CS215 DISCRETE MATHEMATICS FOR COMPUTER SCIENCE

Dr. Qi Wang

Department of Computer Science and Engineering
Office: Room413, CoE South Tower
Email: wangqi@sustech.edu.cn

Euler's Formula

- **Theorem** (Euler's Formula) Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G. Then $r = e - v + 2$.

Euler's Formula

- **Theorem** (Euler's Formula) Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G. Then $r = e - v + 2$.

- **Corollary 1** If G is a connected planar simple graph with e edges and v vertices, where $v \geq 3$, then $e \leq 3v - 6$.

  Corollary 2 If G is a connected planar simple graph, then G has a vertex of degree not exceeding 5.

  Corollary 3 In a connected planar simple graph has e edges and v vertices with $v \geq 3$ and no circuits of length three, then $e \leq 2v - 4$.

Graph Coloring

- A *coloring* of a simple graph is the assignment of a color to each vertex of the graph so that no two adjacent vertices are assigned the same color.

Graph Coloring

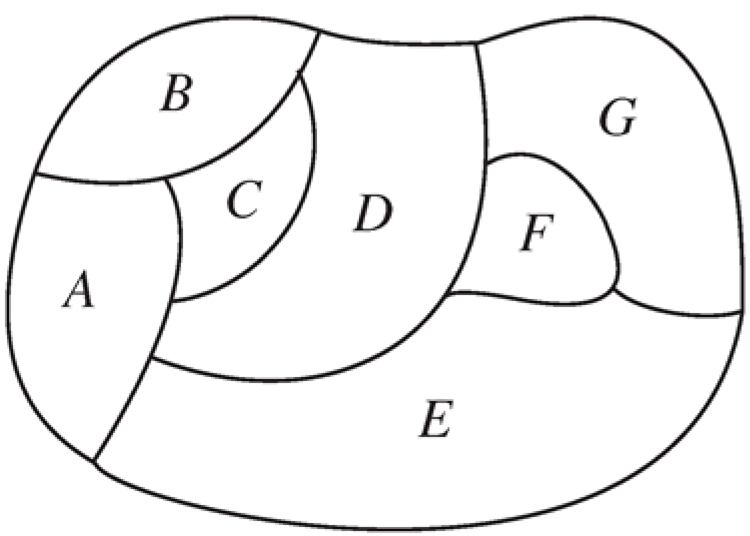
- A *coloring* of a simple graph is the assignment of a color to each vertex of the graph so that no two adjacent vertices are assigned the same color.

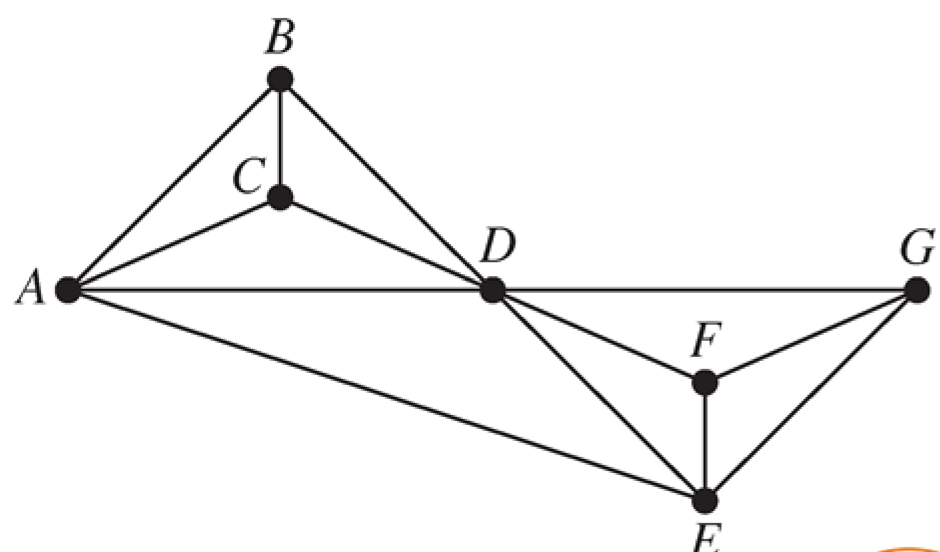
  The *chromatic number* of a graph is the least number of colors needed for a coloring of this graph, denoted by $\chi(G)$.

Graph Coloring

- A *coloring* of a simple graph is the assignment of a color to each vertex of the graph so that no two adjacent vertices are assigned the same color.

  The *chromatic number* of a graph is the least number of colors needed for a coloring of this graph, denoted by $\chi(G)$.



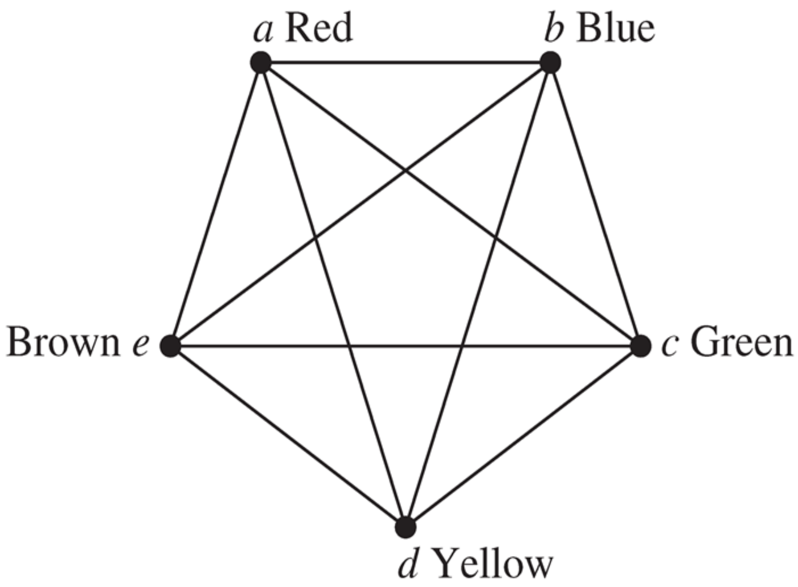


Examples

- What is the chromatic number of K_n, $K_{m,n}$, C_n?

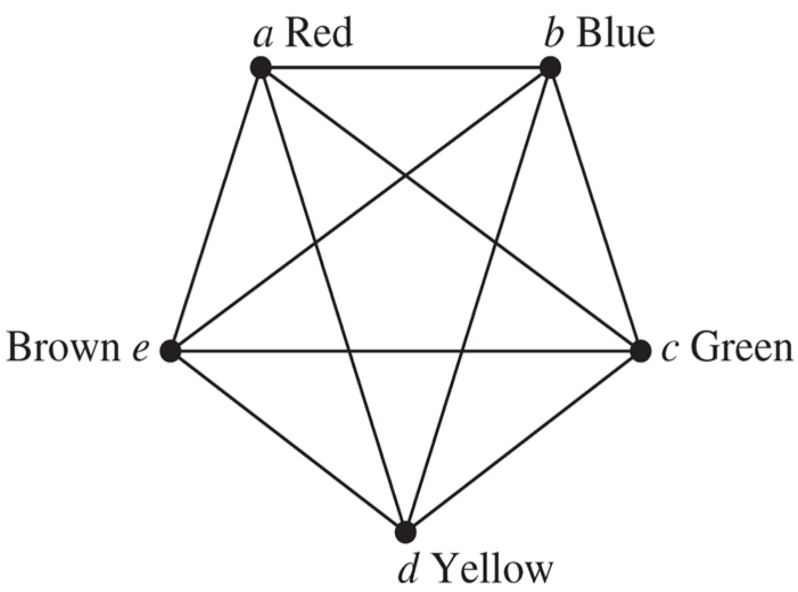
Examples

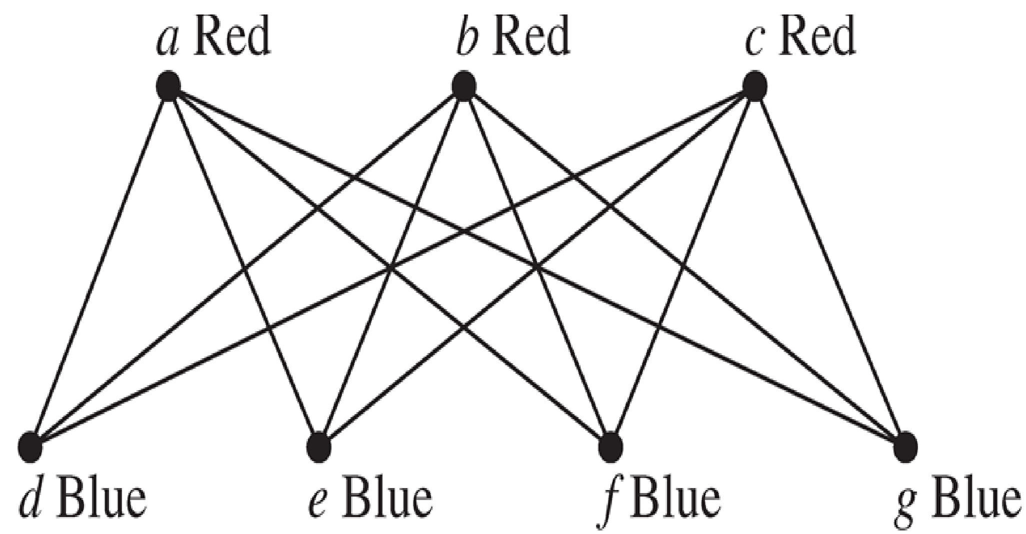
- What is the chromatic number of K_n, $K_{m,n}$, C_n?



Examples

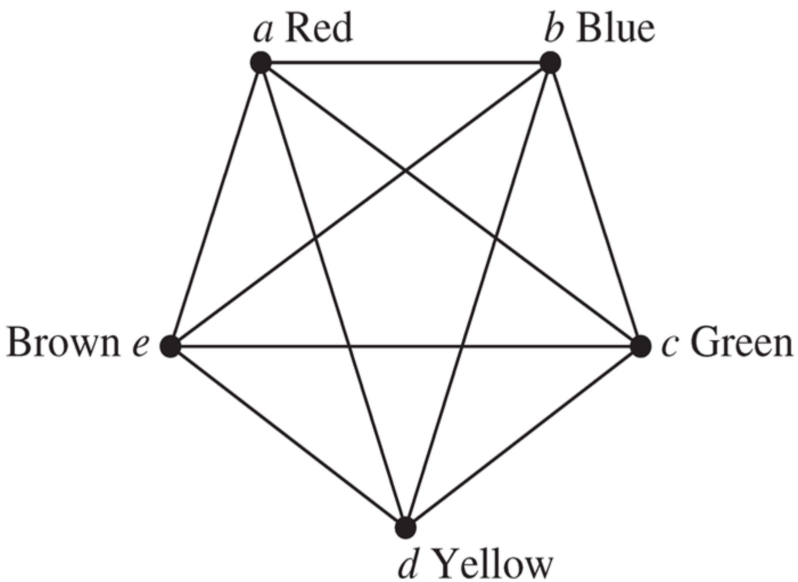
- What is the chromatic number of K_n, $K_{m,n}$, C_n?

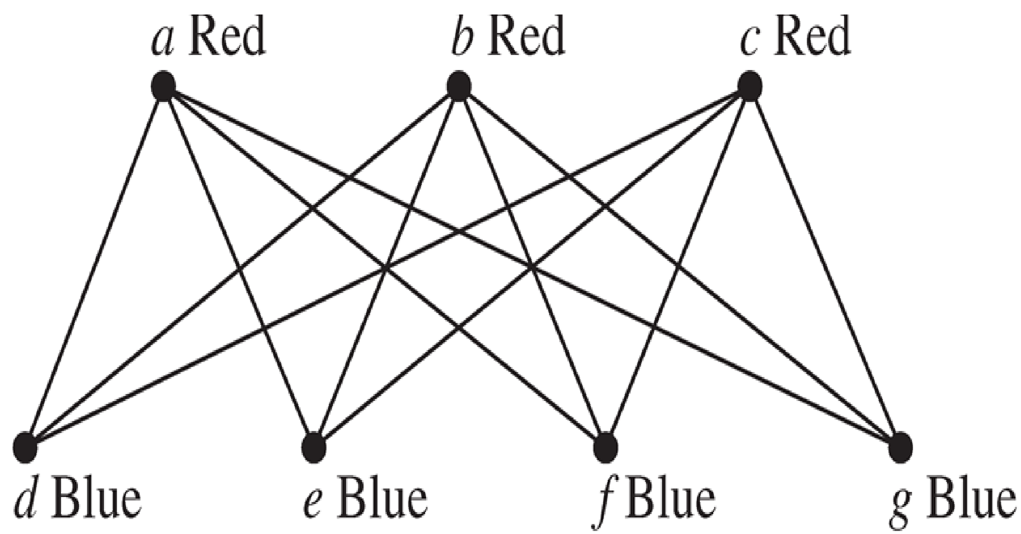


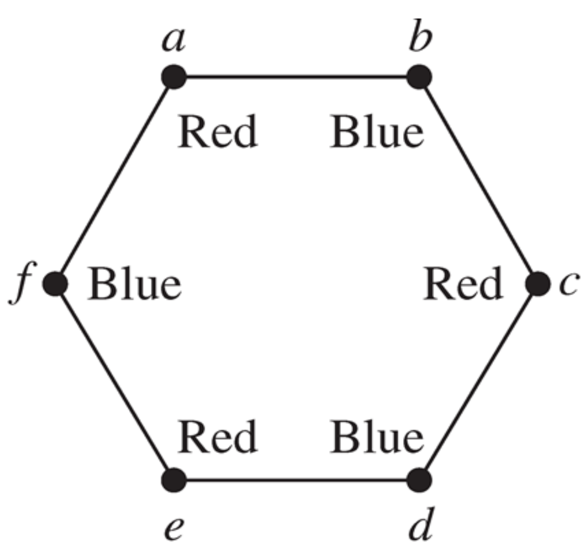


Examples

- What is the chromatic number of K_n, $K_{m,n}$, C_n?

