CS201 DISCRETE MATHEMATICS FOR COMPUTER SCIENCE

Dr. QI WANG

Department of Computer Science and Engineering
Office: Room903, Nanshan iPark A7 Building
Email: wangqi@sustech.edu.cn

Zero Knowledge Proofs

- **Problem**: I know that P is true, and I want to convince you of that. I try to present all the facts I know and the inferences from the facts that imply P is true.

Zero Knowledge Proofs

- **Problem**: I know that P is true, and I want to convince you of that. I try to present all the facts I know and the inferences from the facts that imply P is true.

  Example: P : 26781 is not a prime since $26781 = 113 \times 237$.

Zero Knowledge Proofs

- **Problem**: I know that P is true, and I want to convince you of that. I try to present all the facts I know and the inferences from the facts that imply P is true.

  Example: $P:$ 26781 is not a prime since $26781 = 113 \times 237$.

  Given this factorization, other than that you are convinced that P is true, you gained some knowledge (the factorization).

Zero Knowledge Proofs

- **Problem**: I know that P is true, and I want to convince you of that. I try to present all the facts I know and the inferences from the facts that imply P is true.

  Example: P : 26781 is not a prime since $26781 = 113 \times 237$.

  Given this factorization, other than that you are convinced that P is true, you gained some knowledge (the factorization).

  In a *Zero Knowledge Proof*, Alice will prove to Bob that a statement P is true. Bob will be completely convinced that P is true, but will not learn anything as a result of this process. That is, Bob will gain zero knowledge.

Applications of ZKPs

- *Protocol design*. A *protocol* is an algorithm for interactive parties to achieve a certain goal. However, in crypto, we often want to design protocols that should achieve security even when one of the parties is "cheating". Alice can prove in *zero knowledge* that she followed the instructions.

Applications of ZKPs

- *Protocol design*. A *protocol* is an algorithm for interactive parties to achieve a certain goal. However, in crypto, we often want to design protocols that should achieve security even when one of the parties is "cheating". Alice can prove in *zero knowledge* that she followed the instructions.

Proofs that Yield Nothing But their Validity and a Methodology of Cryptographic Protocol Design

(Extended Abstract)

Oded Goldreich
Dept. of Computer Sc.
Technion
Haifa, Israel

Silvio Micali
Lab. for Computer Sc.
MIT
Cambridge, MA 02139

Avi Wigderson
Inst. of Math. and CS
Hebrew University
Jerusalem, Israel

Applications of ZKPs

- *Identification scheme*. How should Alice prove to Bob that she is who she claimed to be? For example, how to design a control access system to the CSE dept.?

Applications of ZKPs

- *Identification scheme*. How should Alice prove to Bob that she is who she claimed to be? For example, how to design a control access system to the CSE dept.?

  A direct solution is to have a box on the door and give authorized people a secret PIN number. However, a drawback is that the box remains outside all the time and if someone could examine the box, they would perhaps be able to view its memory and extract the secrets keys of all people.

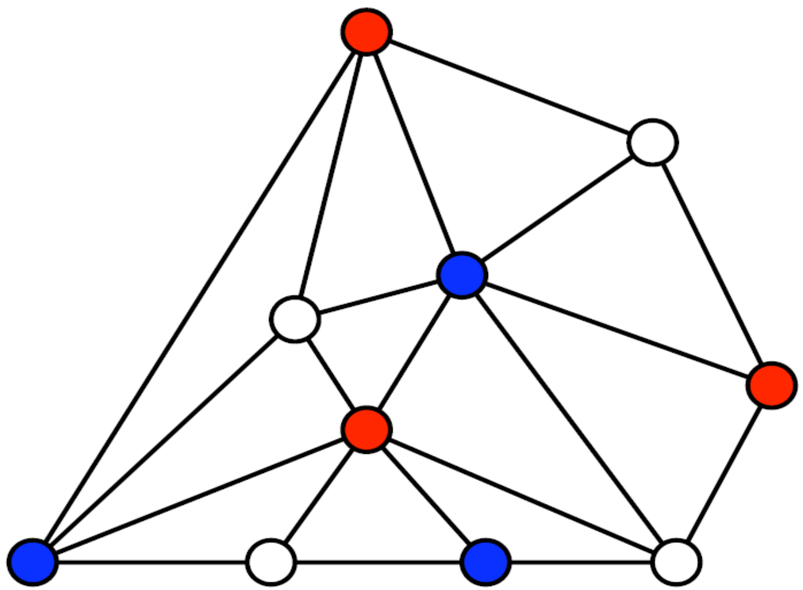
Applications of ZKPs

- *Identification scheme*. How should Alice prove to Bob that she is who she claimed to be? For example, how to design a control access system to the CSE dept.?

  A direct solution is to have a box on the door and give authorized people a secret PIN number. However, a drawback is that the box remains outside all the time and if someone could examine the box, they would perhaps be able to view its memory and extract the secrets keys of all people.

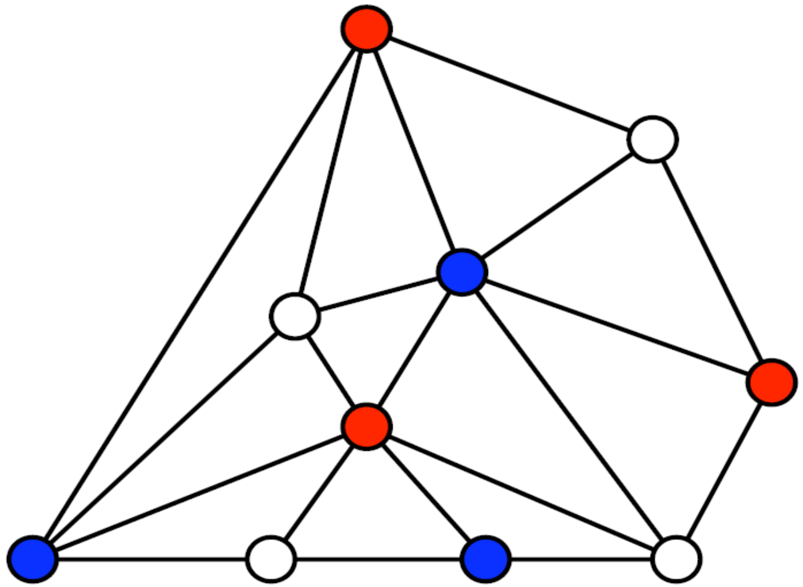
  Ideas using ZKPs:
  - Let the box contain an *instance* of a hard problem.
  - Give the authorized people the *solution* to the instance.
  - The authorized people will *prove* to the box that they know the solution in zero knowledge.

An Example



- Alice knows how to 3-color a graph: no two adjacent vertices have the same color; this is an NPC problem.

An Example



- Alice knows how to 3-color a graph: no two adjacent vertices have the same color; this is an NPC problem.
  - can impress your friends
  - useful for identification

An Example

- How can Alice convince Bob that she can 3-color the graph without
  - letting him steal her work?
  - letting him impersonate her?

An Example

- How can Alice convince Bob that she can 3-color the graph without
  - letting him steal her work?
  - letting him impersonate her?

