} & & & & & & & & \\ & & 6 - 5 = 1 & & & & & & & & \\ + & 1 & 2 & & & & & & & & \\ & \underbrace{\phantom{1 + 2}} & & & & & & & & & \\ & 1 + 2 = 3 & & & & & & & & & \\ \text{Value of expression: } & 3 & & & & & & & & & \end{array}$$

**Example 9** What is the value of the postfix expression  
 $7\ 2\ 3\ *\ -\ 4\ \uparrow\ 9\ 3\ /\ +?$

**Sol.**

Operations from left to right,  
First sign of operation (For  
example,  $*$ ) Two numbers on  
the left of this operation, Result  
replaces the original location,  
and so on.

$$7\ 2\ 3\ *\ -\ 4\ \uparrow\ 9\ 3\ /\ +$$

$$2 * 3 = 6$$

$$7\ 6\ -\ 4\ \uparrow\ 9\ 3\ /\ +$$

$$7 - 6 = 1$$

$$1\ 4\ \uparrow\ 9\ 3\ /\ +$$

$$1^4 = 1$$

$$1\ 9\ 3\ /\ +$$

$$9 / 3 = 3$$

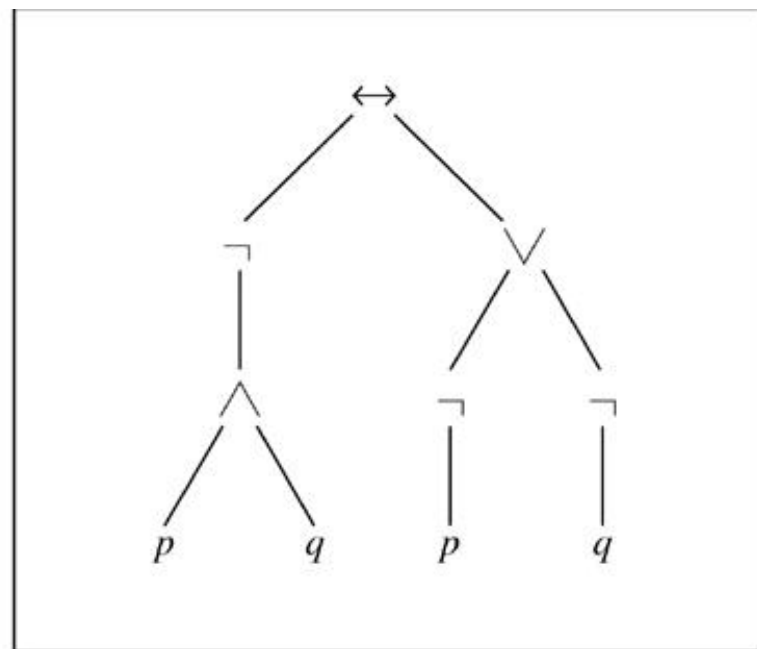
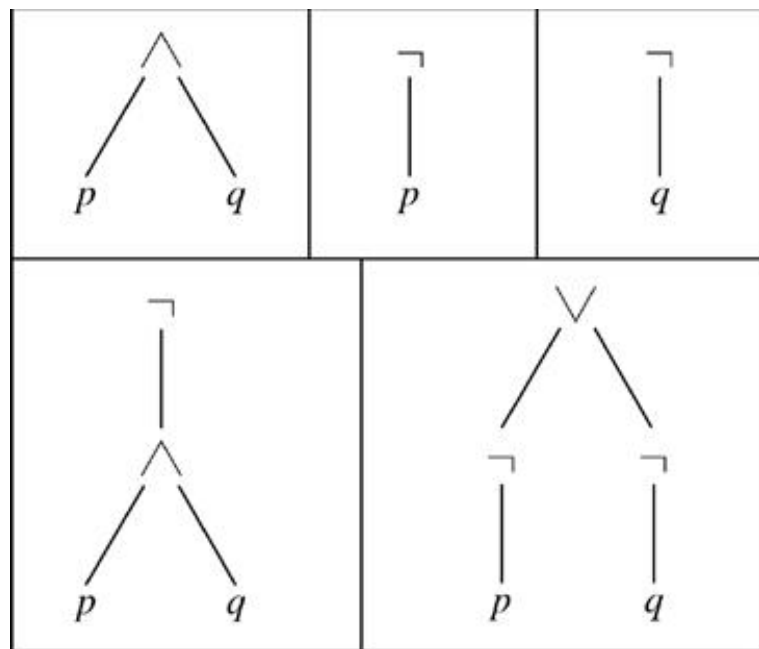
$$1\ 3\ +$$

$$1 + 3 = 4$$

Value of expression: 4

**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.

**Sol.**



prefix:  $\leftrightarrow \neg \wedge p q \vee \neg p \neg q$

postfix:  $p q \wedge \neg p \neg q \neg \vee \leftrightarrow$

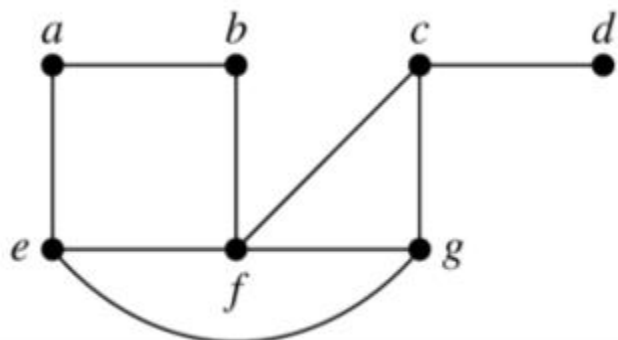
infix:  $(\neg(p \wedge q)) \leftrightarrow ((\neg p) \vee (\neg q))$