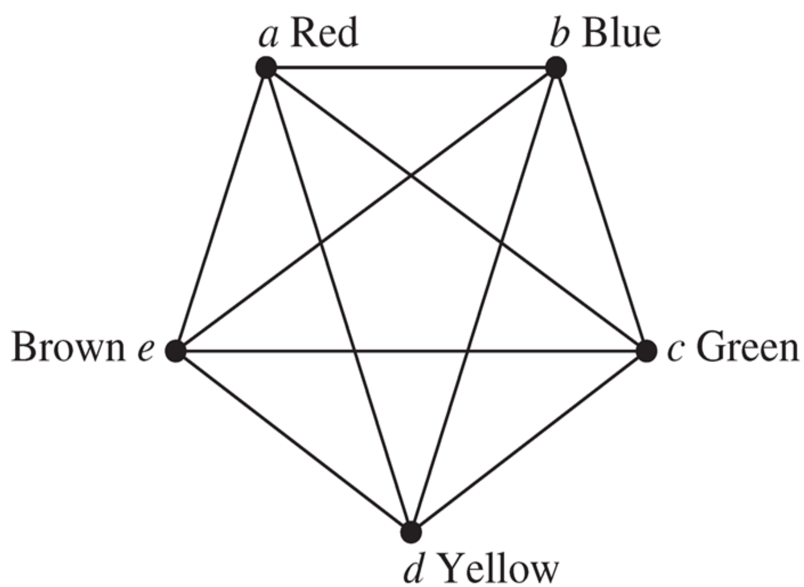


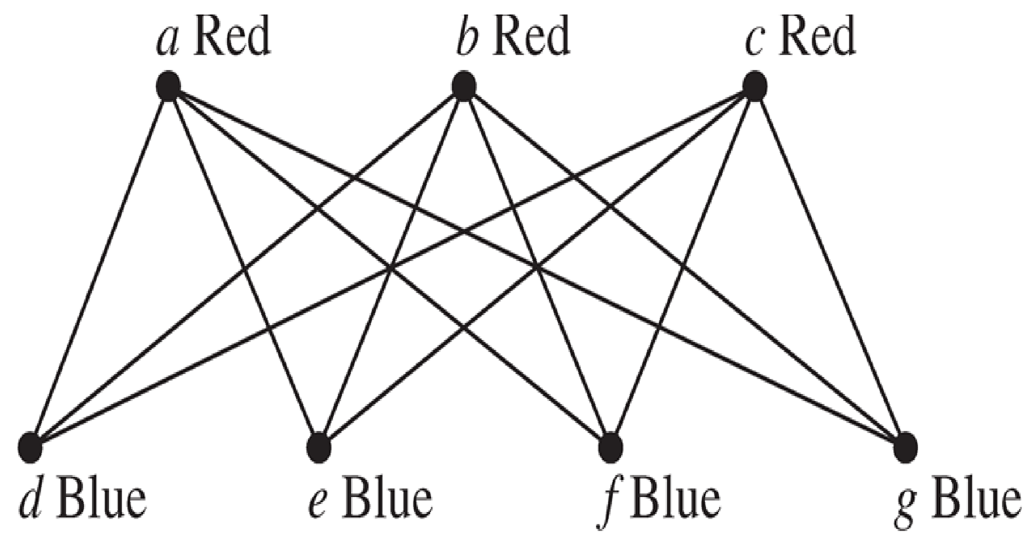


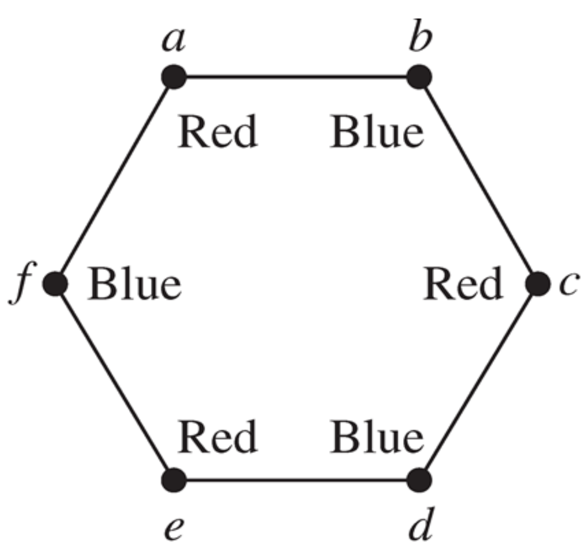


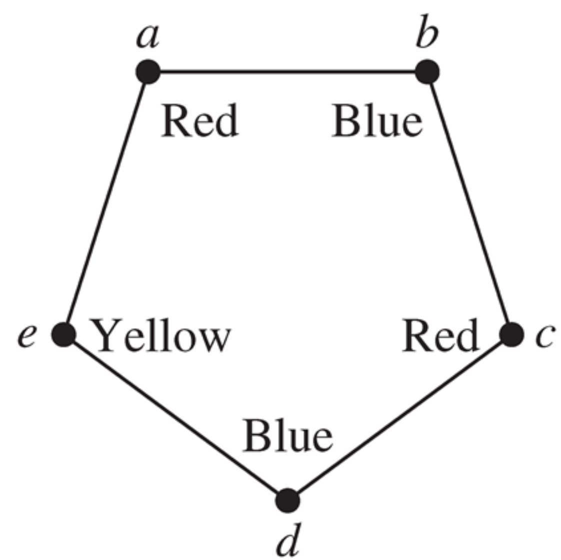
Examples

- What is the chromatic number of K_n, $K_{m,n}$, C_n?





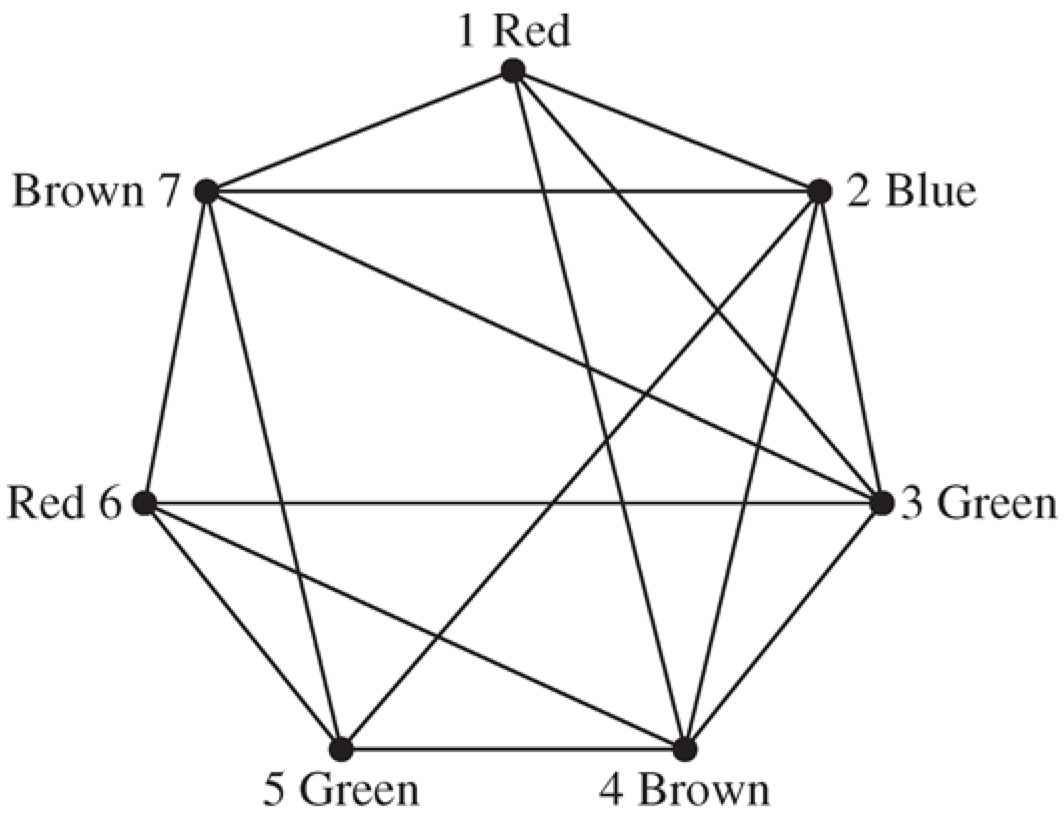




Applications of Graph Coloring

- Scheduling Final Exams

  Vertices represent courses, and there is an edge between two vertices if there is a common student in the courses.



Time Period	Courses
I	1, 6
II	2
III	3, 5
IV	4, 7

Applications of Graph Coloring

- ## Channel Assignments

  Television channels 2 through 13 are assigned to stations in North America so that no two stations within 150 miles can operate on the same channel . How can the assignment of channels be modeled by graph coloring?

Applications of Graph Coloring

- Channel Assignments

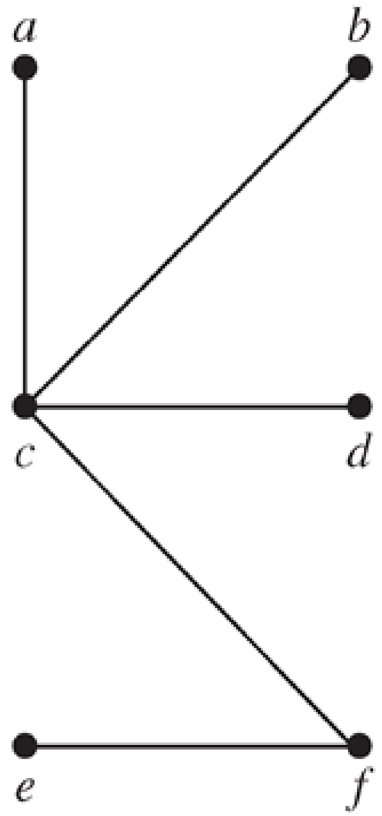
  Television channels 2 through 13 are assigned to stations in North America so that no two stations within 150 miles can operate on the same channel . How can the assignment of channels be modeled by graph coloring?

Graph Coloring $\in$ NPC

Trees

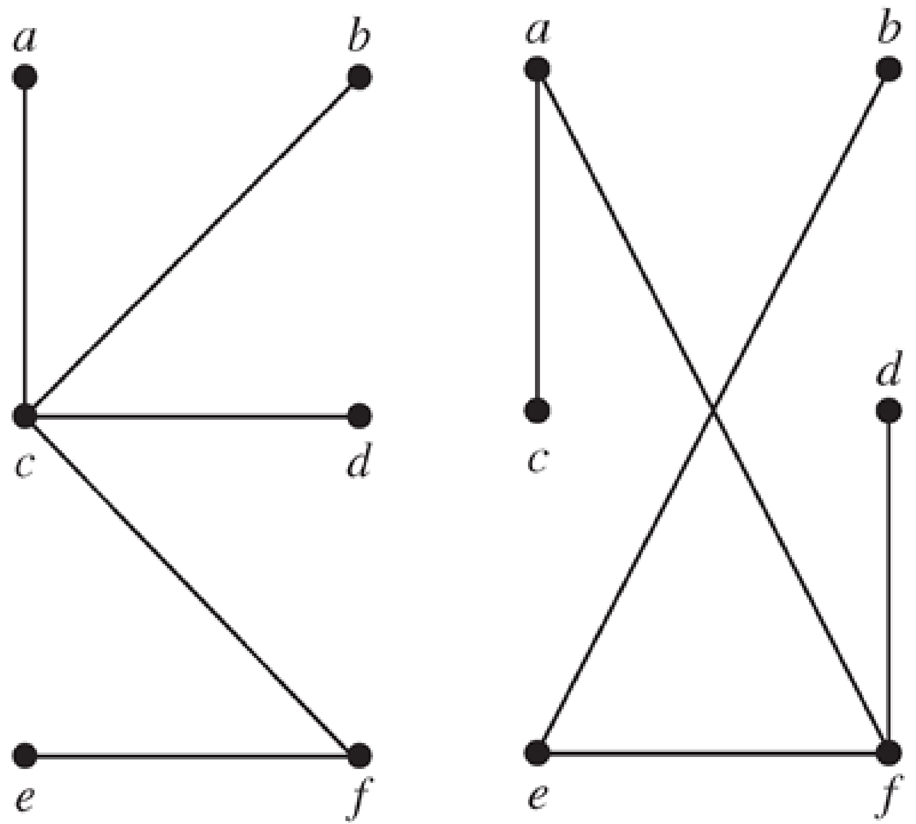
- **Definition** A *tree* is a connected undirected graph with no simple circuits.