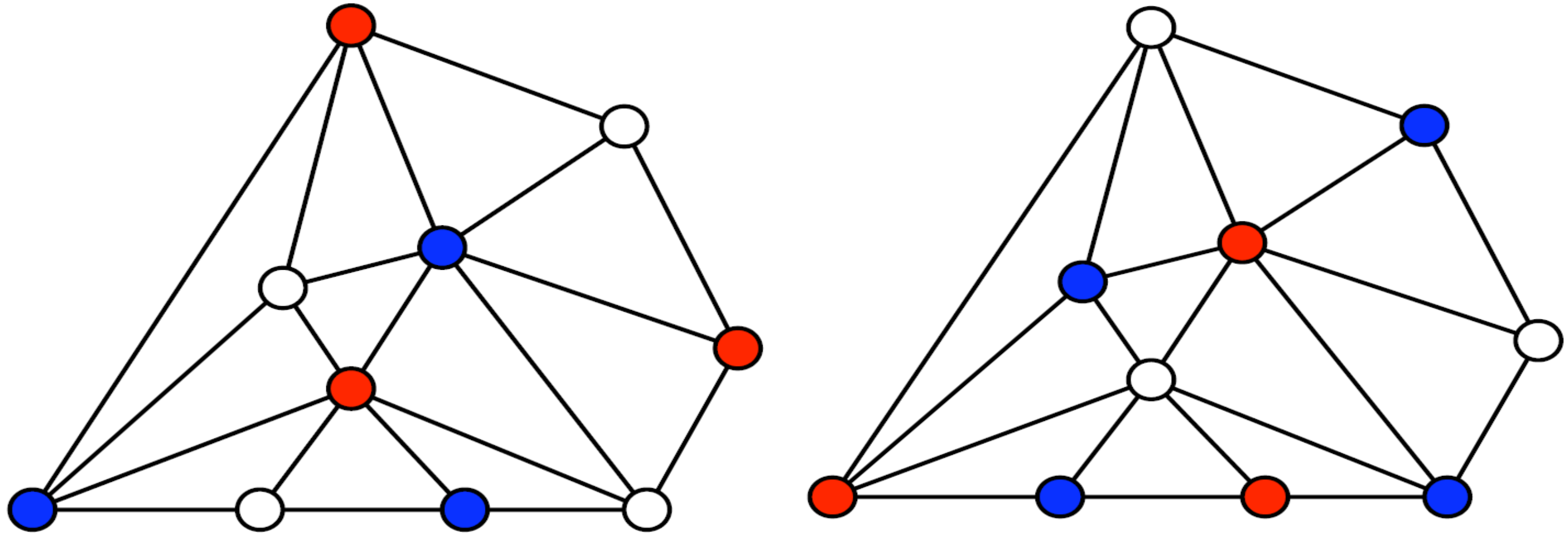
  - Bob is convinced that Alice can do this.
  - Bob has no idea how to do it himself.

An Example

- How can Alice convince Bob that she can 3-color the graph without
  - letting him steal her work?
  - letting him impersonate her?

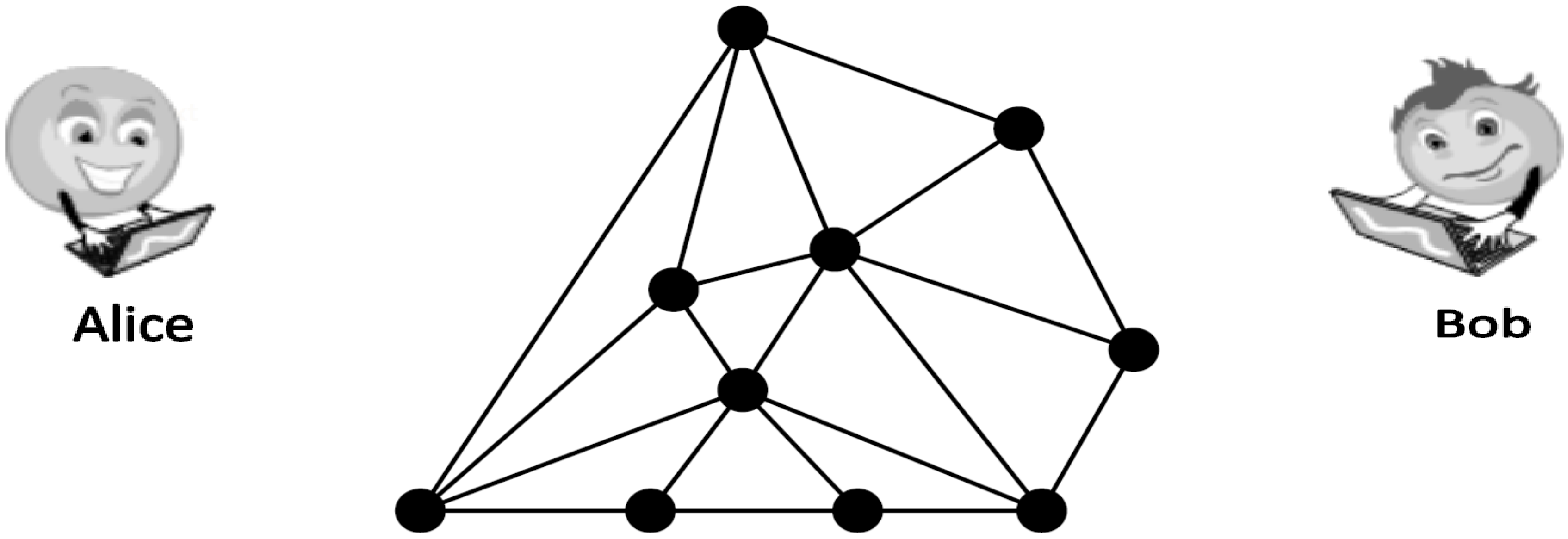
  - Bob is convinced that Alice can do this.
  - Bob has no idea how to do it himself.

Alice may permute the vertex colors.



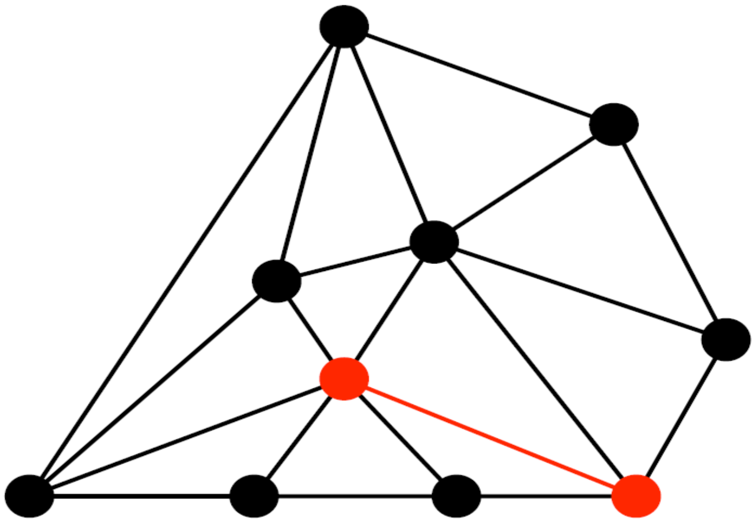
An Example

- Alice then encrypts all vertex colors (one key per vertex), and sends the graph to Bob.