# 10.4 Spanning Trees

## Introduction

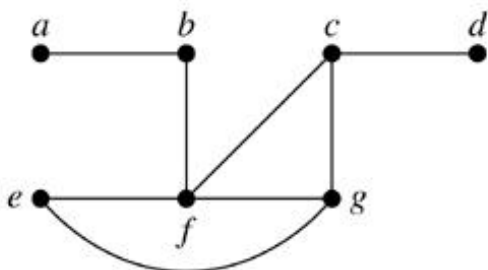
**Def.** Let  $G$  be a simple graph. A **spanning tree** of  $G$  is a subgraph of  $G$  that is a tree containing every vertex of  $G$ .

**Example 1** Find a spanning tree of  $G$ .



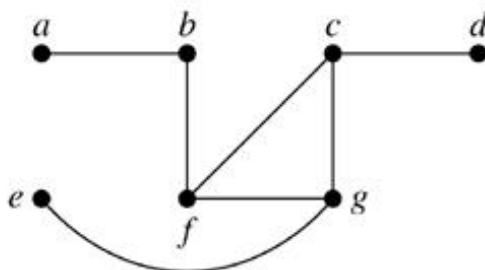
**Sol.**

Remove an edge from any circuit.  
(repeat until no circuit exists)



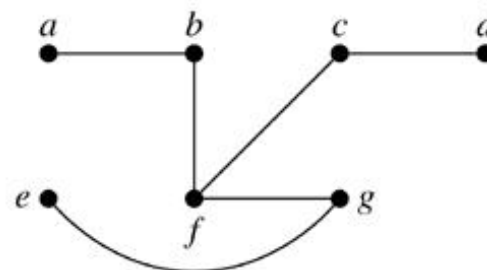
Edge removed:  $\{a, e\}$

(a)



$\{e, f\}$

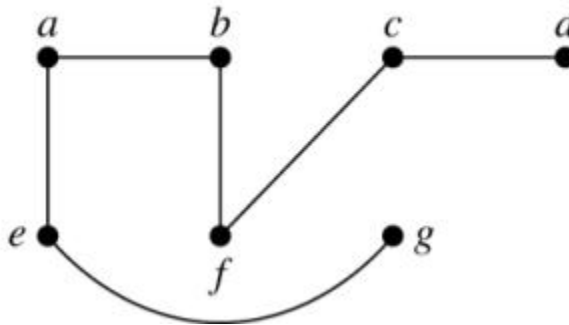
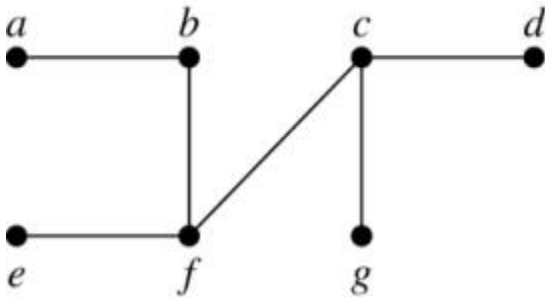
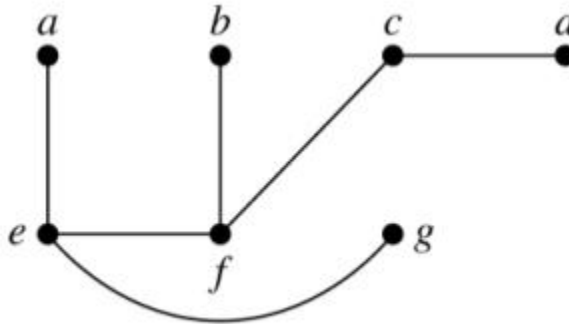
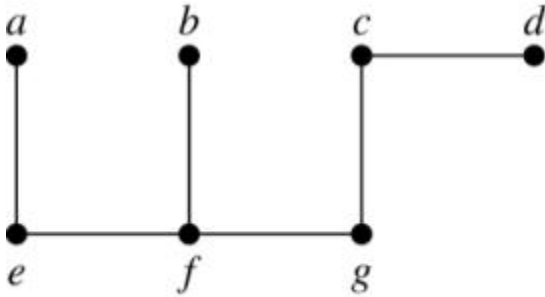
(b)



$\{c, g\}$

(c)

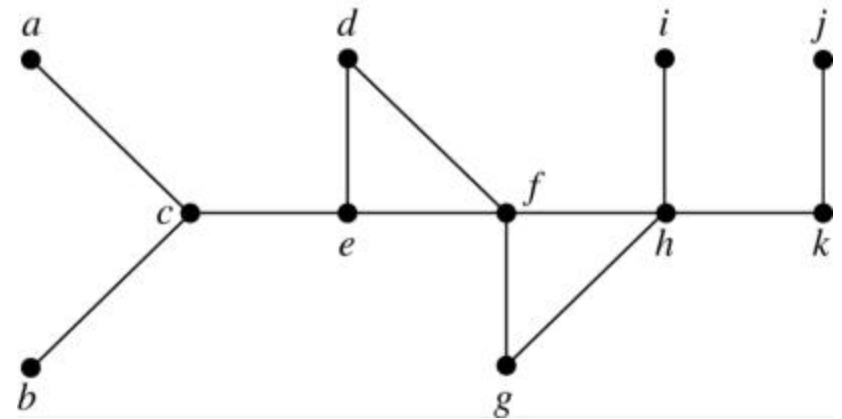
## Four spanning trees of $G$ :