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

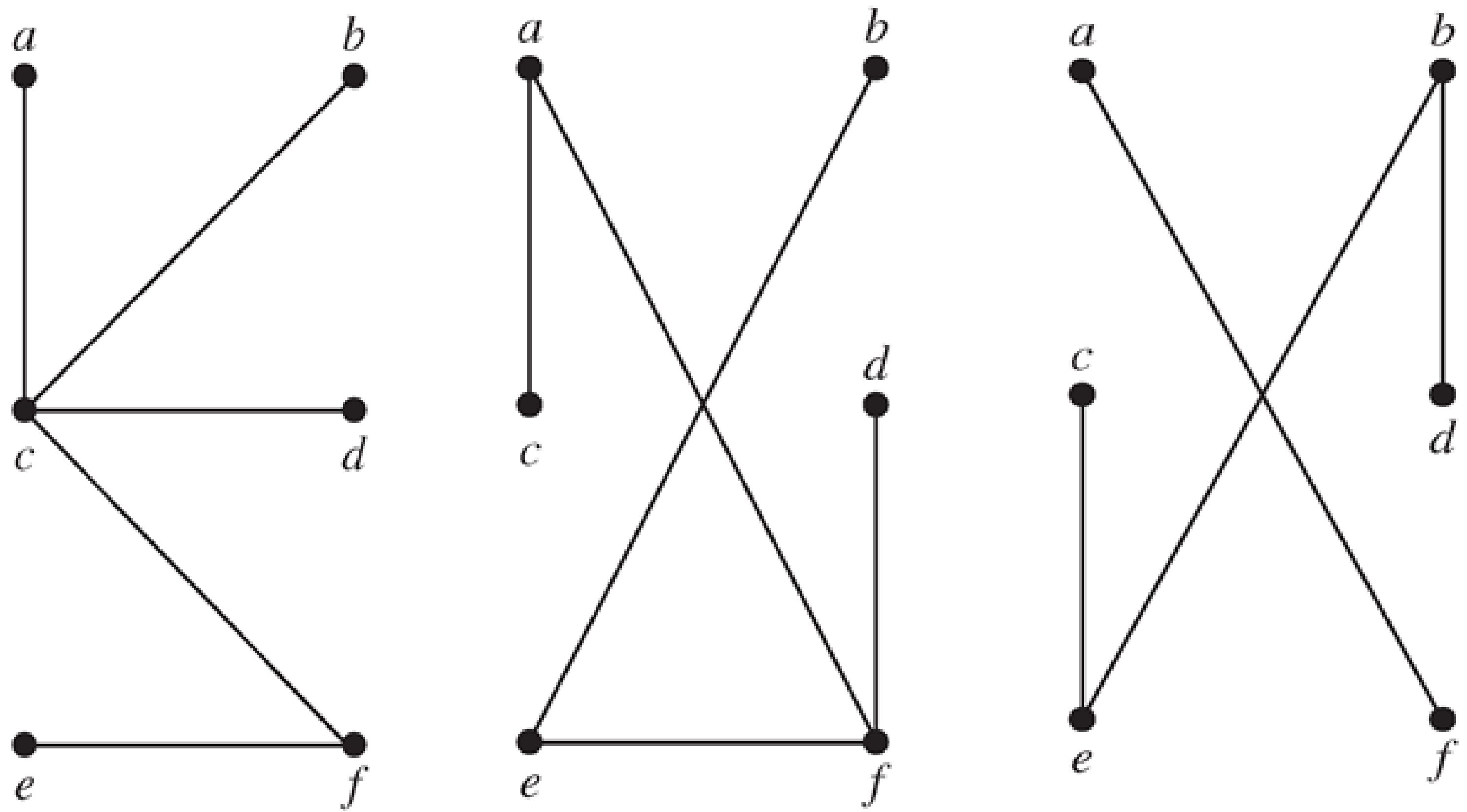


Trees

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

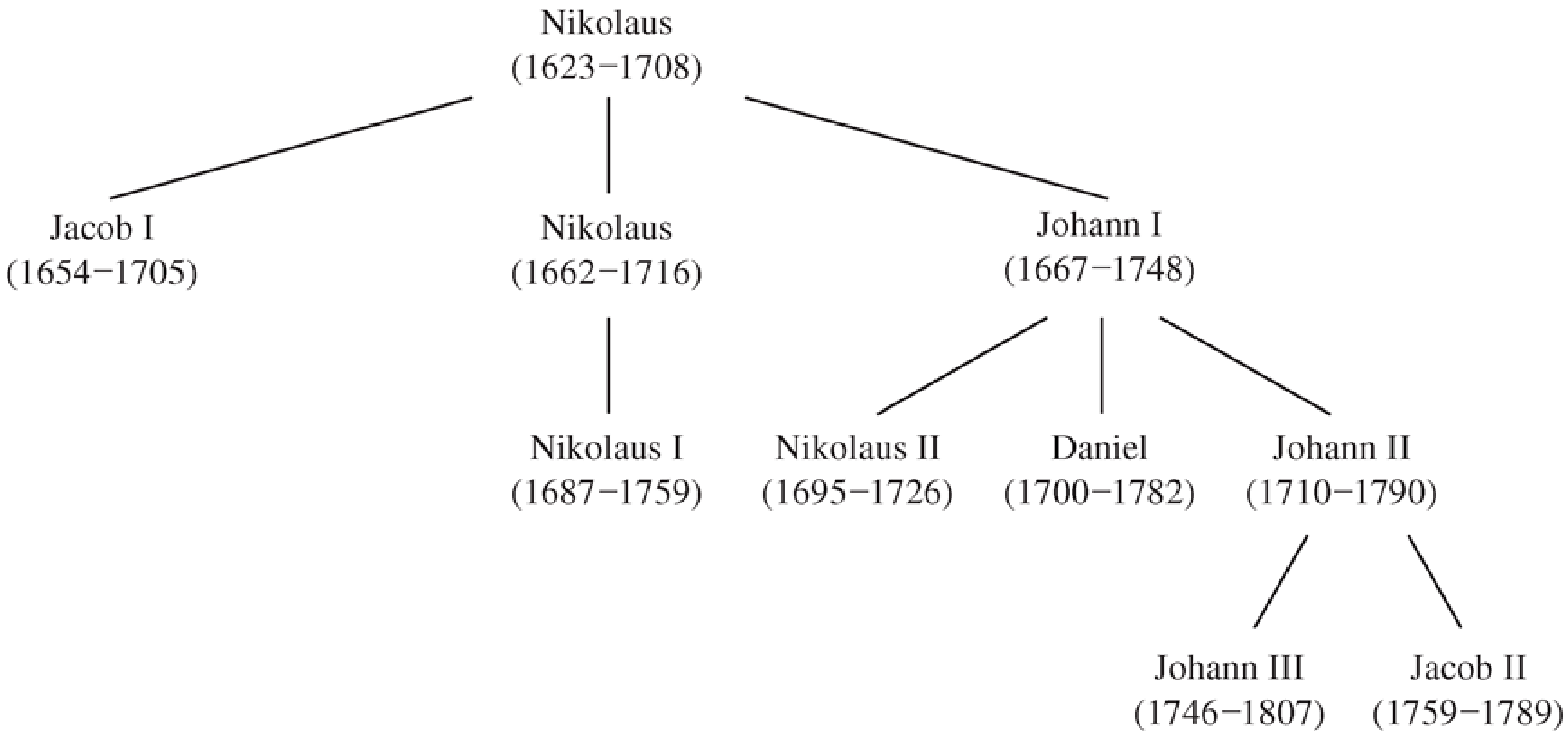


- **Definition** A *tree* is a connected undirected graph with no simple circuits.



Trees

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



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

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

Proof

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

Proof

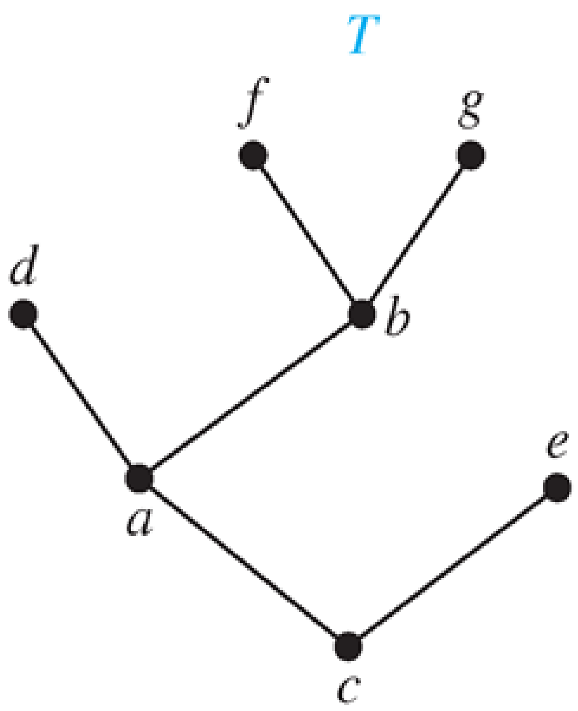
Two properties of tree: connected, no circuit

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.

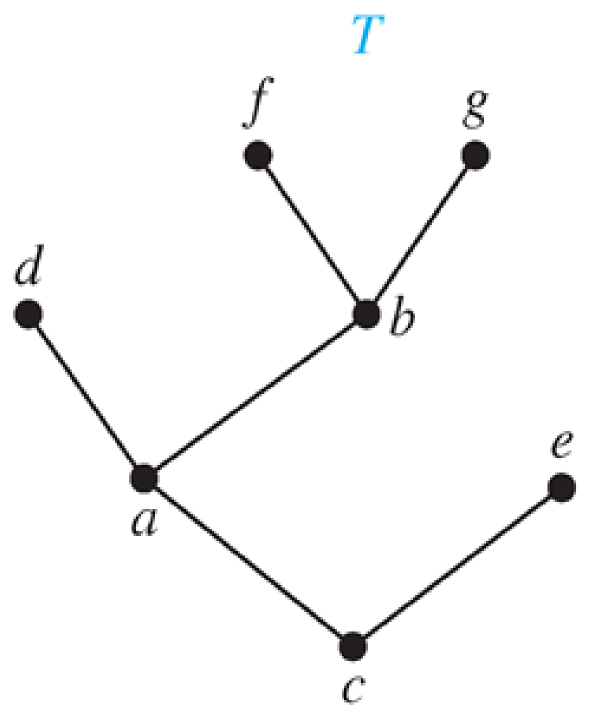
Rooted Trees

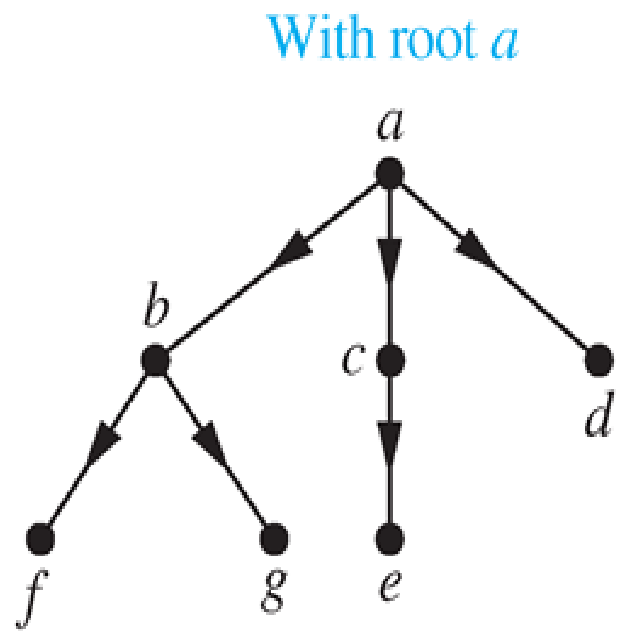
- **Definition** A *rooted tree* is a tree in which one vertex has been designated as the root and every edge is directed away from the root.



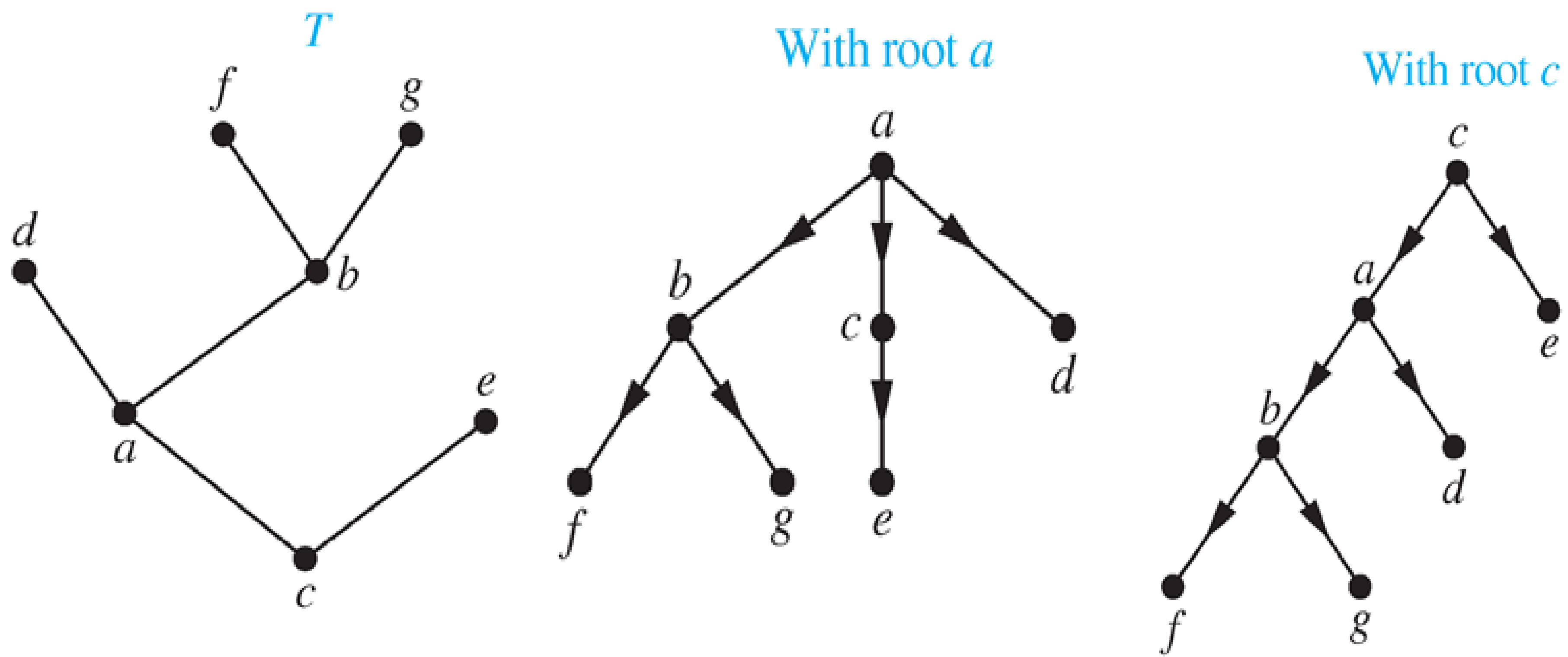
Rooted Trees

- **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.





- **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.



Rooted Trees

- *parent, child, sibling*

Rooted Trees

- *parent, child, sibling*

  ancestor, descendant

Rooted Trees

- *parent, child, sibling*
  ancestor, descendant
  leaf, internal vertex

Rooted Trees

- *parent, child, sibling*

  ancestor, descendant

  leaf, internal vertex

  subtree with a as its root: consists of a and its descendants and all edges incident to these descendants

- **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. In particular, an m-ary tree with $m = 2$ is called a *binary tree*.

m-Ary 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. In particular, an m-ary tree with $m = 2$ is called a *binary tree*.

  Definition A binary tree is an *ordered rooted tree* where the children of each internal vertex are ordered. In a binary tree, the first child is called the *left child*, and the second child is called the *right child*.

- **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. In particular, an m-ary tree with $m = 2$ is called a *binary tree*.

  Definition A binary tree is an *ordered rooted tree* where the children of each internal vertex are ordered. In a binary tree, the first child is called the *left child*, and the second child is called the *right child*.

  left subtree, *right subtree*

Counting Vertices in a Full m-Ary Trees

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

Counting Vertices in a Full m-Ary Trees

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

  Theorem A full m-ary tree with

  (i) n vertices
  (ii) i internal vertices
  (iii) ℓ leaves

Counting Vertices in a Full m-Ary Trees

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

  Theorem A full m-ary tree with

  (i) n vertices
  (ii) i internal vertices
  (iii) ℓ leaves

  (i) n vertices, $i = (n-1)/m$, $\ell = [(m-1)n+1)]/m$
  (ii) i internal vertices, $n = mi + 1$, $\ell = (m-1)i + 1$
  (iii) ℓ leaves, $n = (m\ell - 1)/(m-1)$, $i = (\ell - 1)/(m-1)$

Counting Vertices in a Full m-Ary Trees

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

  Theorem A full m-ary tree with

  (i) n vertices
  (ii) i internal vertices
  (iii) ℓ leaves

  (i) n vertices, $i = (n-1)/m$, $\ell = [(m-1)n+1)]/m$
  (ii) i internal vertices, $n = mi + 1$, $\ell = (m-1)i + 1$
  (iii) ℓ leaves, $n = (m\ell - 1)/(m-1)$, $i = (\ell - 1)/(m-1)$

  using $n = mi + 1$ and $n = i + \ell$

Level and Height

- The *level* of a vertex v in a rooted tree is the length of the unique path from the root to this vertex.