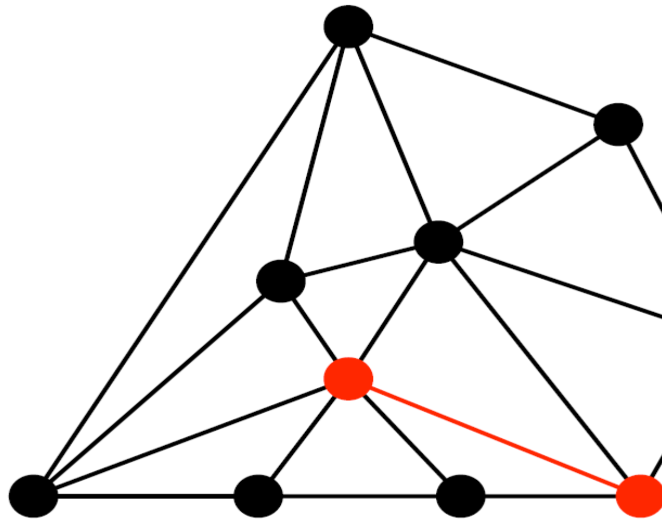
An Example

- Bob picks an edge at random.

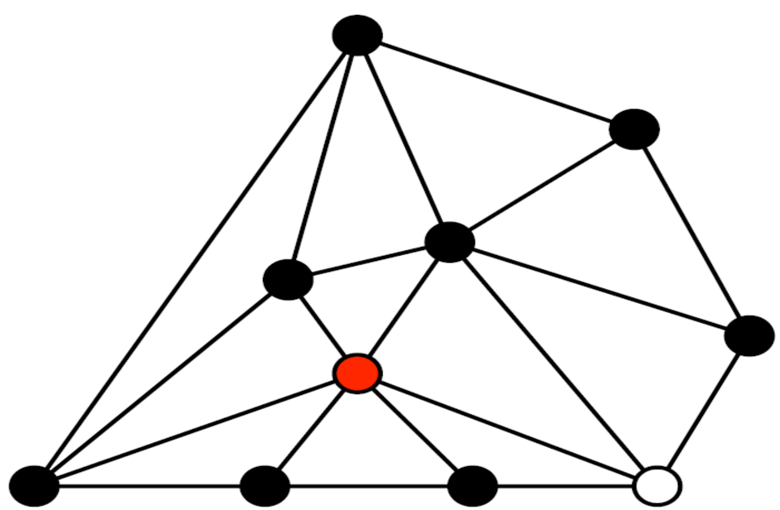


An Example

- Bob picks an edge at random.



Alice reveals colors of those two keys.



An Example

- Repeat as much as needed:
  - Alice permutes graph coloring
  - Alice encrypts all vertices with distinct keys
  - Alice sends permuted encrypted colors to Bob
  - Bob picks an edge
  - Alice sends keys for two vertices
  - Bob checks whether these two colors are distinct

An Example

- Repeat as much as needed:
  - Alice permutes graph coloring
  - Alice encrypts all vertices with distinct keys
  - Alice sends permuted encrypted colors to Bob
  - Bob picks an edge
  - Alice sends keys for two vertices
  - Bob checks whether these two colors are distinct

If Alice is lying, with probability $\frac{1}{|E|}$ she will be caught.
If she is telling the truth, she will never be caught.

An Example

- Repeat as much as needed:
  - Alice permutes graph coloring
  - Alice encrypts all vertices with distinct keys
  - Alice sends permuted encrypted colors to Bob
  - Bob picks an edge
  - Alice sends keys for two vertices
  - Bob checks whether these two colors are distinct

If Alice is lying, with probability $\frac{1}{|E|}$ she will be caught.
If she is telling the truth, she will never be caught.

After k repetitions, the probability she fools Bob is $(1 - \frac{1}{|E|})^k$.

An Example

- What does Bob see?
  - randomly-generated keys
  - randomly-generated colors

An Example

- What does Bob see?
  - randomly-generated keys
  - randomly-generated colors

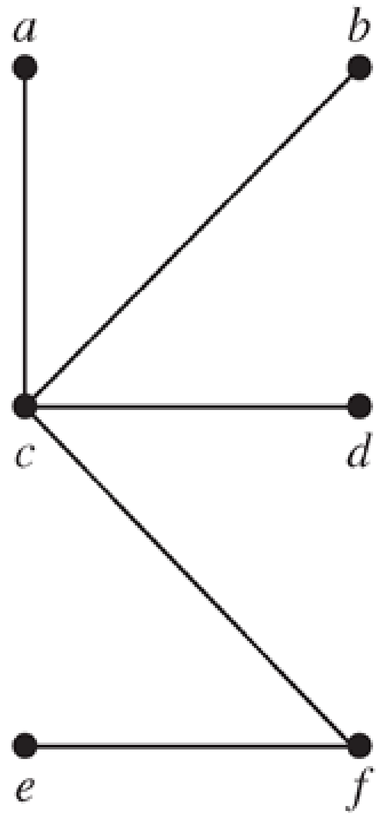
  Because Bob could have generated those keys and colors by himself, he learns nothing from the graph coloring.