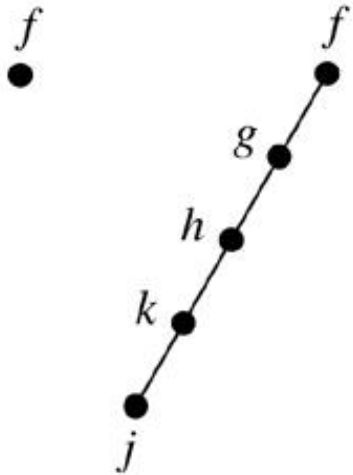
**Thm 1** A simple graph is connected if and only if it has a spanning tree.

# Depth-First Search (DFS)

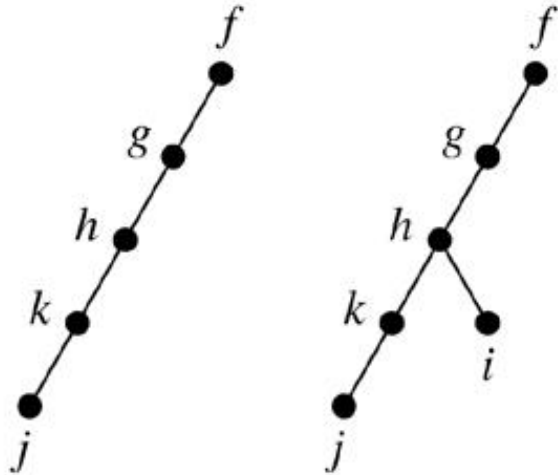
**Example 3** Use depth-first search to find a spanning tree for the graph.



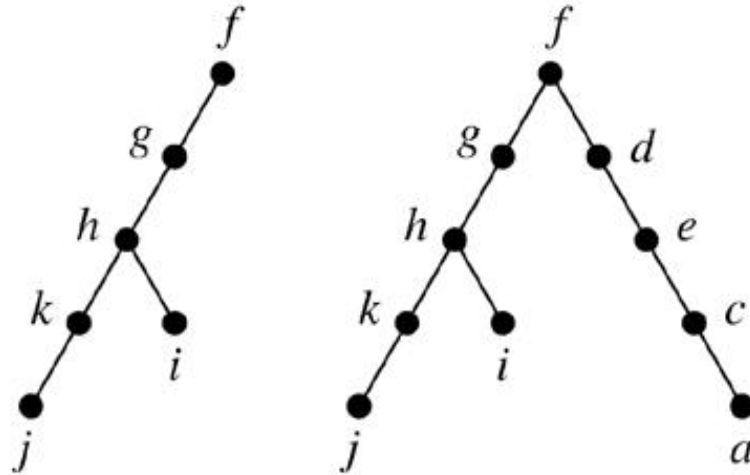
**Sol.** (arbitrarily start with the vertex  $f$ )



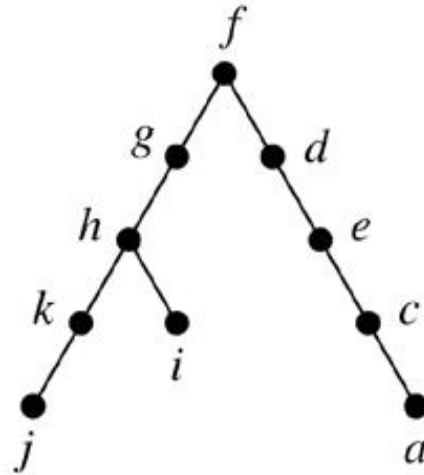
(a)



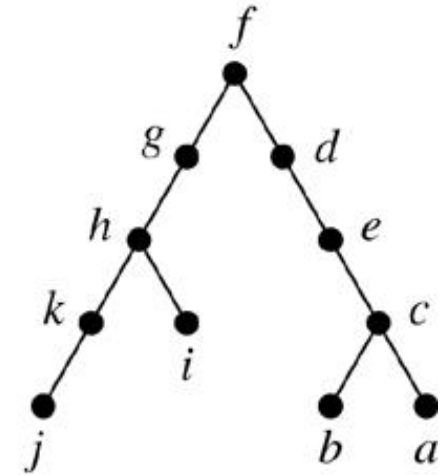
(b)



(c)



(d)