Level and Height

- 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 and Height

- 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.

  Definition A rooted m-ary tree of height h is *balanced* if all leaves are at levels h or $h-1$. (differ no greater than 1)

The Number of Leaves

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

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

  Proof (by induction)

The Number of Leaves

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

  Proof (by induction)

  Corollary If an m-ary tree of height h has ℓ leaves, then $h \geq \lceil \log_m \ell \rceil$. If the m-ary tree is full and balanced, then $h = \lceil \log_m \ell \rceil$.

The Number of Leaves

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

  Proof (by induction)

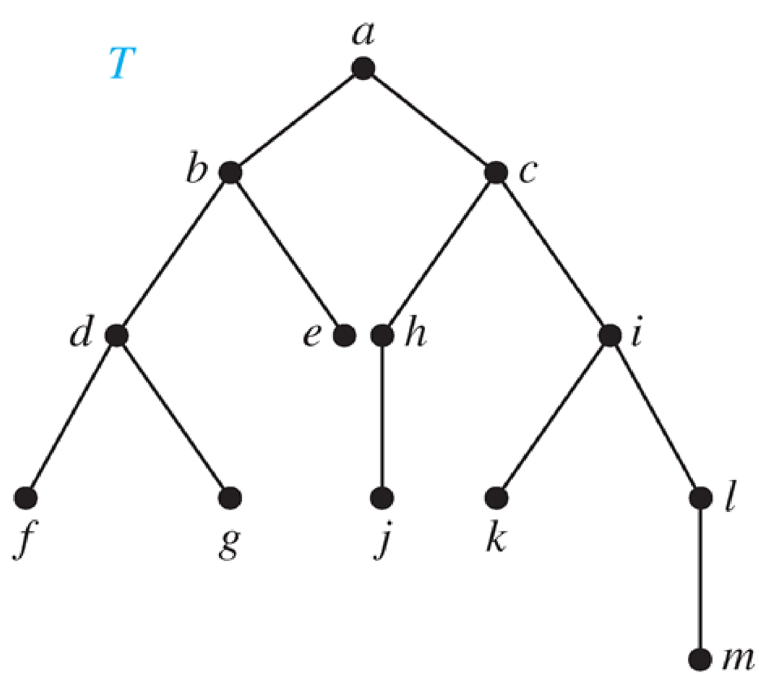
  Corollary If an m-ary tree of height h has ℓ leaves, then $h \geq \lceil \log_m \ell \rceil$. If the m-ary tree is full and balanced, then $h = \lceil \log_m \ell \rceil$.

  Proof

Binary Trees

- **Definition** A *binary tree* is an ordered rooted tree where each internal tree has two children, the first is called the *left child* and the second is 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.

- **Definition** A *binary tree* is an ordered rooted tree where each internal tree has two children, the first is called the *left child* and the second is 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.



Tree Traversal

- The procedures for systematically visiting every vertex of an ordered tree are called *traversals*.

Tree Traversal

- The procedures for systematically visiting every vertex of an ordered tree are called *traversals*.

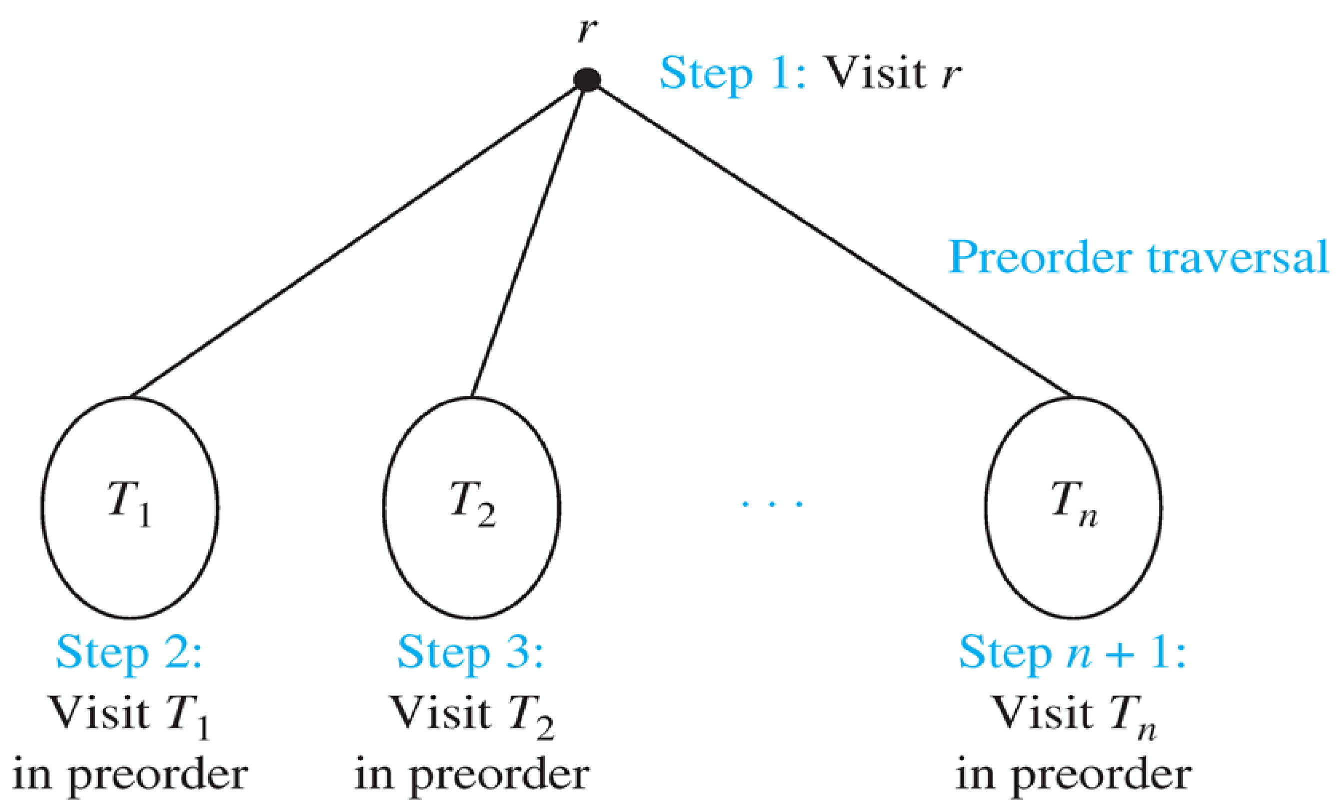
  The three most commonly used traversals are *preorder traversal*, *inorder traversal*, *postorder traversal*.

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 $T_1, T_2, \ldots, T_n$ 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.

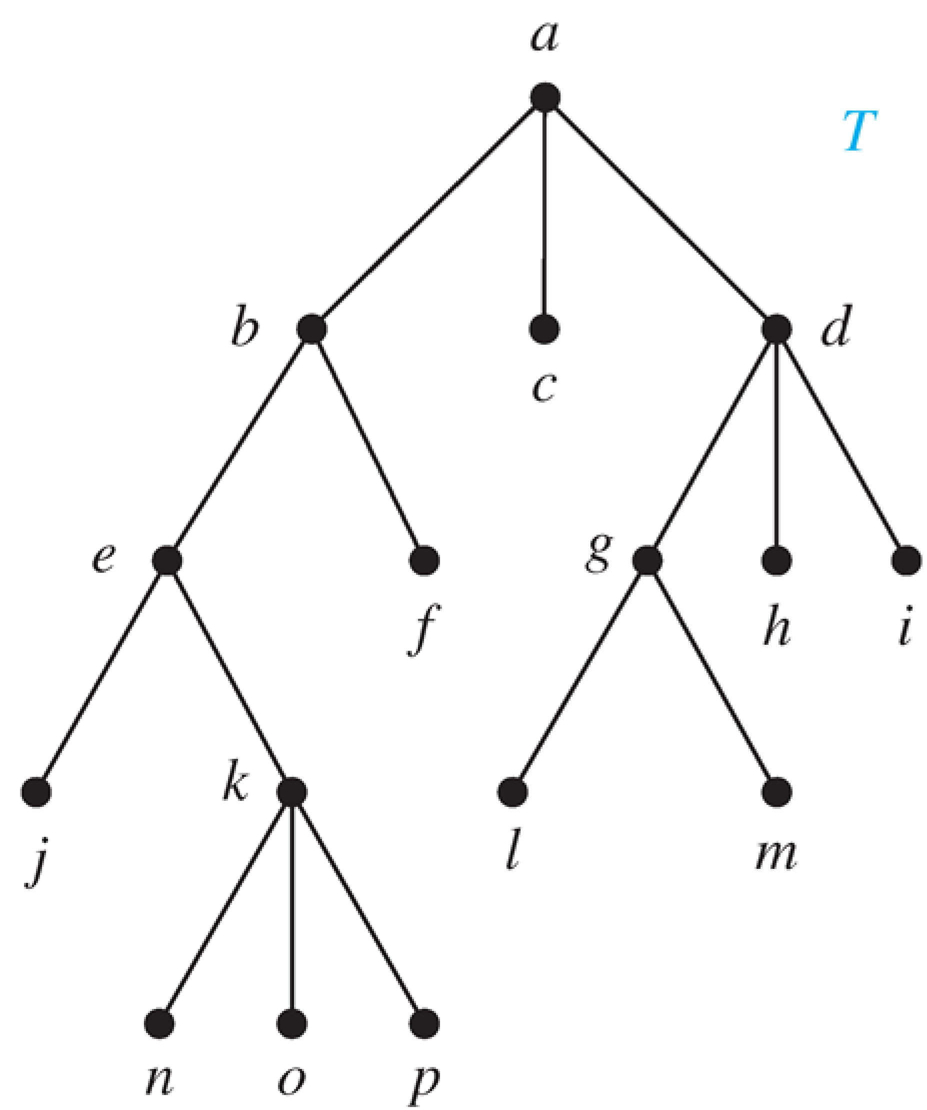
Preorder Traversal

- **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 $T_1, T_2, \ldots, T_n$ 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.