Trees

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

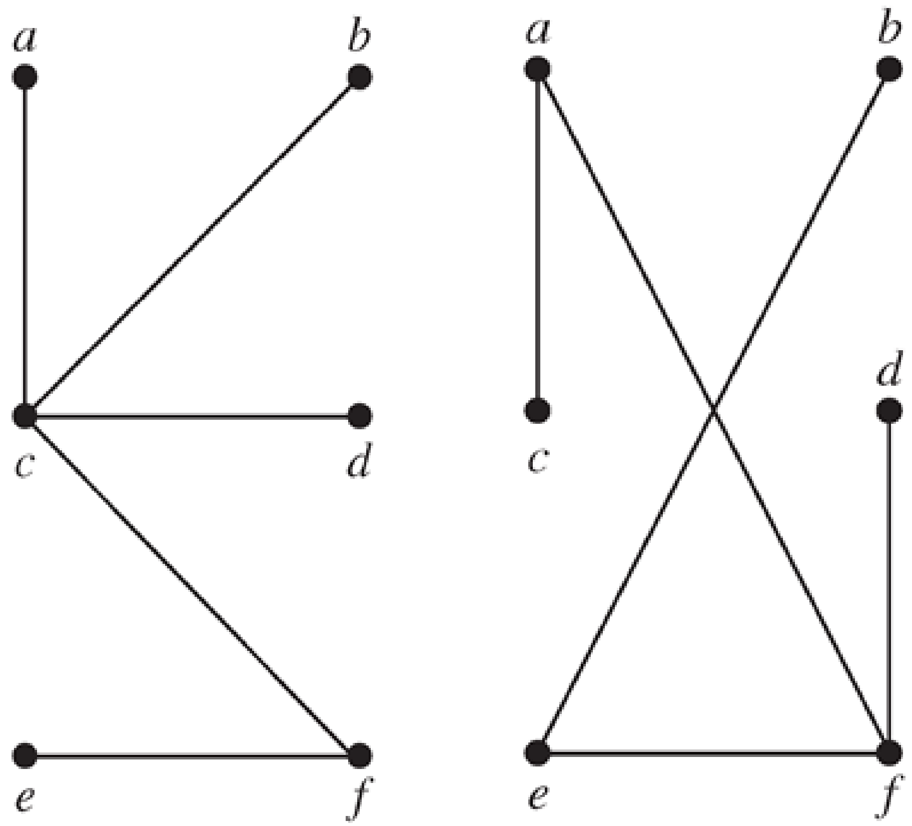
Trees

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

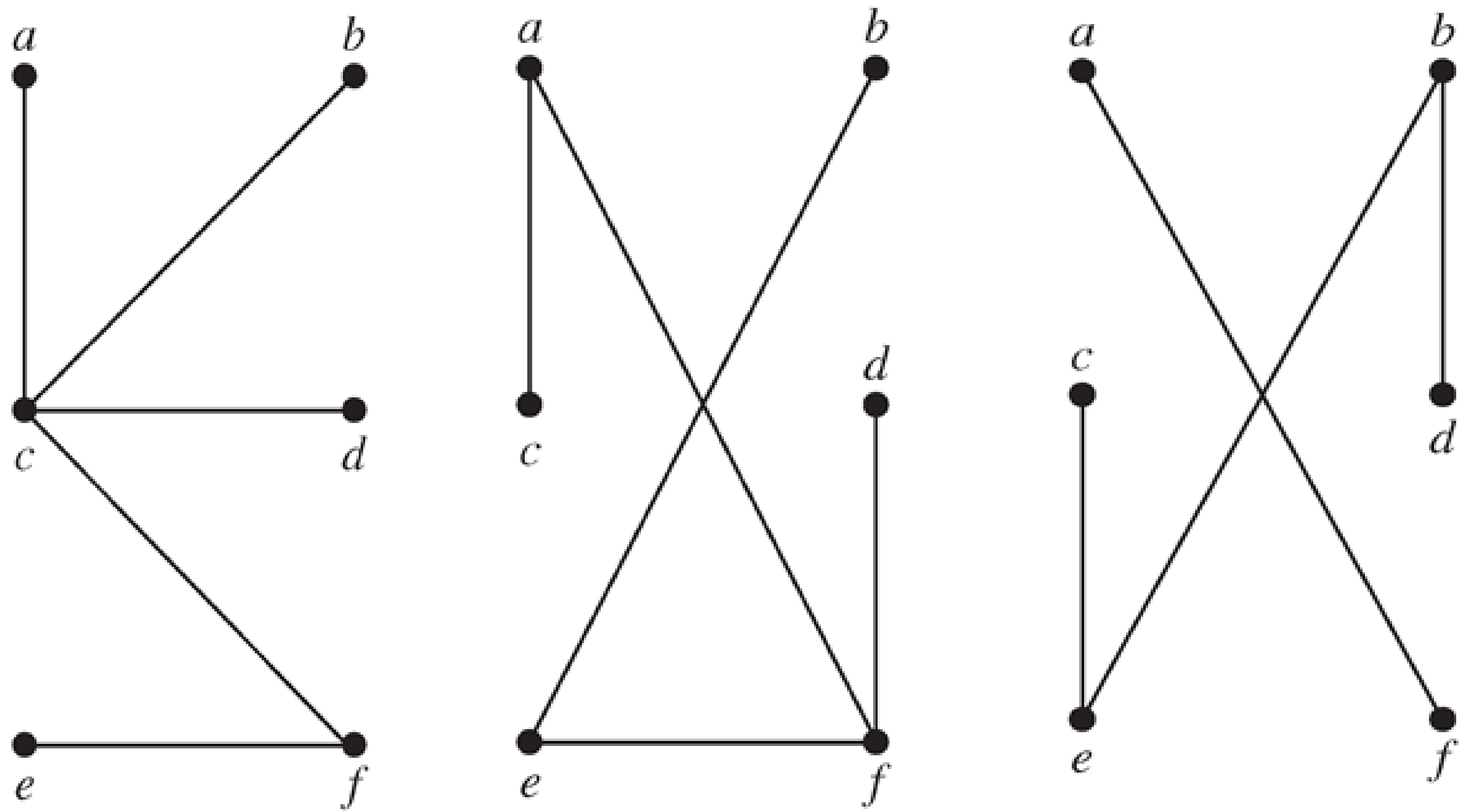


Trees

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

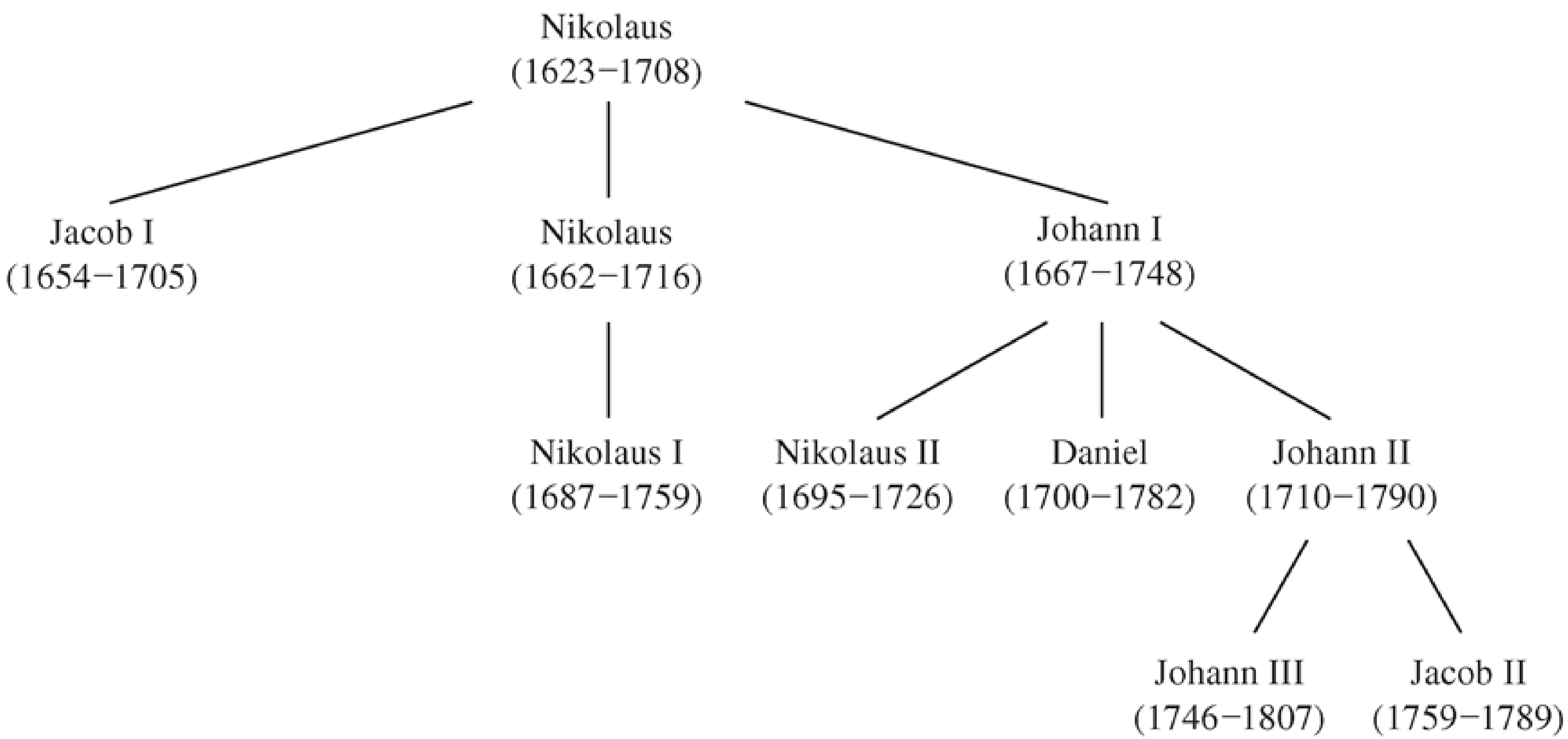


- **Definition** A *tree* is a connected undirected graph with no simple circuits.



Trees

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



Trees

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