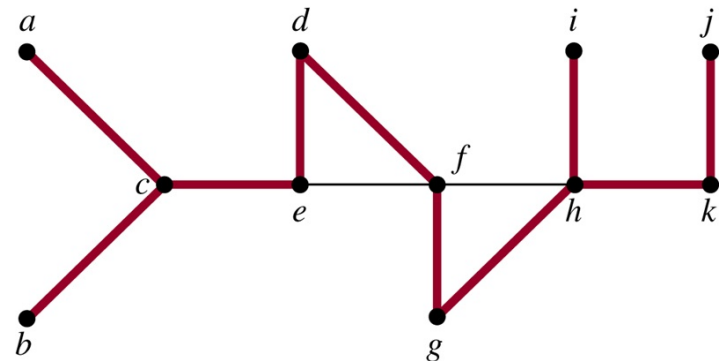
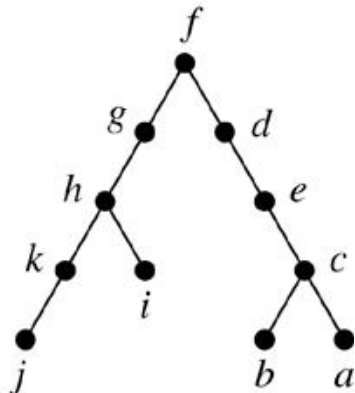
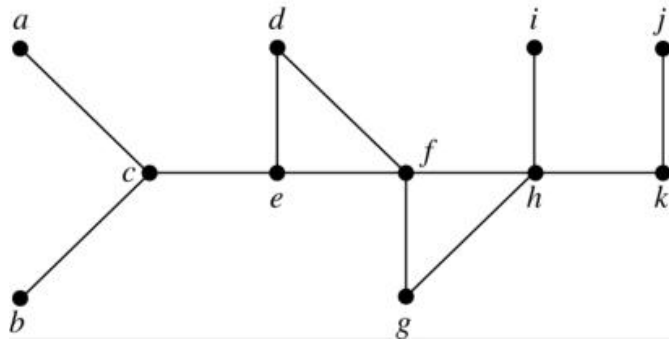


(e)



The edges selected by DFS 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**.

### Example 4 (Previous question)



The tree edges (red)  
and back edges (black)

# Algorithm 1 (Depth-First Search)

**Procedure** *DFS*( $G$ : connected graph with vertices  $v_1, v_2, \dots, v_n$ )

$T :=$  tree consisting only of the vertex  $v_1$

*visit*( $v_1$ )

**procedure** *visit*( $v$ : vertex of  $G$ )

**for** each vertex  $w$  adjacent to  $v$  and not yet in  $T$

**begin**

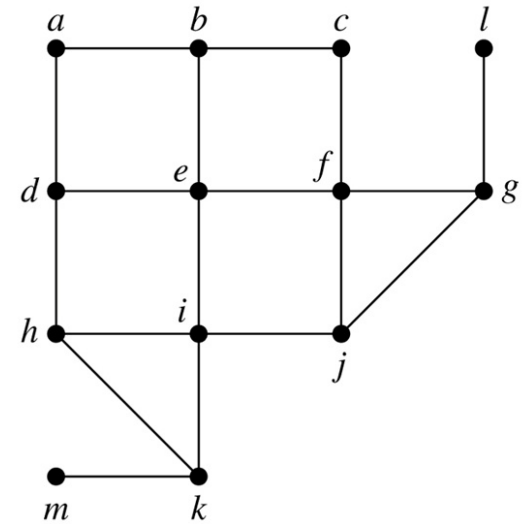
    add vertex  $w$  and edge  $\{v, w\}$  to  $T$

*visit*( $w$ )

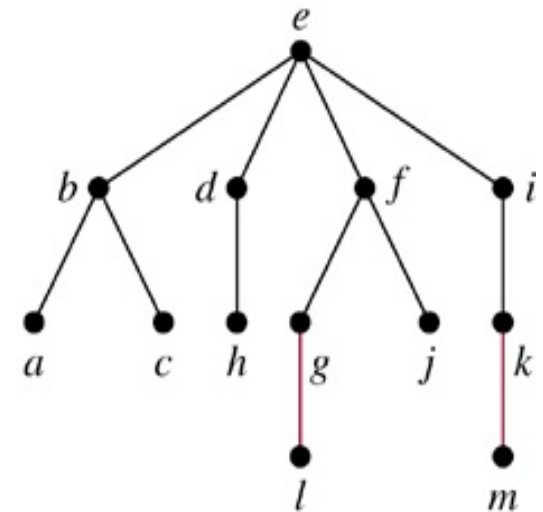
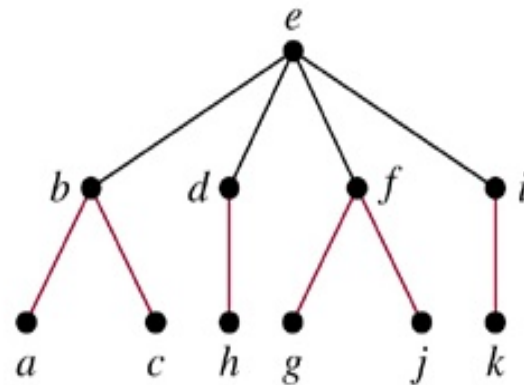
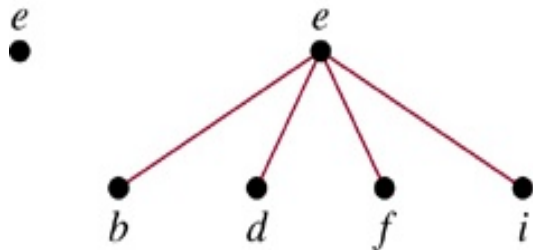
**end**