Preorder Traversal

- **Example**



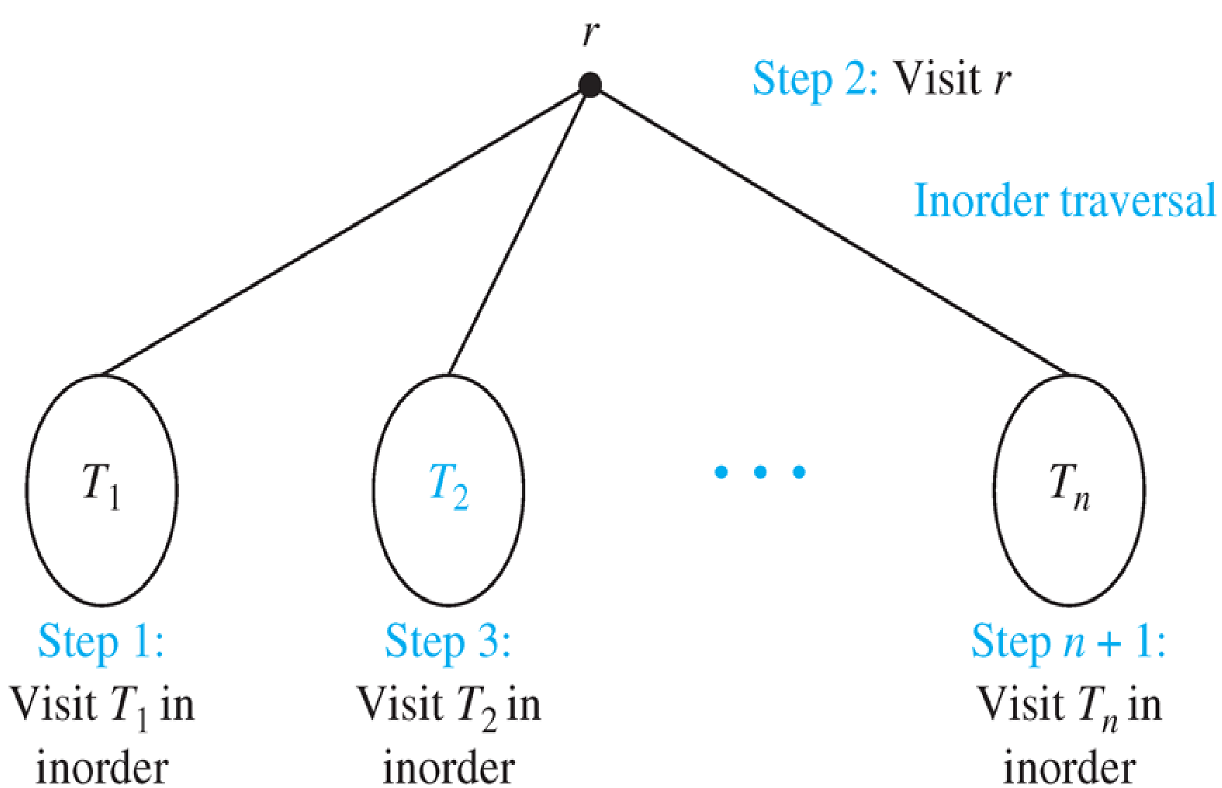
Preorder Traversal

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

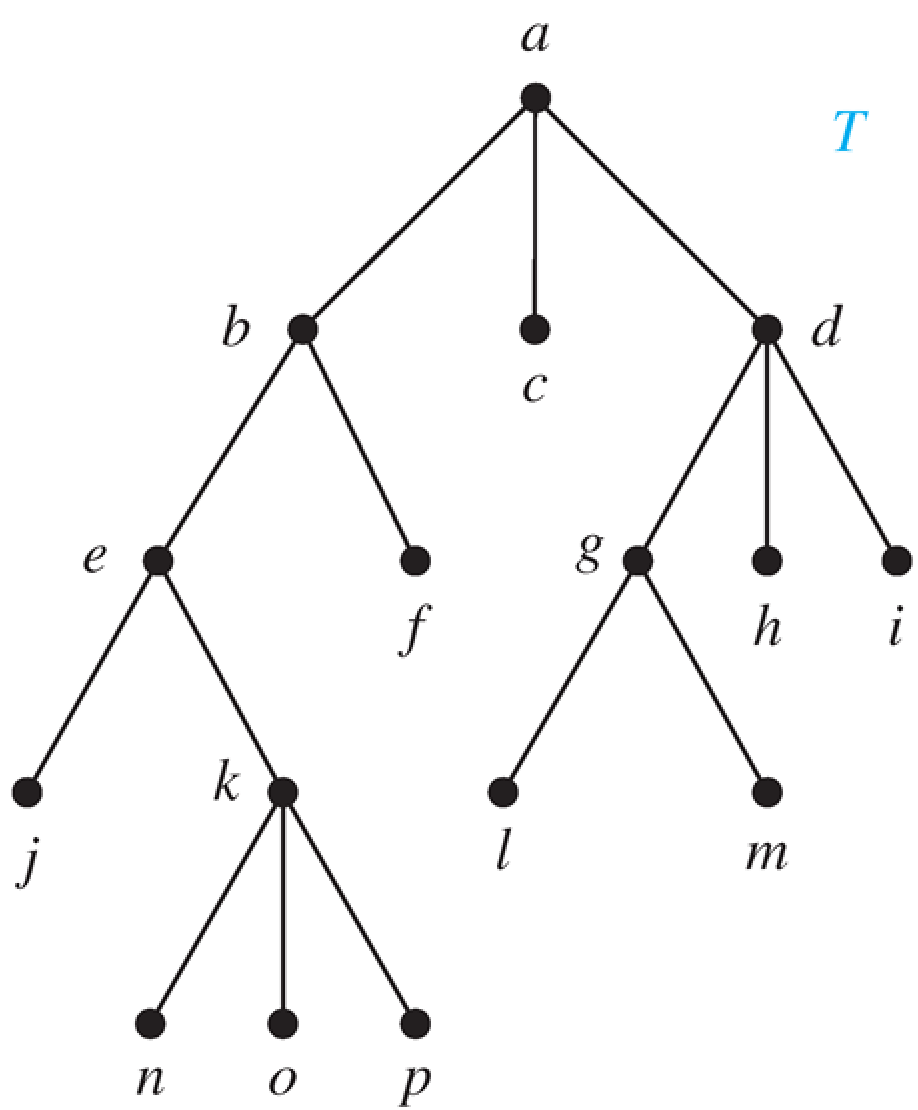
- **Definition** Let T be an ordered rooted tree with root r. If T consists only of r, then r is the *inorder traversal* of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees of r from left to right in T. The *inorder traversal* begins by traversing T_1 in inorder, then visiting r, and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

- **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 $T_1, T_2, \ldots, T_n$ 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.



Inorder Traversal

- **Example**



Inorder Traversal

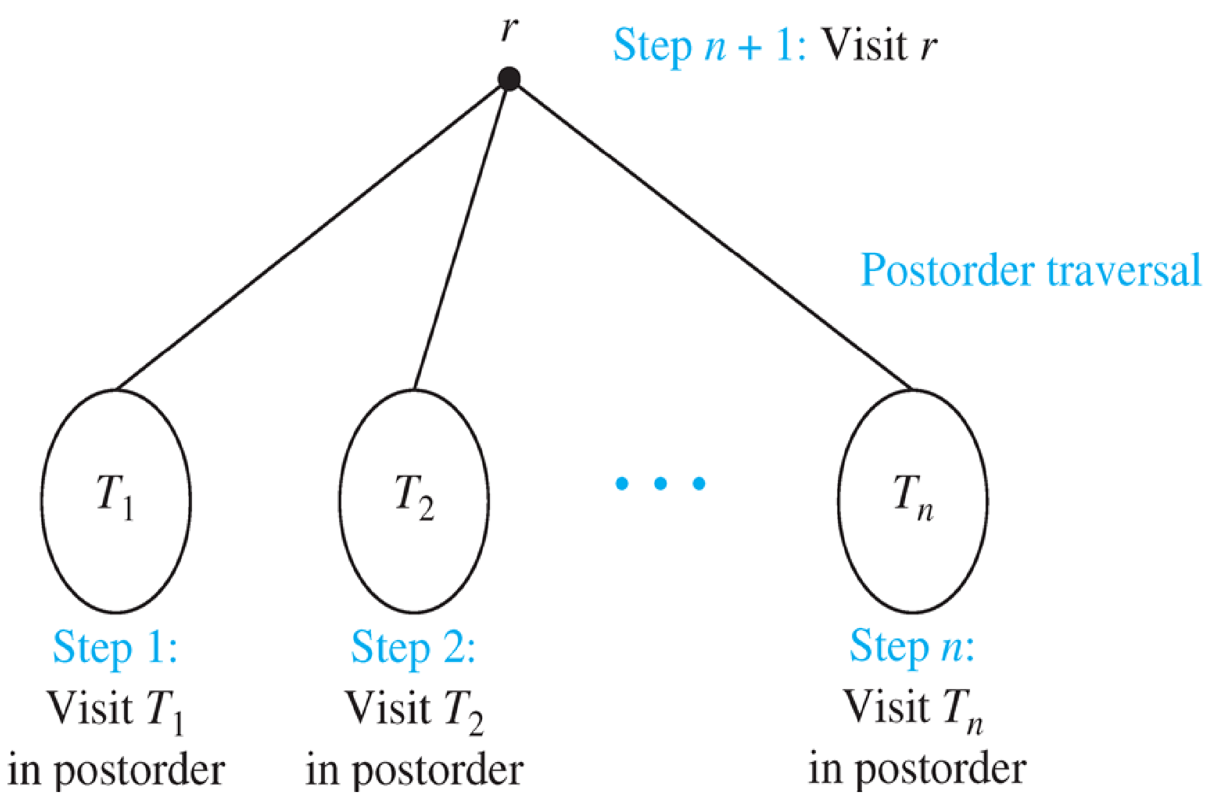
procedure *inorder* (T: ordered rooted tree)
r := root of T
if r is a leaf **then** list r
else
  l := first child of r from left to right
  $T(l)$:= subtree with l as its root
  inorder($T(l)$)
  list(r)
  for each child c of r from left to right
    $T(c)$:= subtree with c as root
    inorder($T(c)$)

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 $T_1, T_2, \ldots, T_n$ 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.

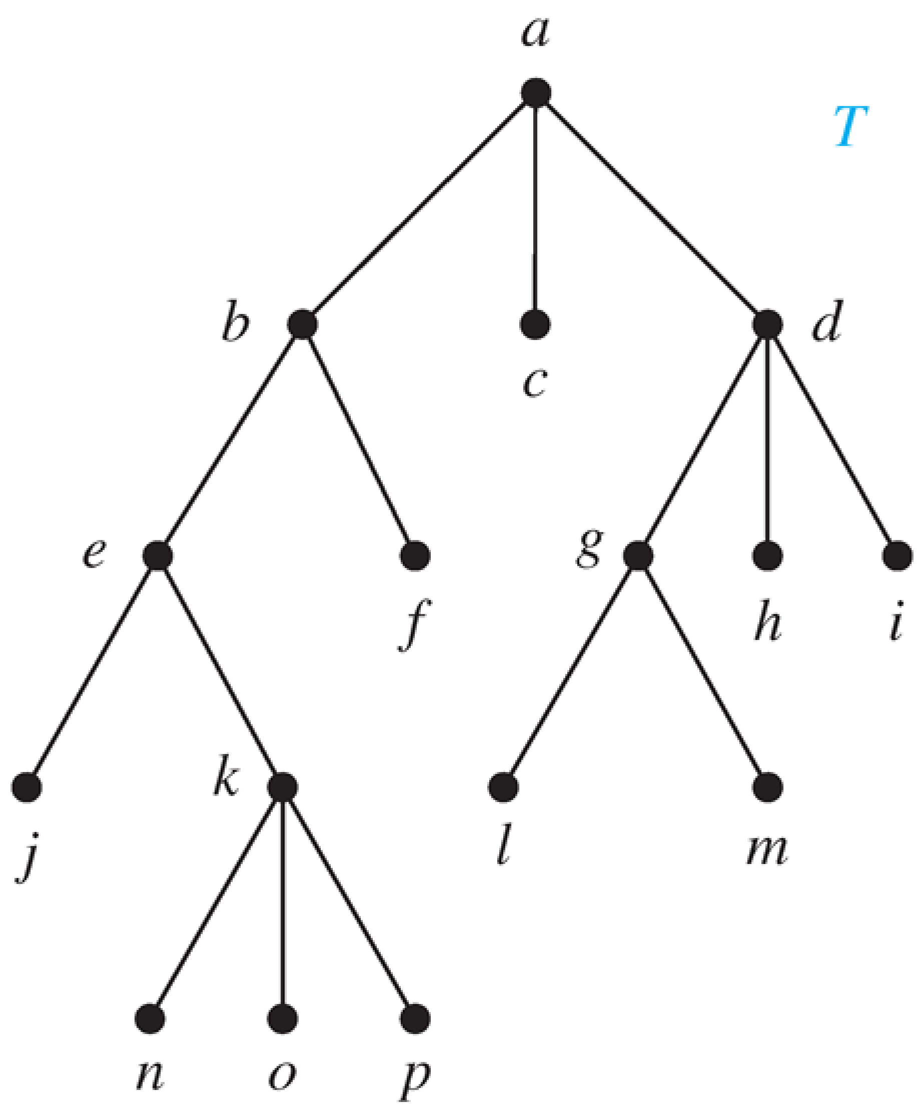
Postorder Traversal

- **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 $T_1, T_2, \ldots, T_n$ 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.



Postorder Traversal

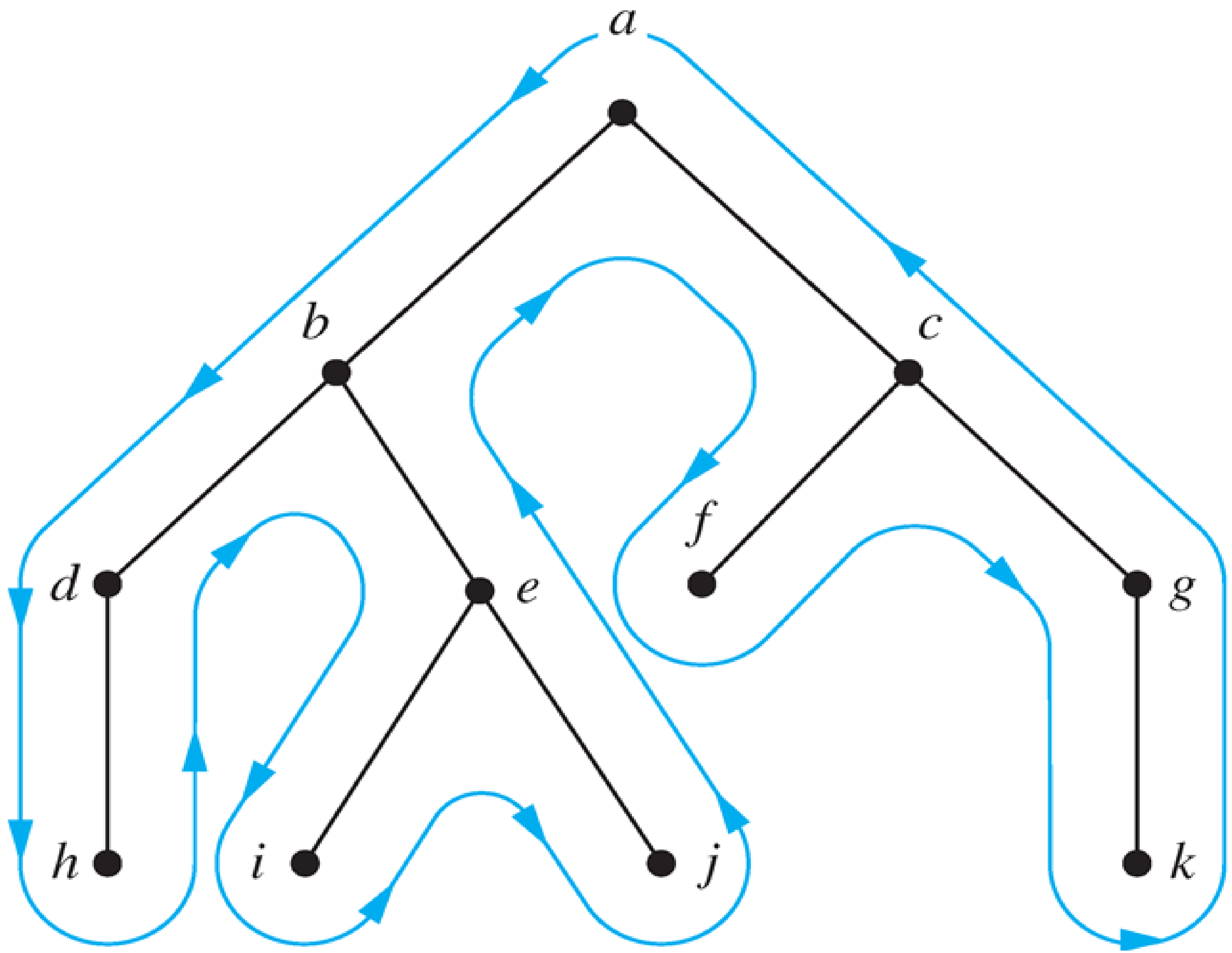
- **Example**



Postorder Traversal

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

Preorder, Inorder, Postorder Traversal



Expression Trees

- Complex expressions can be represented using ordered rooted trees

Expression Trees

- Complex expressions can be represented using ordered rooted trees

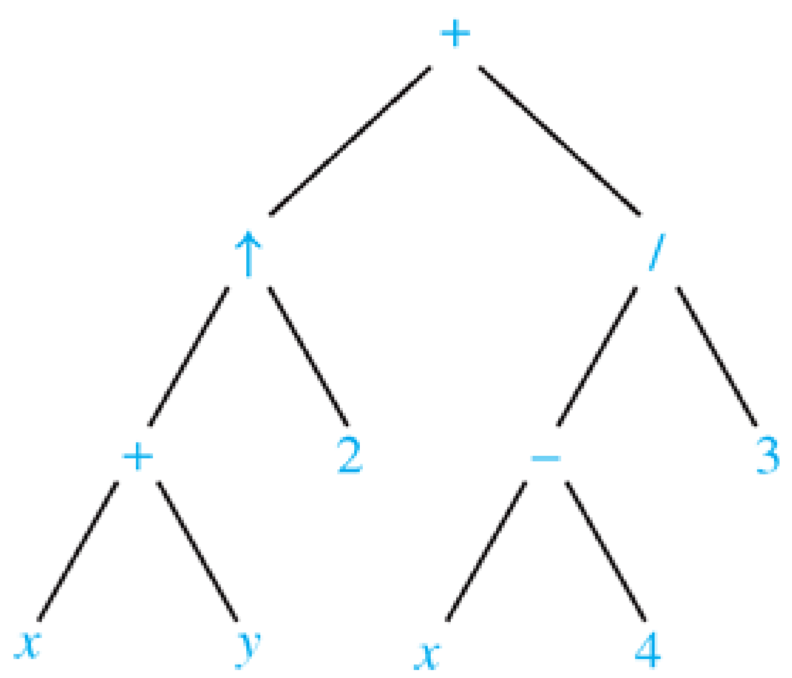
Example

consider the expression $((x + y) \uparrow 2) + ((x - 4)/3)$

- Complex expressions can be represented using ordered rooted trees

Example

consider the expression $((x + y) \uparrow 2) + ((x - 4)/3)$



Infix Notation

- An inorder traversal of the tree representing an expression produces the original expression when parentheses are included except for unary operation.