Trees

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

Proof

Trees

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

  Proof

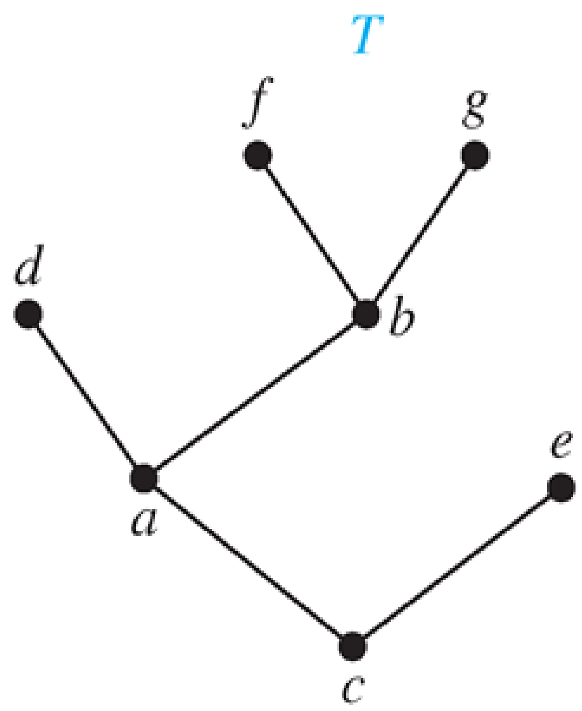
  Two properties of tree: connected, no circuit

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.

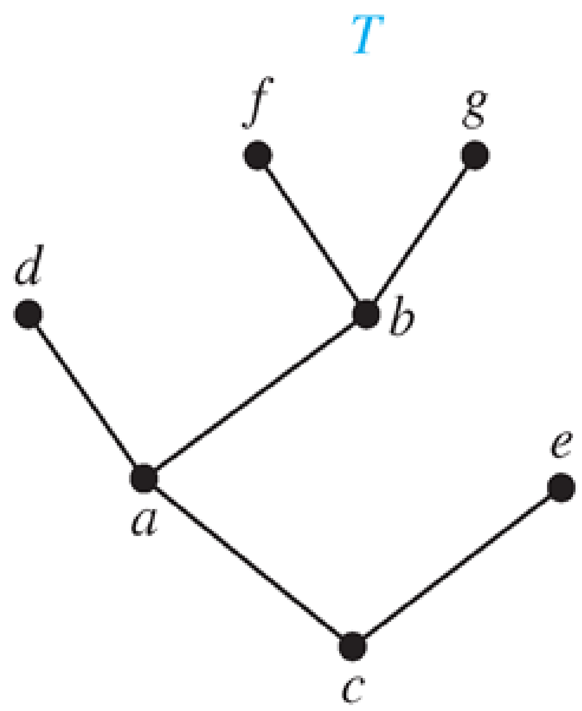
Rooted Trees

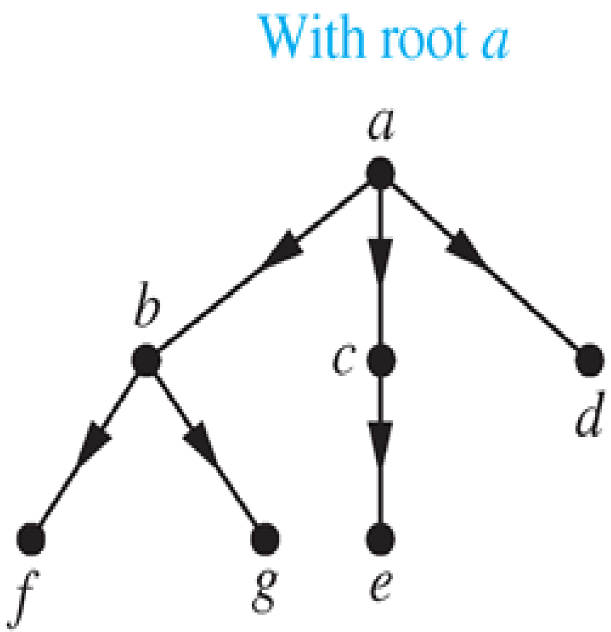
- **Definition** A *rooted tree* is a tree in which one vertex has been designated as the root and every edge is directed away from the root.