# Breadth-First Search (BFS)

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



**Sol.** (arbitrarily start with the vertex *e*)



## Algorithm 2 (Breadth-First Search)

```
Procedure BFS(G: connected graph with vertices  $v_1, v_2, \dots, v_n$ )  
   $T :=$  tree consisting only of vertex  $v_1$   
   $L :=$  empty list  
  put  $v_1$  in the list  $L$  of unprocessed vertices  
  while  $L$  is not empty  
  begin  
    remove the first vertex  $v$  from  $L$   
    for each neighbor  $w$  of  $v$   
      if  $w$  is not in  $L$  and not in  $T$  then  
      begin  
        add  $w$  to the end of the list  $L$   
        add  $w$  and edge  $\{v, w\}$  to  $T$   
      end  
    end  
  end
```

## Backtracking Applications

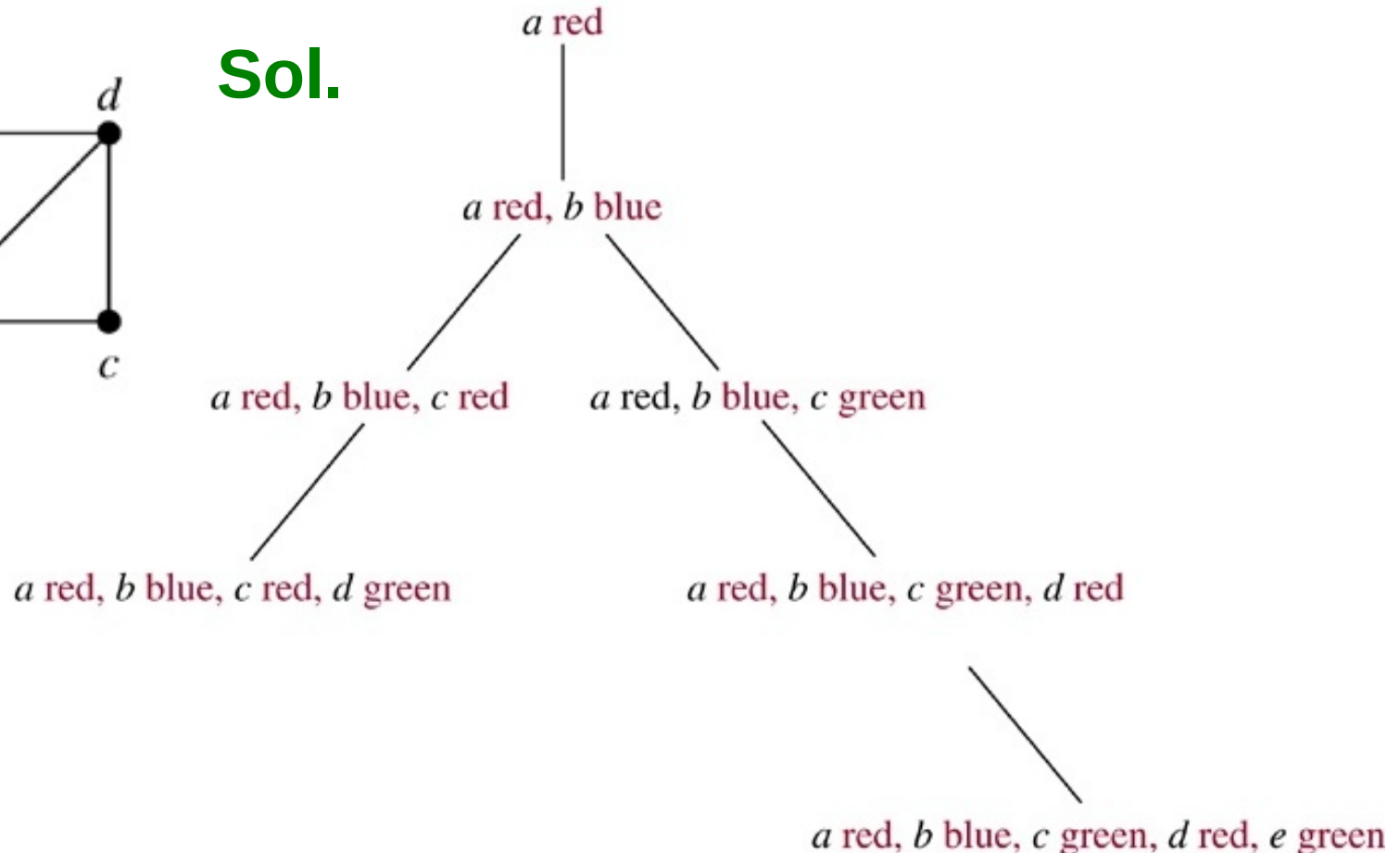
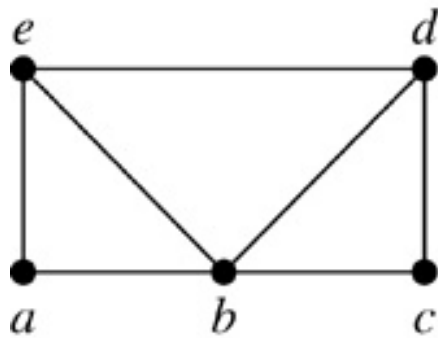
There are problems that can be solved only by performing an exhaustive (Complete) search of all possible solutions.

**Decision tree**: each internal vertex represents a decision, and each leaf is a possible solution.

To find a solution via **backtracking**: On the decision tree from the root to do a series of decision to the leaf, If the leaf is not the solution, or the entire subtree check solution is not found, Back to the top parent instead of looking for another tree.

**Example 6** (Graph Colorings) How can backtracking be used to decide whether the following graph can be colored using 3 colors?