Infix Notation

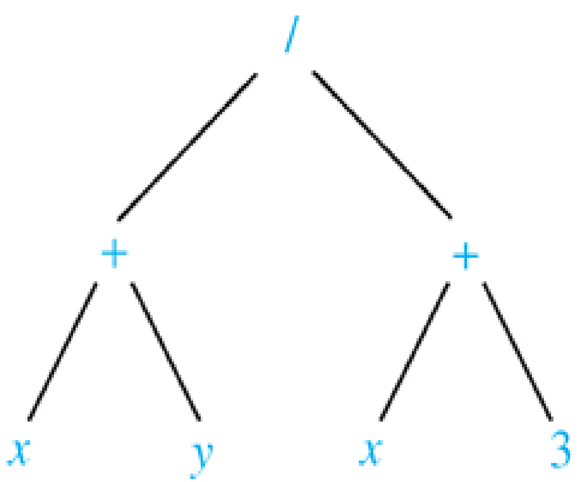
- An inorder traversal of the tree representing an expression produces the original expression when parentheses are included except for unary operation.

  Why parentheses are needed?

Infix Notation

- An inorder traversal of the tree representing an expression produces the original expression when parentheses are included except for unary operation.

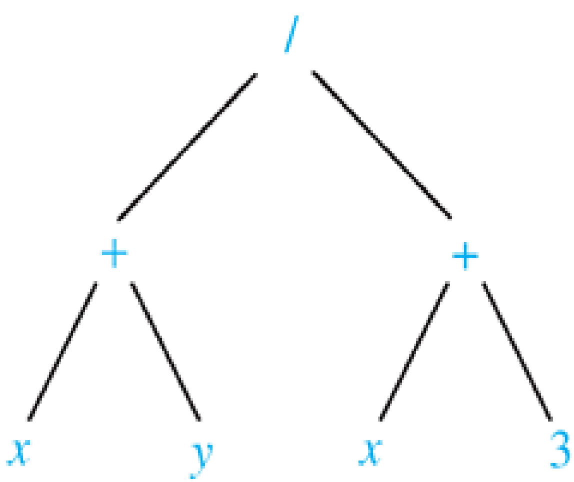
  Why parentheses are needed?

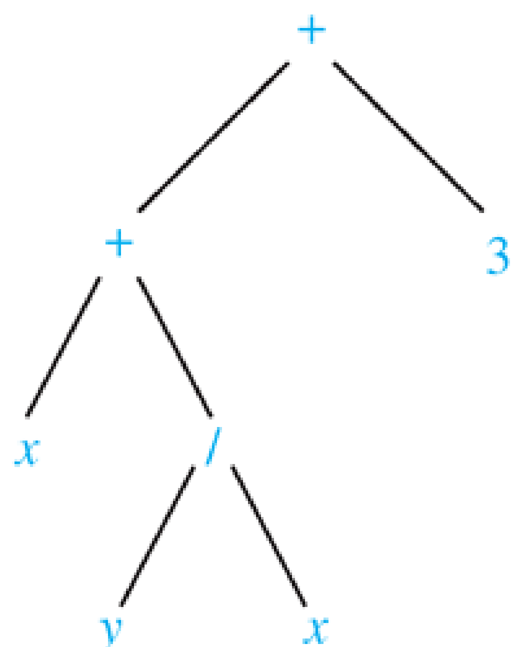


Infix Notation

- An inorder traversal of the tree representing an expression produces the original expression when parentheses are included except for unary operation.

  Why parentheses are needed?

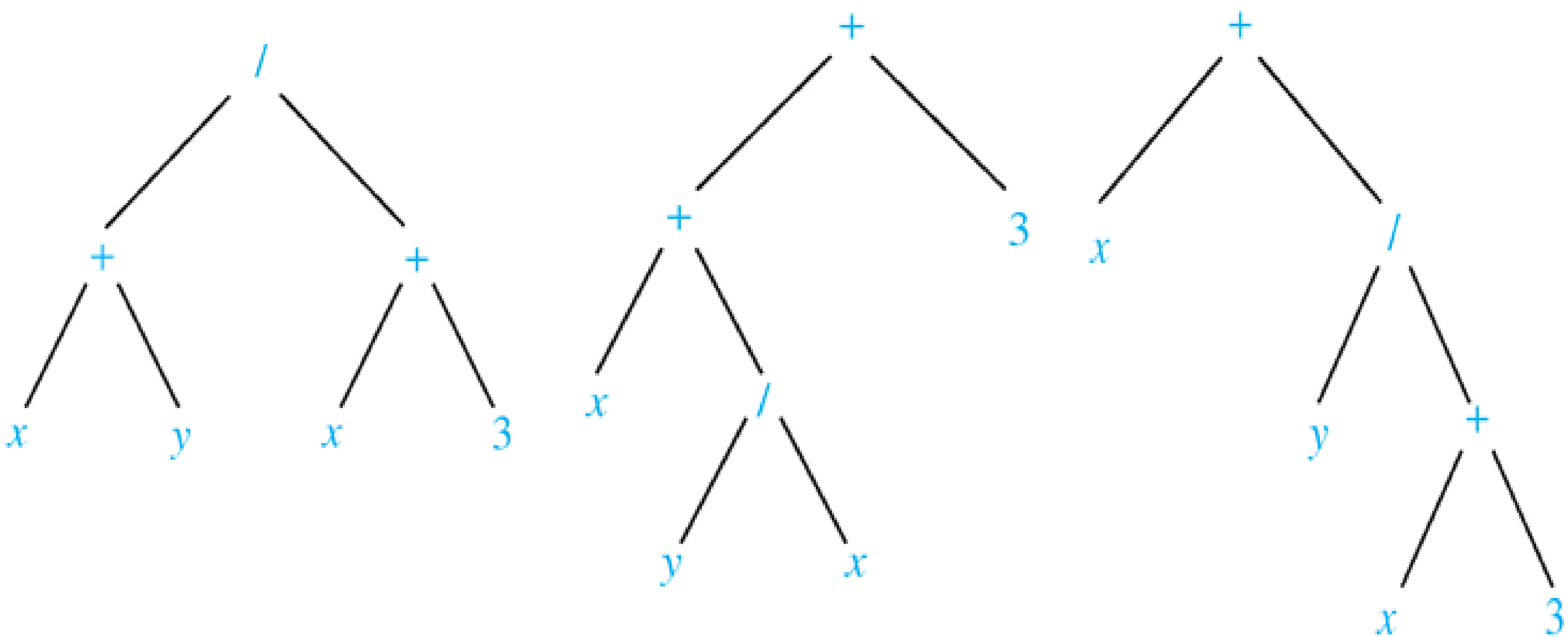




Infix Notation

- An inorder traversal of the tree representing an expression produces the original expression when parentheses are included except for unary operation.

  Why parentheses are needed?



Prefix Notation

- The preorder traversal of expression trees leads to the *prefix form* of the expression (*Polish notation*).

Prefix Notation

- The preorder traversal of expression trees leads to the *prefix form* of the expression (*Polish notation*).

  Operators precede their operands in the prefix notation. Parentheses are not needed as the representation is unambiguous.

Prefix Notation

- The preorder traversal of expression trees leads to the *prefix form* of the expression (*Polish notation*).

  Operators precede their operands in the prefix notation. Parentheses are not needed as the representation is unambiguous.

  Prefix expressions are evaluated by working from right to left. When we encounter an operator, we perform the operation with the two operands to the right.

Prefix Notation

- **Example**

  $+ \; - \; * \, 2 \; 3 \; 5 \; / \; \uparrow \; 2 \; 3 \; 4$

Prefix Notation

- **Example**

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

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

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

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

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

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

Postfix Notation

- The postorder traversal of expression trees leads to the *postfix form* of the expression (*reverse Polish notation*).

Postfix Notation

- The postorder traversal of expression trees leads to the *postfix form* of the expression (*reverse Polish notation*).

  Operators follow their operands in the postfix notation. Parentheses are not needed as the representation is unambiguous.

Postfix Notation

- The postorder traversal of expression trees leads to the *postfix form* of the expression (*reverse Polish notation*).

  Operators follow their operands in the postfix notation. Parentheses are not needed as the representation is unambiguous.

  Postfix expressions are evaluated by working from left to right. When we encounter an operator, we perform the operation with the two operands to the left.

Postfix Notation

- **Example**

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

Postfix Notation

- **Example**

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

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

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

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

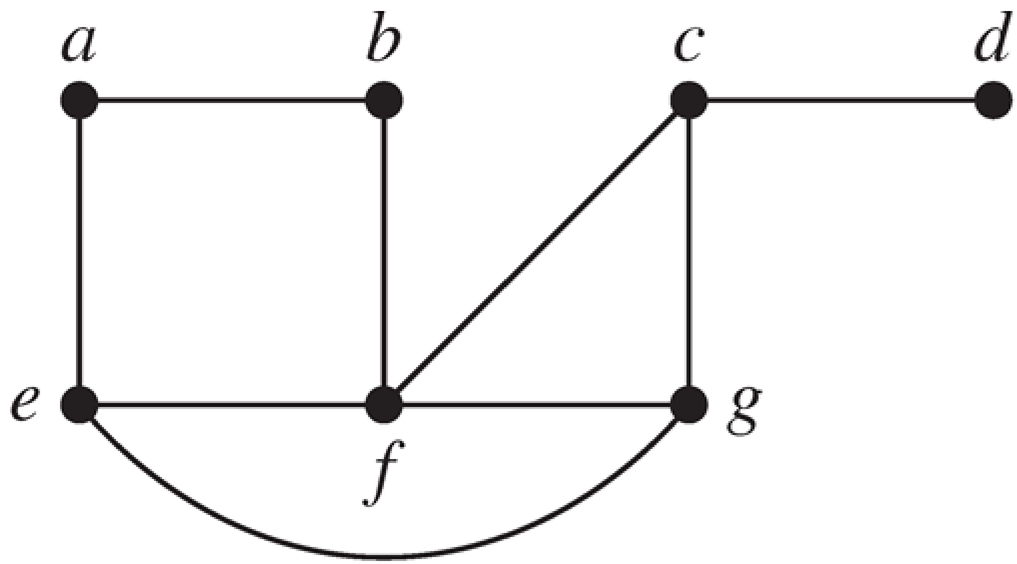
$$1\ \underbrace{9\ 3\ /}_{9/3=3}\ +$$

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

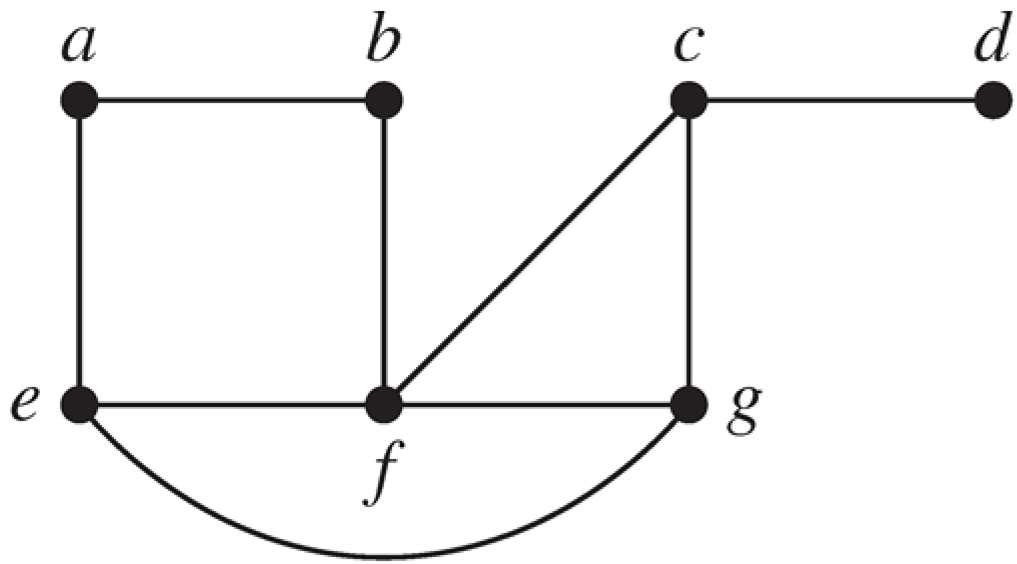
Spanning Trees

- **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.

- **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.



- **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.



remove edges to avoid circuits

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

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

Proof

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

Proof

"only if" part

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

  Proof

  “only if ” part

  The spanning tree can be obtained by removing edges from simple circuits.

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

Proof

"only if " part

The spanning tree can be obtained by removing edges from simple circuits.

"if " part

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

Proof

"only if" part

The spanning tree can be obtained by removing edges from simple circuits.

"if" part

easy

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.
  Instead, we build up spanning trees by successively adding edges.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.
  Instead, we build up spanning trees by successively adding edges.
  - ◇ First arbitrarily choose a vertex of the graph as the root.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.
  Instead, we build up spanning trees by successively adding edges.

  ◇ First arbitrarily choose a vertex of the graph as the root.

  ◇ Form a path by successively adding vertices and edges.
  Continue adding to this path as long as possible.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.
  Instead, we build up spanning trees by successively adding edges.

  ◇ First arbitrarily choose a vertex of the graph as the root.

  ◇ Form a path by successively adding vertices and edges. Continue adding to this path as long as possible.

  ◇ If the path goes through all vertices of the graph, the tree is a spanning tree.

Depth-First Search

- We can find spanning trees by removing edges from simple circuits.
  But, this is inefficient, since simple circuits should be identified first.
  Instead, we build up spanning trees by successively adding edges.