Rooted Trees

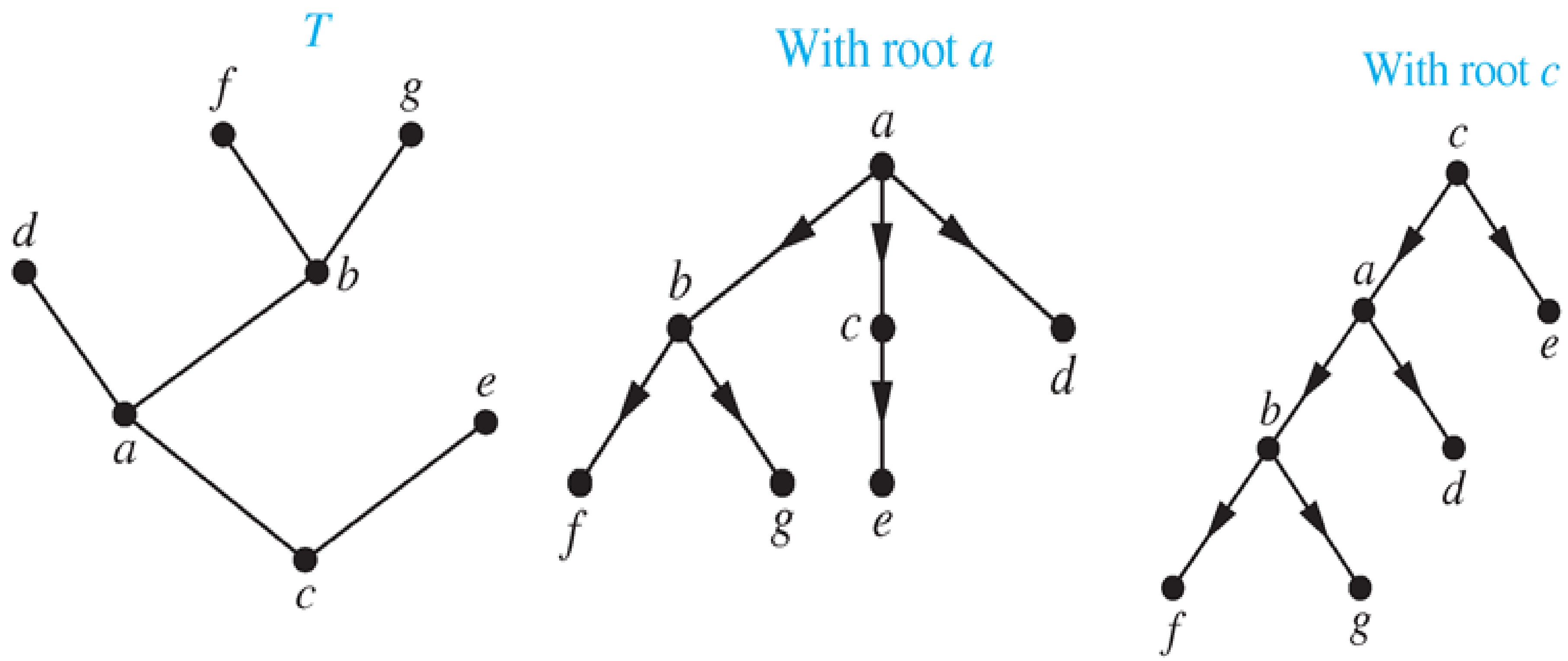
- **Definition** A *rooted tree* is a tree in which one vertex has been designated as the root and every edge is directed away from the root.





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

- *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

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*.

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*.

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*.

  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.

The Number of Leaves

- **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 $h = \lceil \log_m \ell \rceil$.

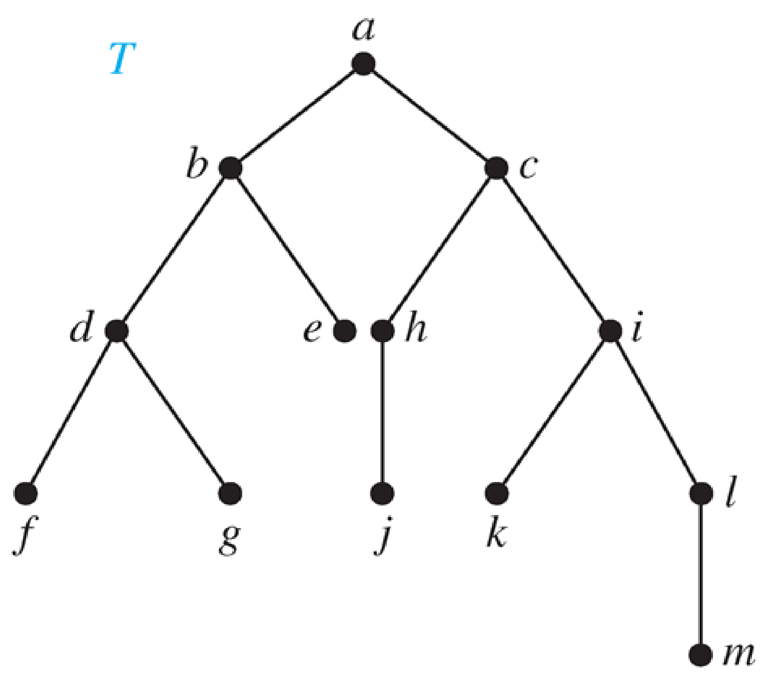
  Proof

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.

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.



Tree Traversal

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

Tree Traversal

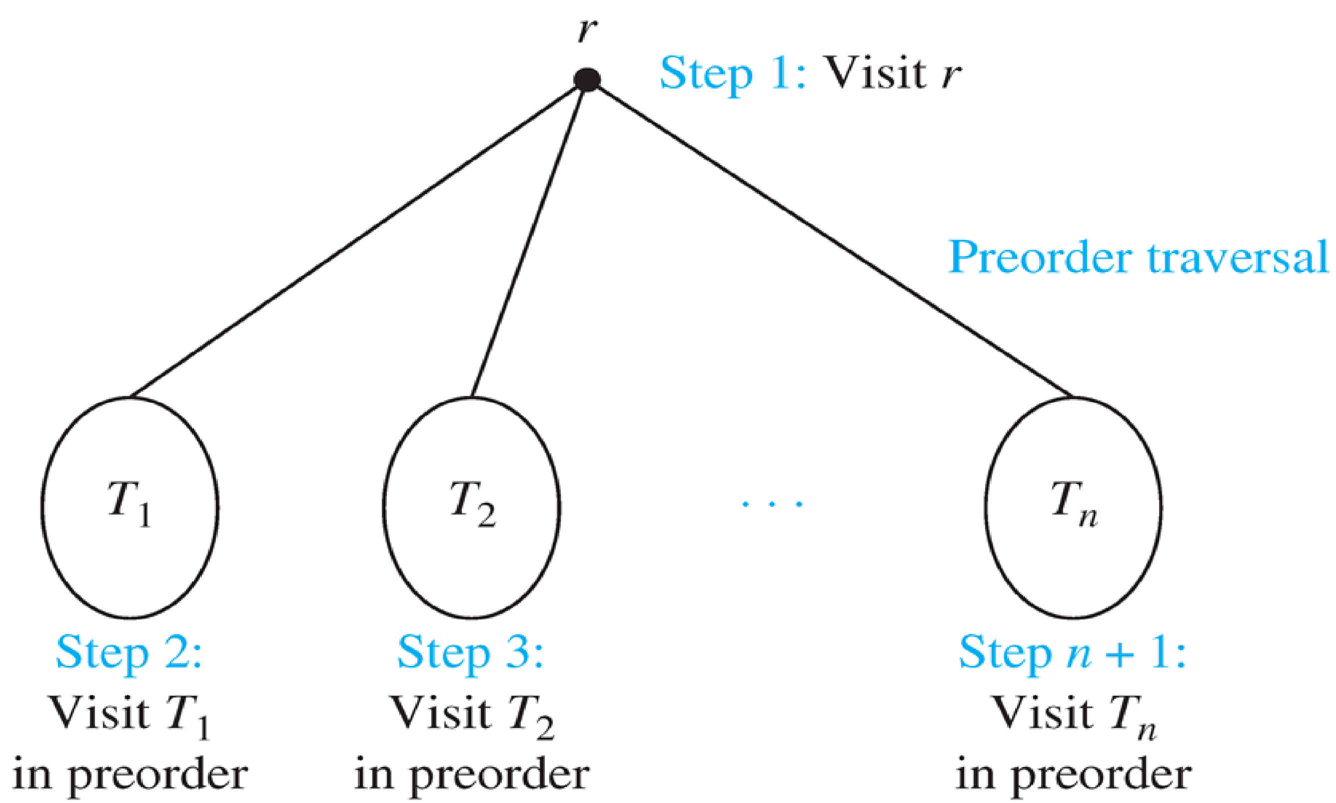
- The procedures for systematically visiting every vertex of an ordered tree are called *traversals*.