**Sol.**

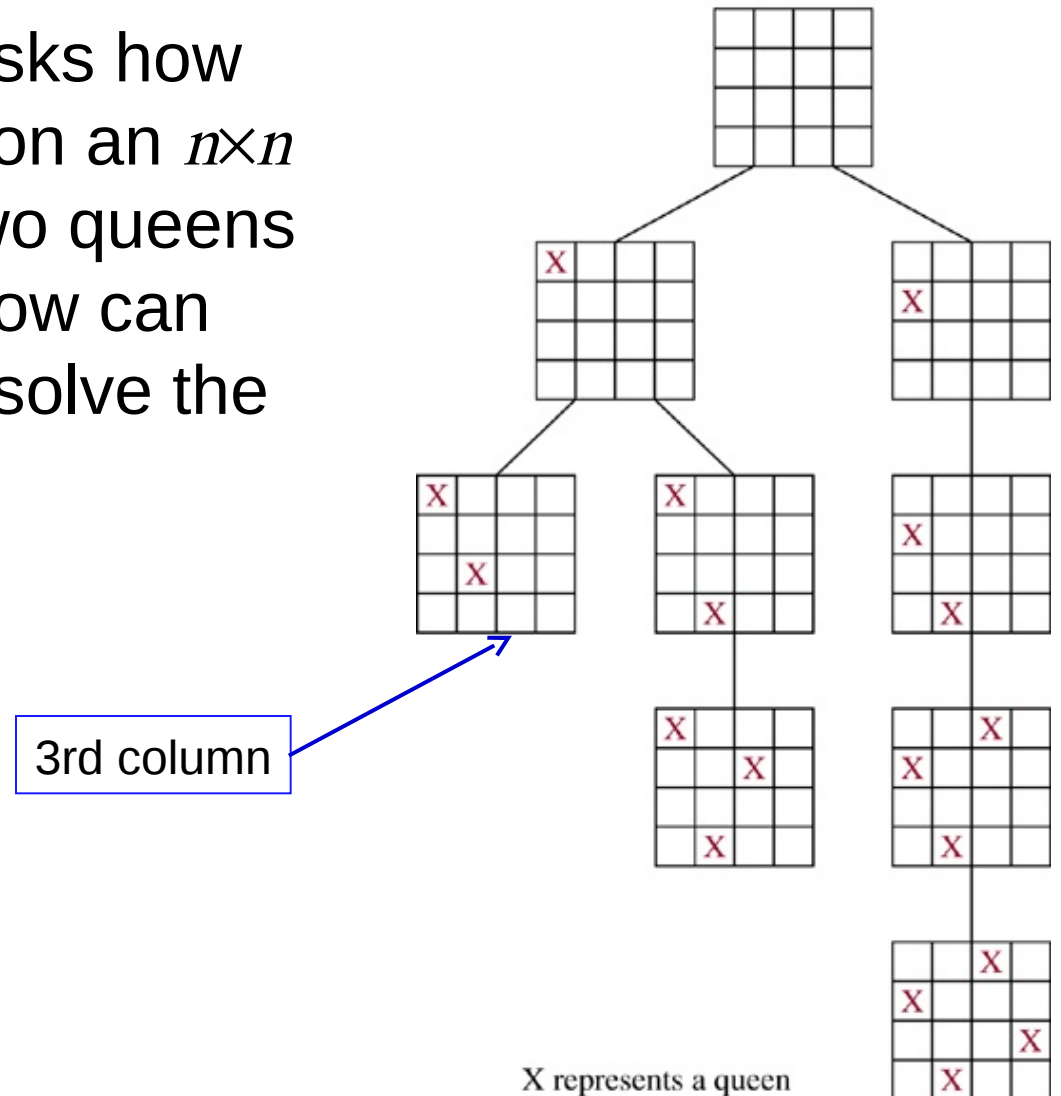


## Example 7

(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 on another. How can backtracking be used to solve the  $n$ -queens problem.