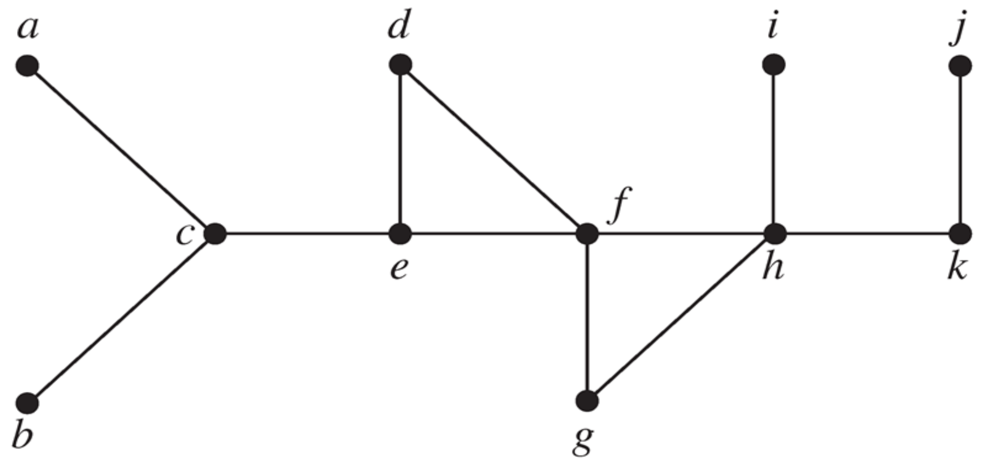
  ◇ First arbitrarily choose a vertex of the graph as the root.

  ◇ Form a path by successively adding vertices and edges. Continue adding to this path as long as possible.

  ◇ If the path goes through all vertices of the graph, the tree is a spanning tree.

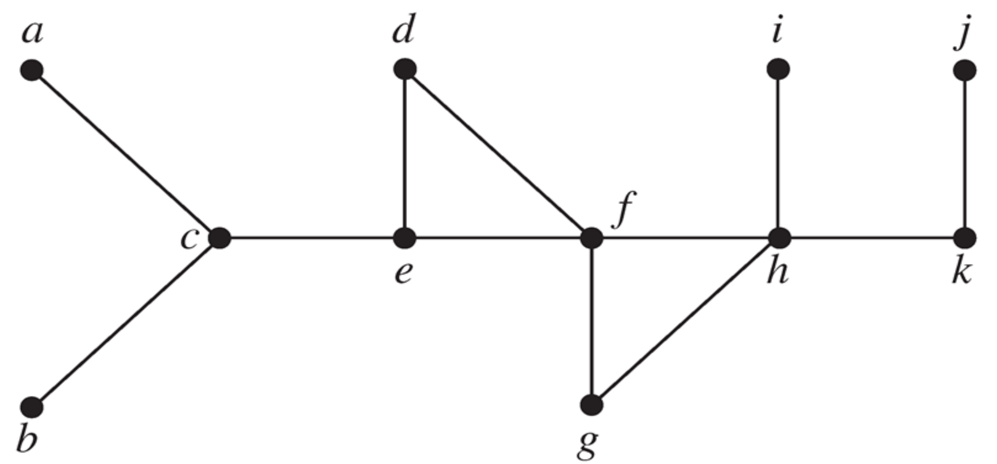
  ◇ Otherwise, move back to some vertex to repeat this procedure (*backtracking*)

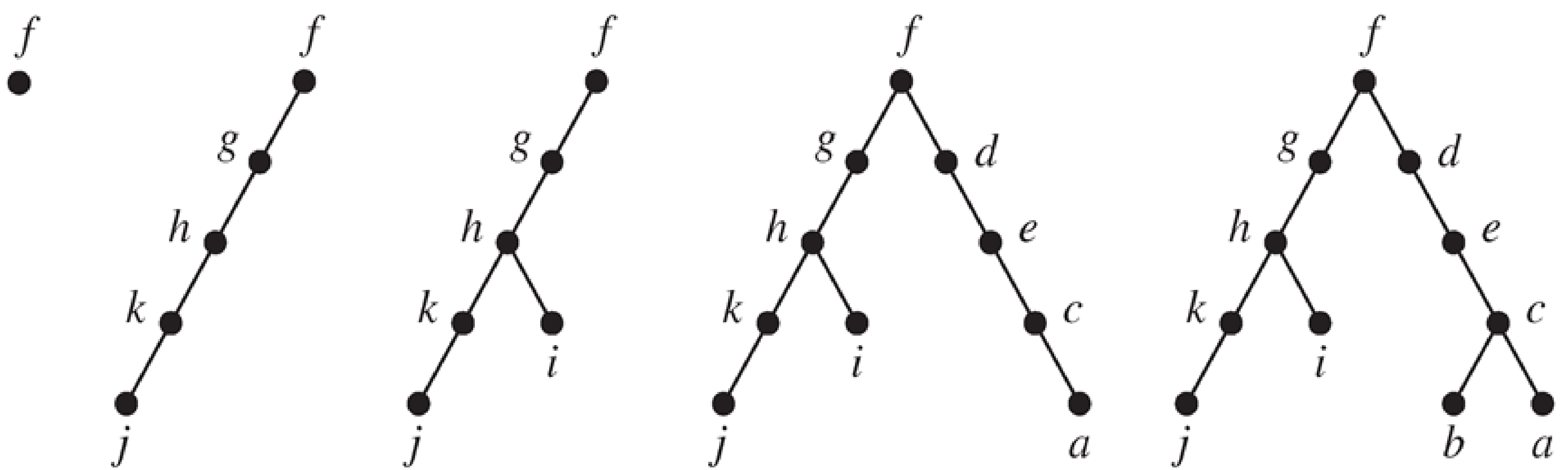
- **Example**



Depth-First Search

- **Example**





Depth-First Search Algorithm

procedure *DFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
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
  add vertex w and edge $\{v, w\}$ to T
  visit(w)

Depth-First Search Algorithm

procedure *DFS*(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
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
  add vertex w and edge $\{v, w\}$ to T
  visit(w)

time complexity: $O(e)$

Breadth-First Search

- This is the second algorithm that we build up spanning trees by successively adding edges.

- This is the second algorithm that we build up spanning trees by successively adding edges.

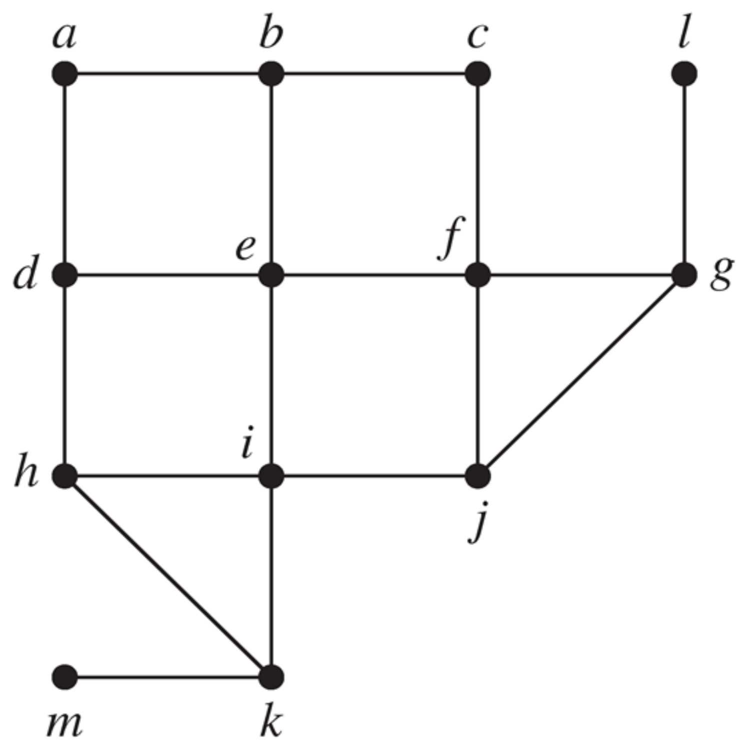
  ◇ First arbitrarily choose a vertex of the graph as the root.

  ◇ Form a path by adding all edges incident to this vertex and the other endpoint of each of these edges

  ◇ For each vertex added at the previous level, add edge incident to this vertex, as long as it does not produce a simple circuit.

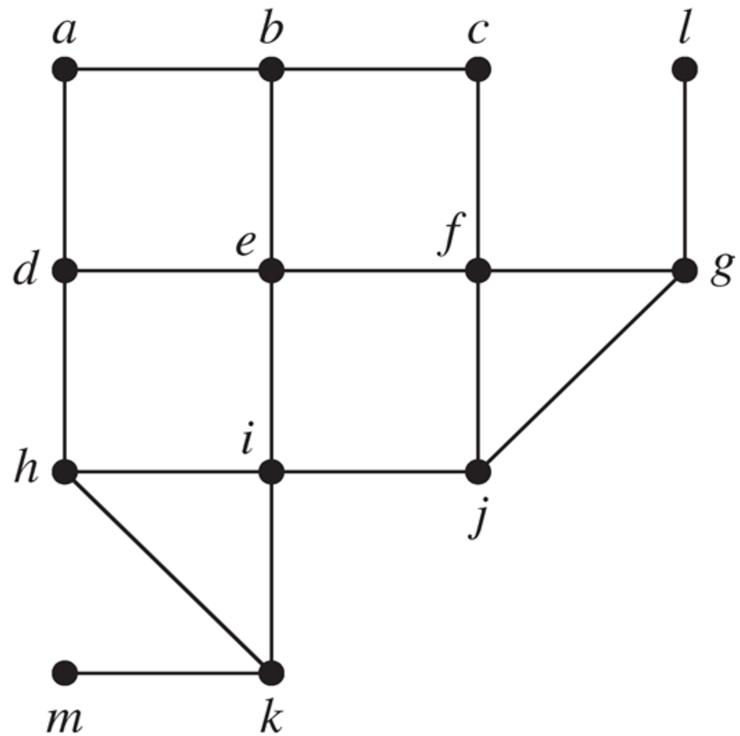
  ◇ Continue in this manner until all vertices have been added.

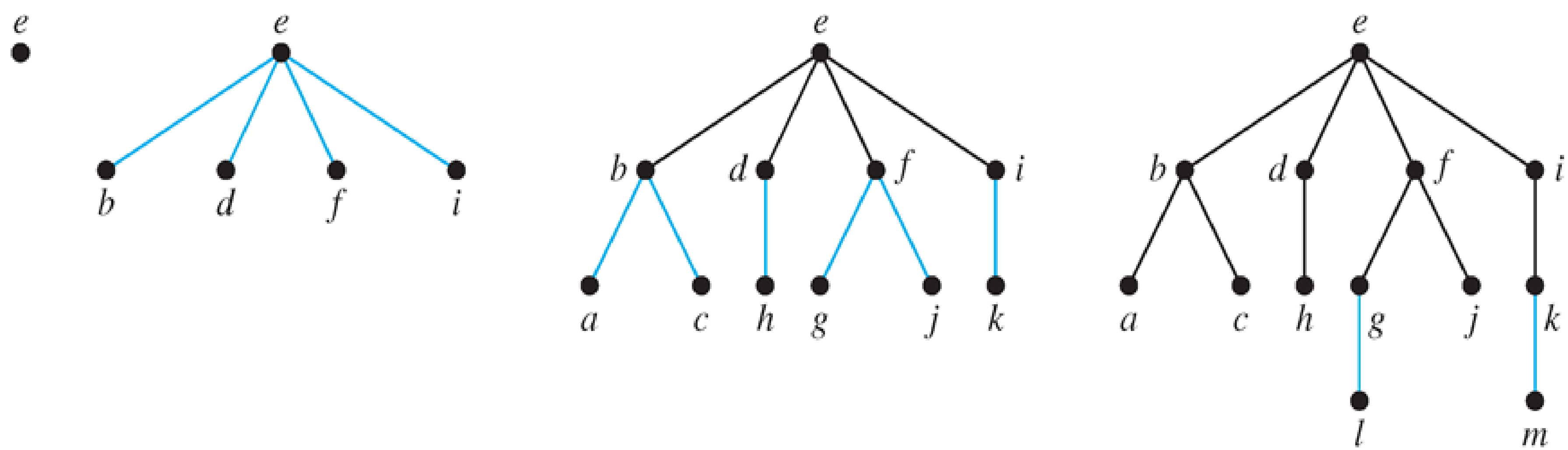
- **Example**



Breadth-First Search

- **Example**





Breadth-First Search

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

Breadth-First Search

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

time complexity: $O(e)$

Applications of DFS, BFS

- find paths, circuits, connected components, cut vertices, ...

Applications of DFS, BFS

- find paths, circuits, connected components, cut vertices, ...

  find shortest paths, determine whether bipartite, ...