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

Preorder Traversal

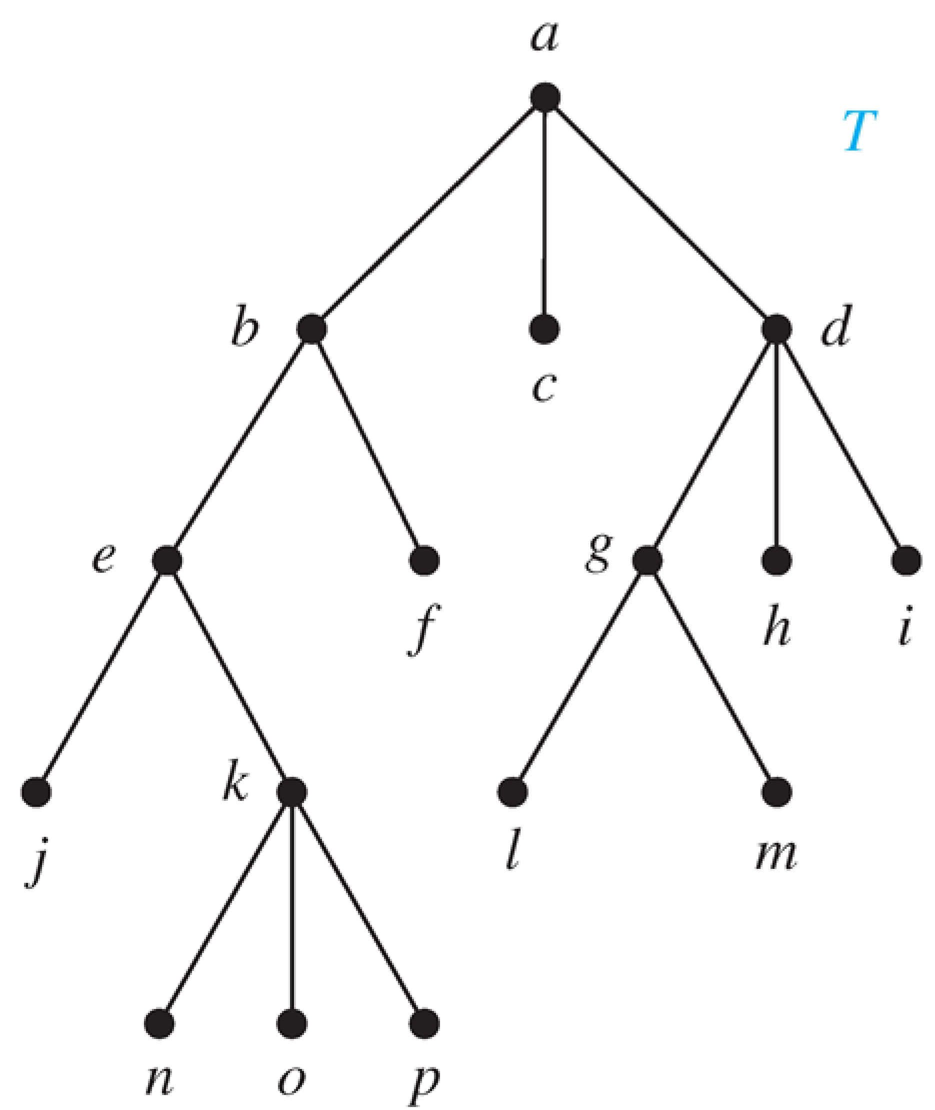
- **Definition** Let T be an ordered rooted tree with root r. If T consists only of r, then r is the *preorder traversal* of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees of r from left to right in T. The *preorder traversal* begins by visiting r, and continues by traversing T_1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

- **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

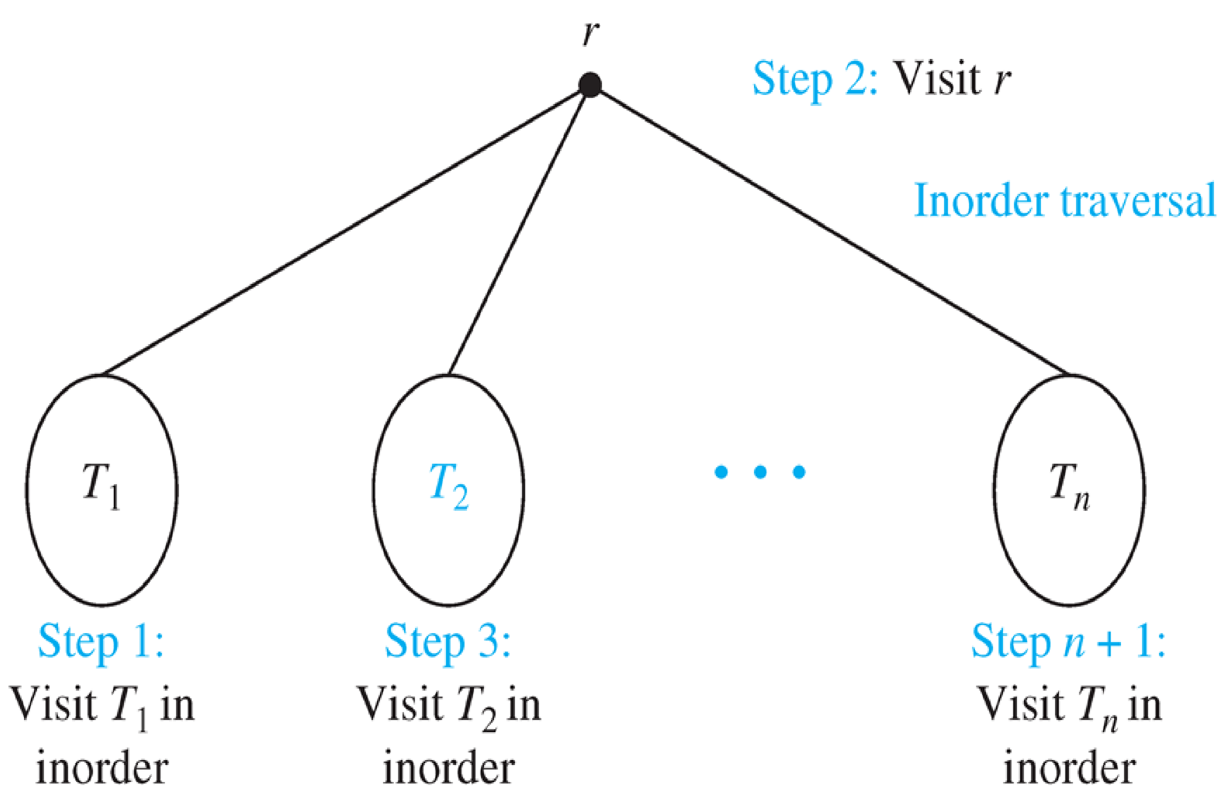
procedure *preorder* (T: ordered rooted tree)
r := root of T
list r
for each child c of r from left to right
    $T(c)$:= subtree with c as root
    preorder($T(c)$)

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.