**Sol.** A case study of  $n=4$



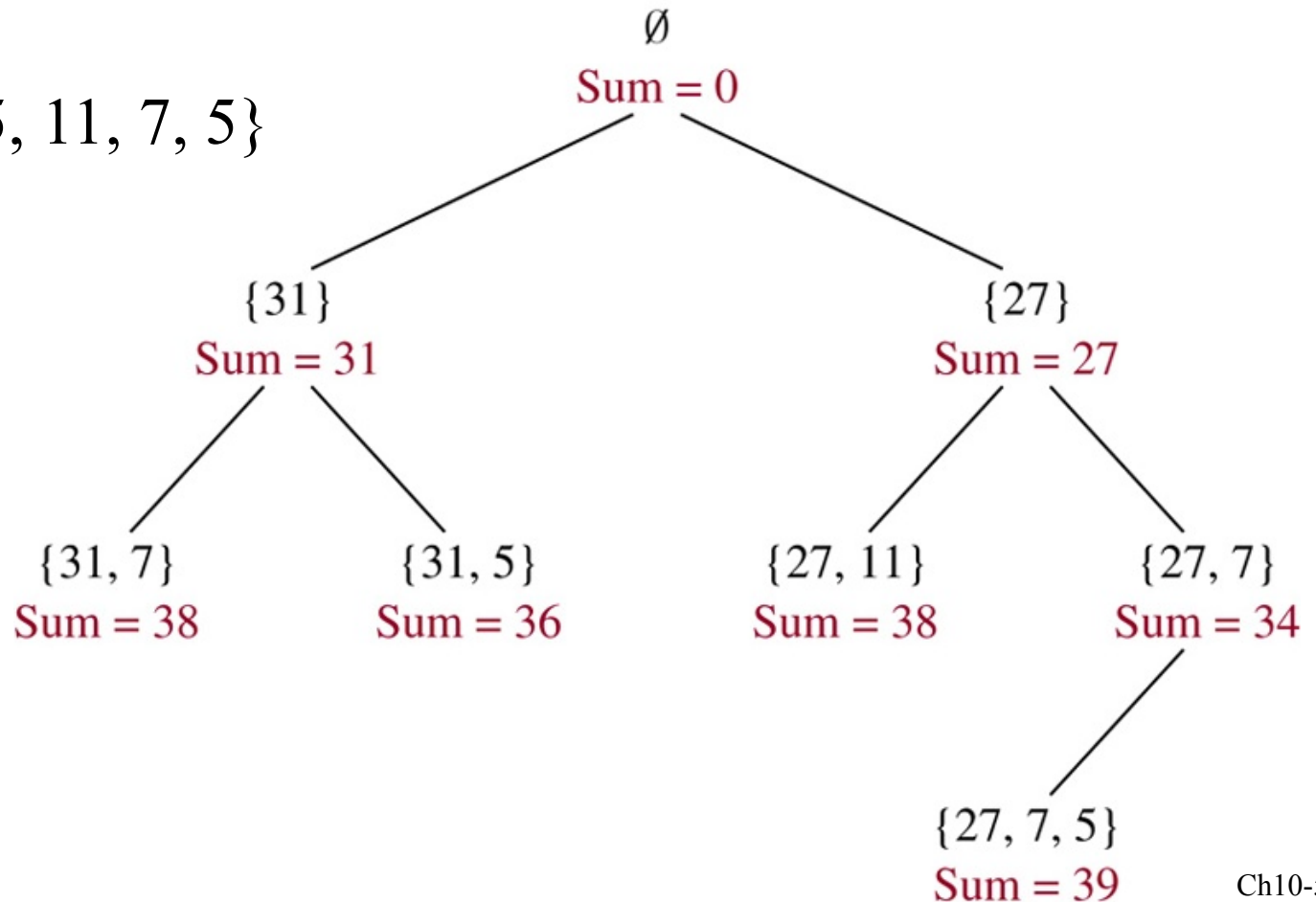
## Example 8 (Sum of Subsets)

Give a set  $S$  of positive integers  $x_1, x_2, \dots, x_n$ , find a subset of  $S$  that has  $M$  as its sum. How can backtracking be used to solve this problem.