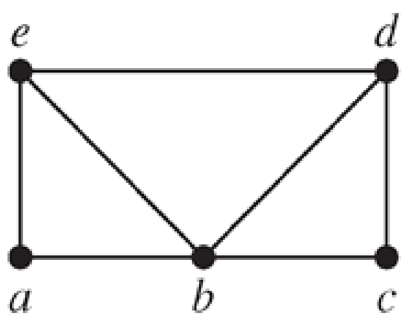
Applications of DFS, BFS

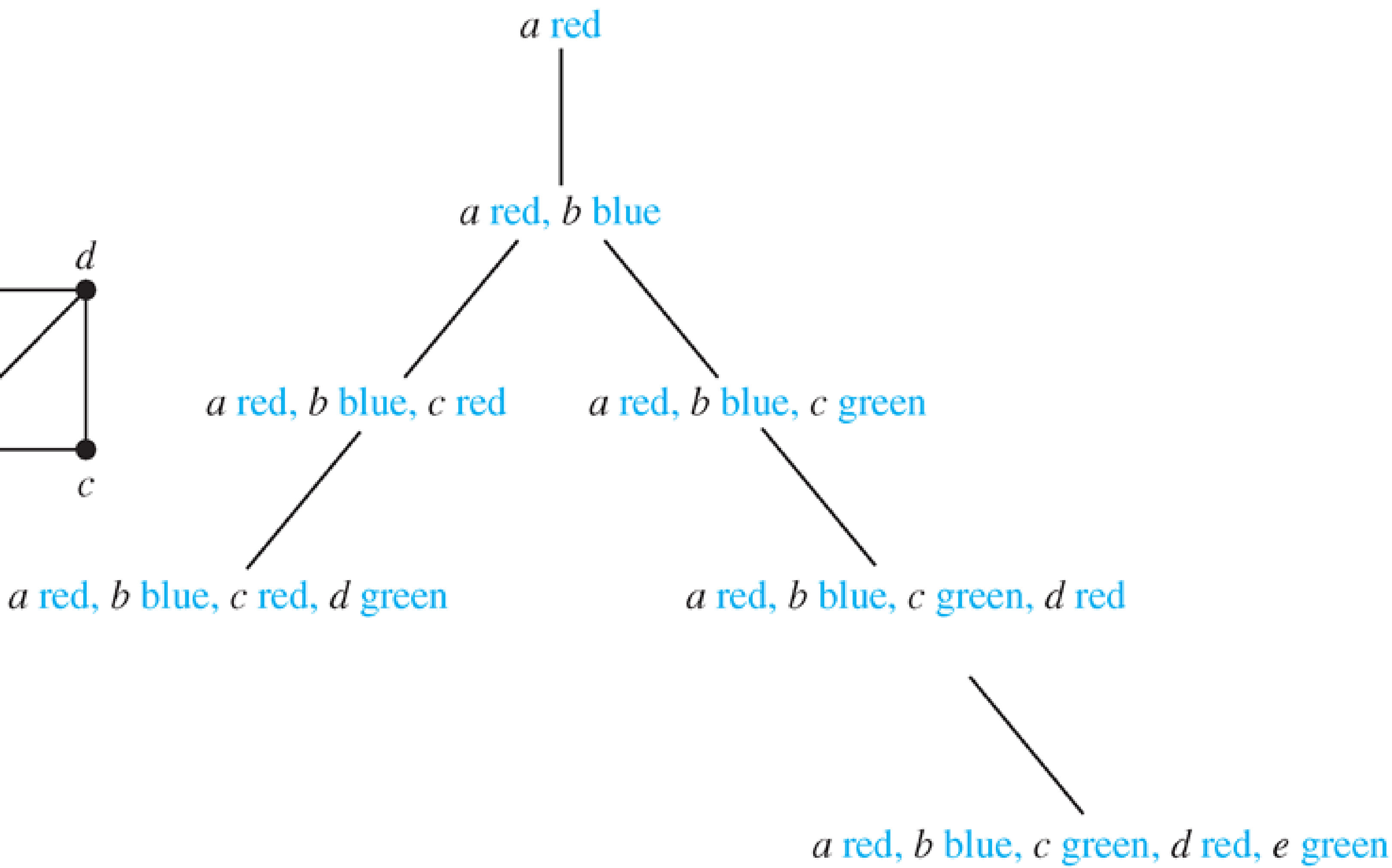
- find paths, circuits, connected components, cut vertices, ...

  find shortest paths, determine whether bipartite, ...

  graph coloring, sums of subsets, ...

Applications of DFS, BFS



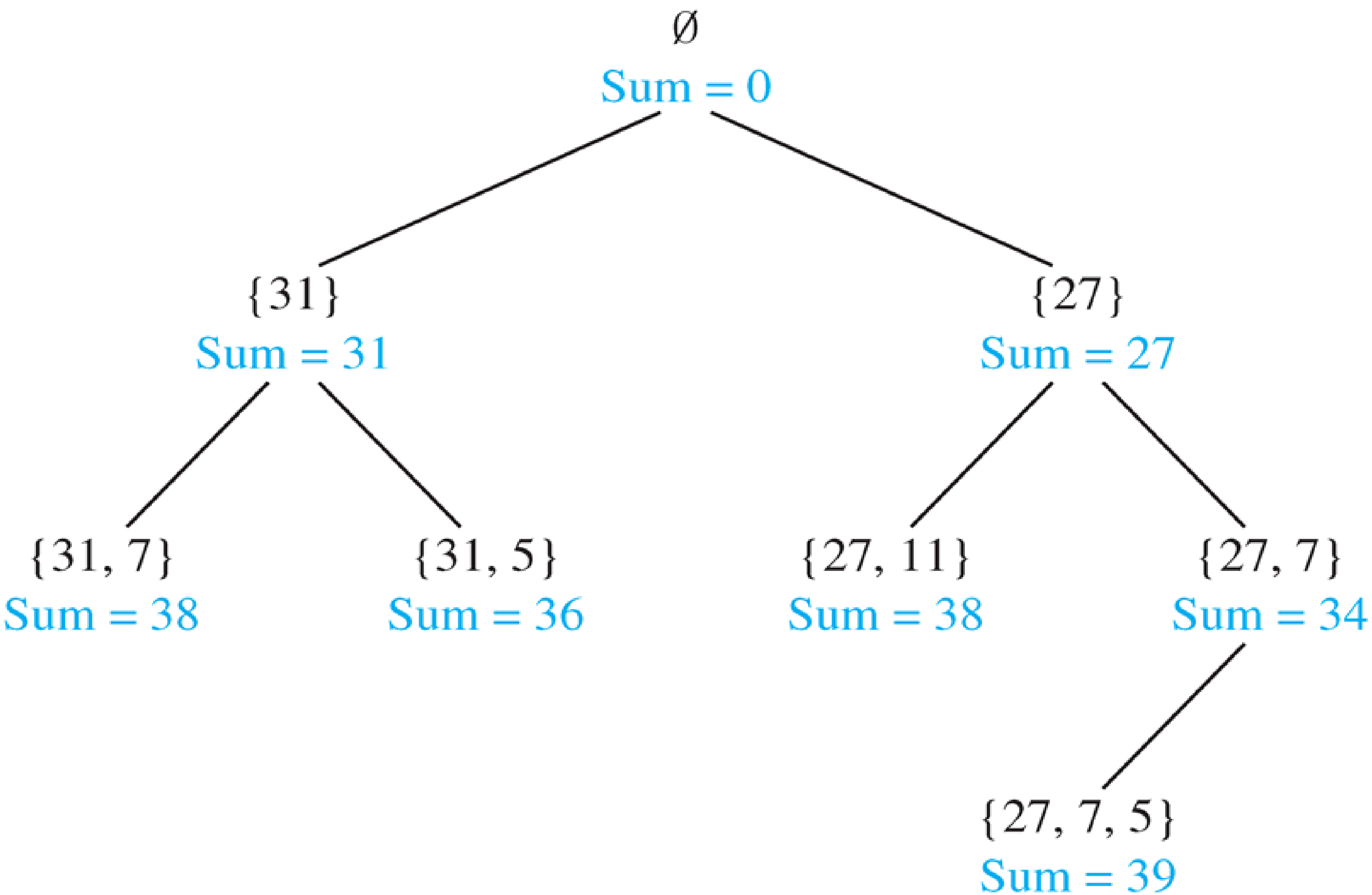


Applications of DFS, BFS

- find paths, circuits, connected components, cut vertices, ...

  find

  grap



find a subset of $\{31, 27, 15, 11, 7, 5\}$ with the sum 39

Minimum Spanning Trees

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

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

  two greedy algorithms: Prim's Algorithm, Kruscal's Algorithm

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}

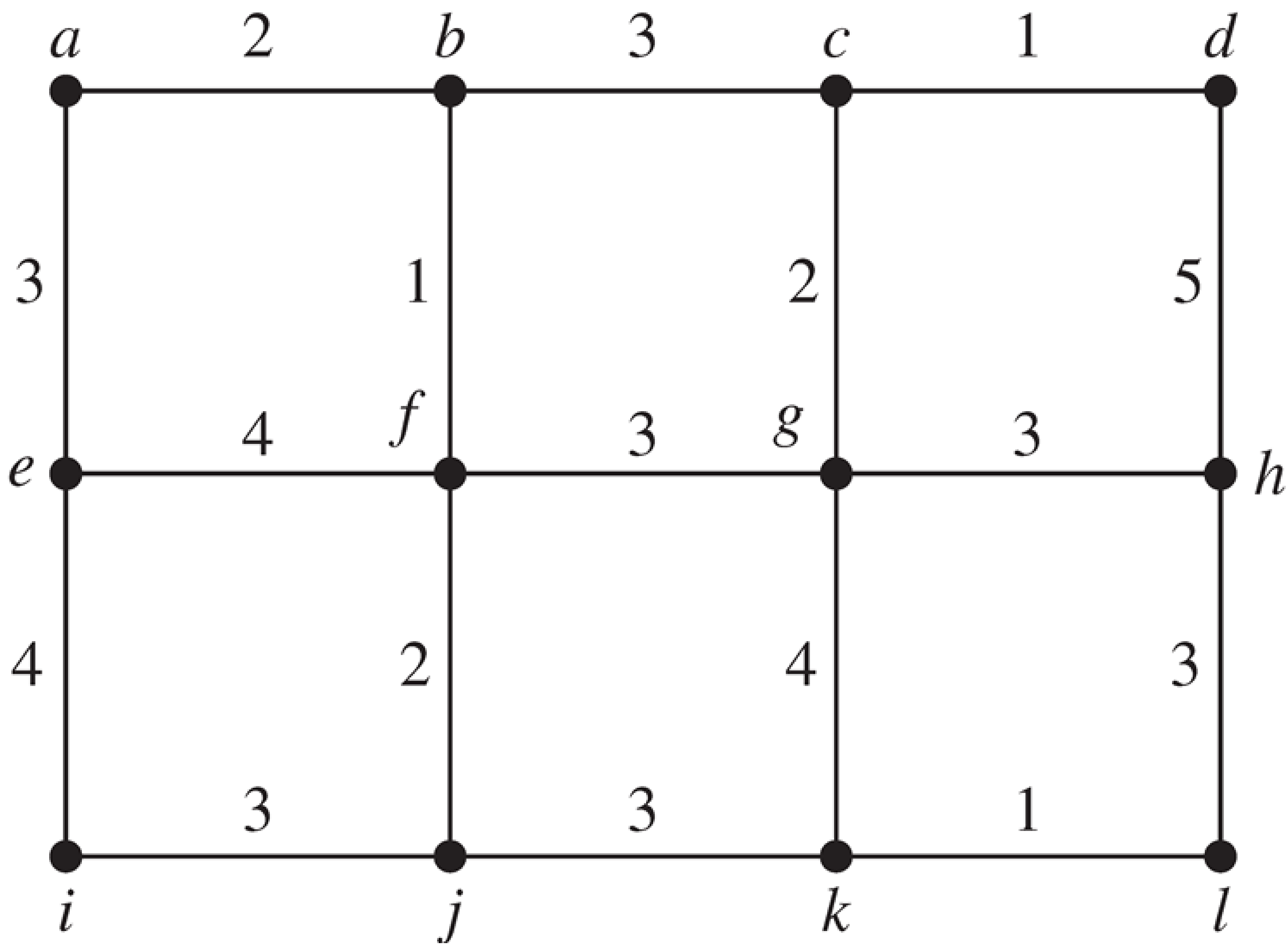
Prim's Algorithm

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}

time complexity: $e \log v$

- **Example**



Kruscal'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 $G\}$

Kruscal'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}

time complexity: $e \log e$

Kruscal's Algorithm

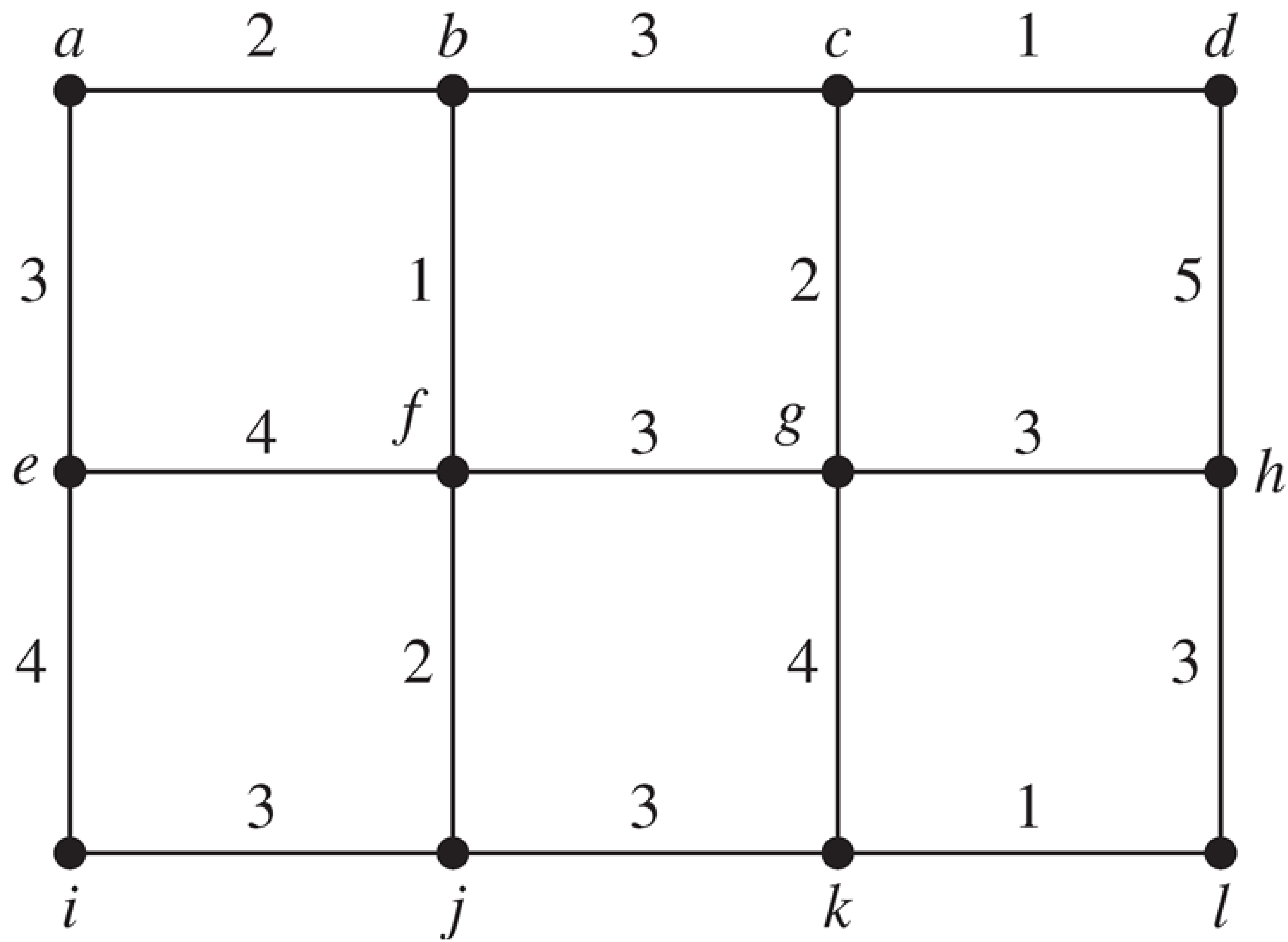
ALGORITHM 2 Kruskal's Algorithm.

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}

time complexity: $e \log e$
see *CLRS* / *Algorithm Design*, J. Kleinberg, E. Tardos

Kruscal's Algorithm

- **Example**



Next Lecture

- course review ...