**Sol.**

$$S = \{31, 27, 15, 11, 7, 5\}$$

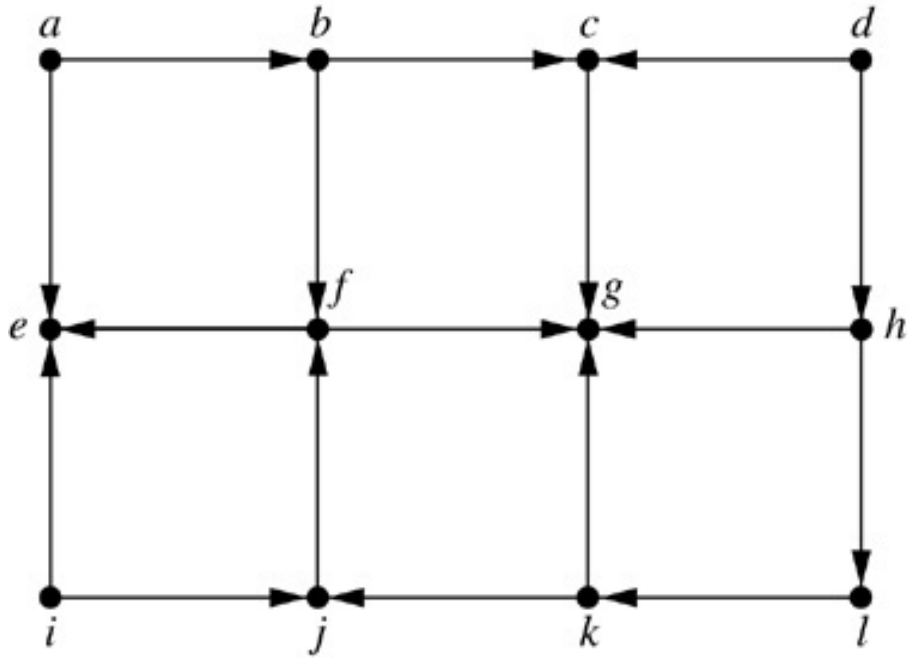
$$M = 39$$





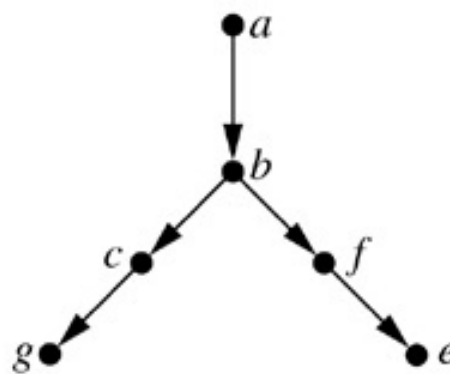
# Depth-First Search in Directed Graphs

**Example 9** What is the output of DFS given the graph G?

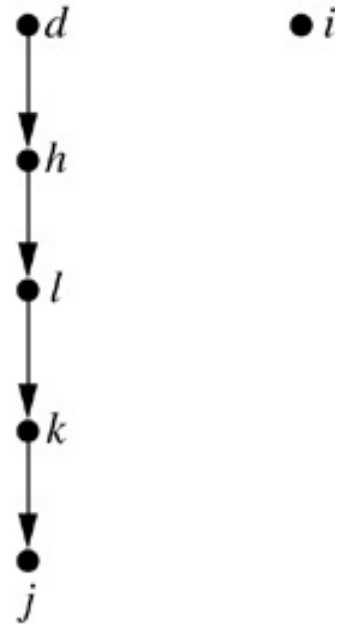


(a)

## Sol.



(b)



# 10.5 Minimum Spanning Trees

$G$ : connected weighted graph (each edge has an weight  $\geq 0$ )

**Def.** **minimum spanning tree** of  $G$ : a spanning tree of  $G$  with smallest sum of weights of its edges.

## Algorithms for Minimum Spanning Trees

### Algorithm 1 (Prim's Algorithm)

**Procedure** *Prim*( $G$ : connected weighted 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  $T$

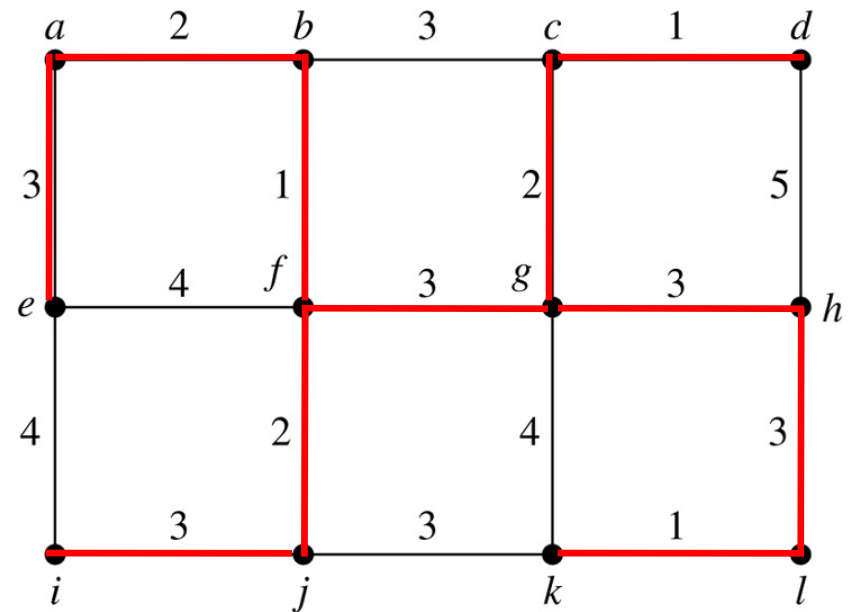
$T := T$  with  $e$  added

**end** {  $T$  is a minimum spanning tree of  $G$  }

**Example 2** Use Prim's algorithm to find a minimum spanning tree of  $G$ .

**Sol.**

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:		<u>24</u>



(Maintain only one process tree)

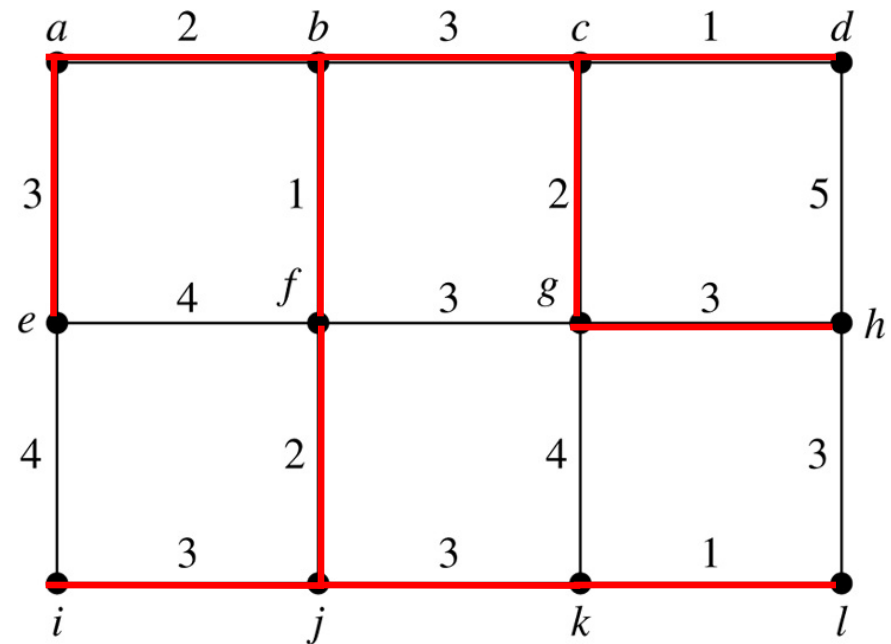
## Algorithm 2 (Kruskal Algorithm)

```
Procedure Kruskal( $G$ : connected weighted 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  $G$  }
```

**Example 3** Use Kruskal algorithm to find a minimum spanning tree of  $G$ .

**Sol.**

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



Process tree will typically have several