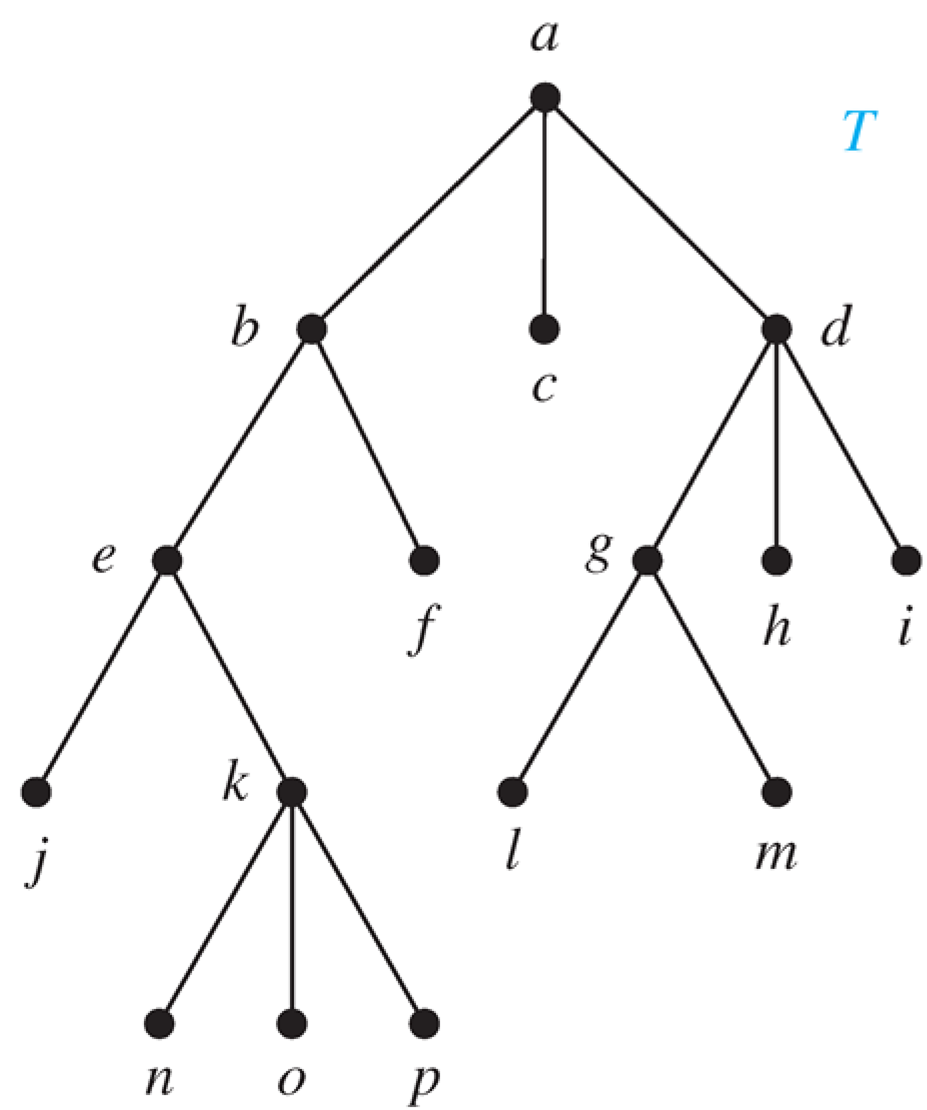
Inorder Traversal

- **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.



Inorder Traversal

- **Example**



Inorder Traversal

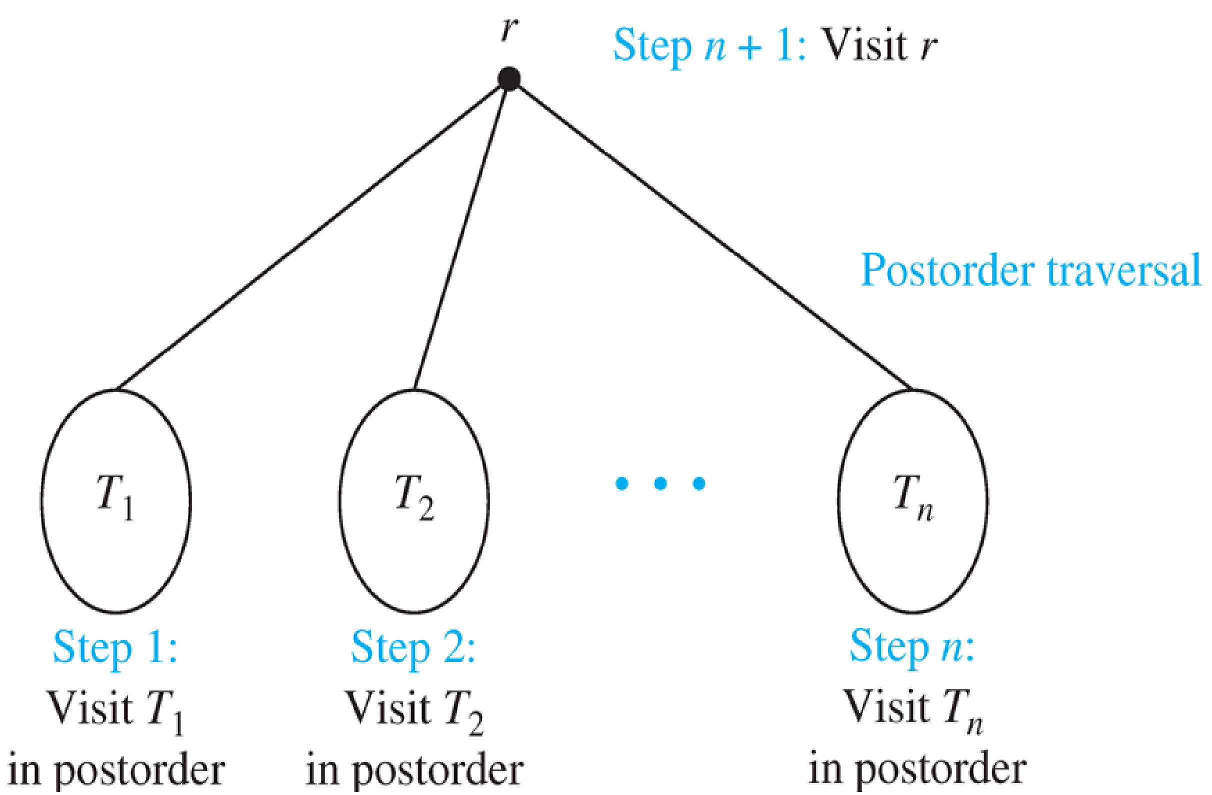
```
procedure inorder (T: ordered rooted tree)
r := root of T
if r is a leaf then list r
else
    l := first child of r from left to right
    T(l) := subtree with l as its root
    inorder(T(l))
    list(r)
    for each child c of r from left to right
        T(c) := subtree with c as root
        inorder(T(c))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

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.

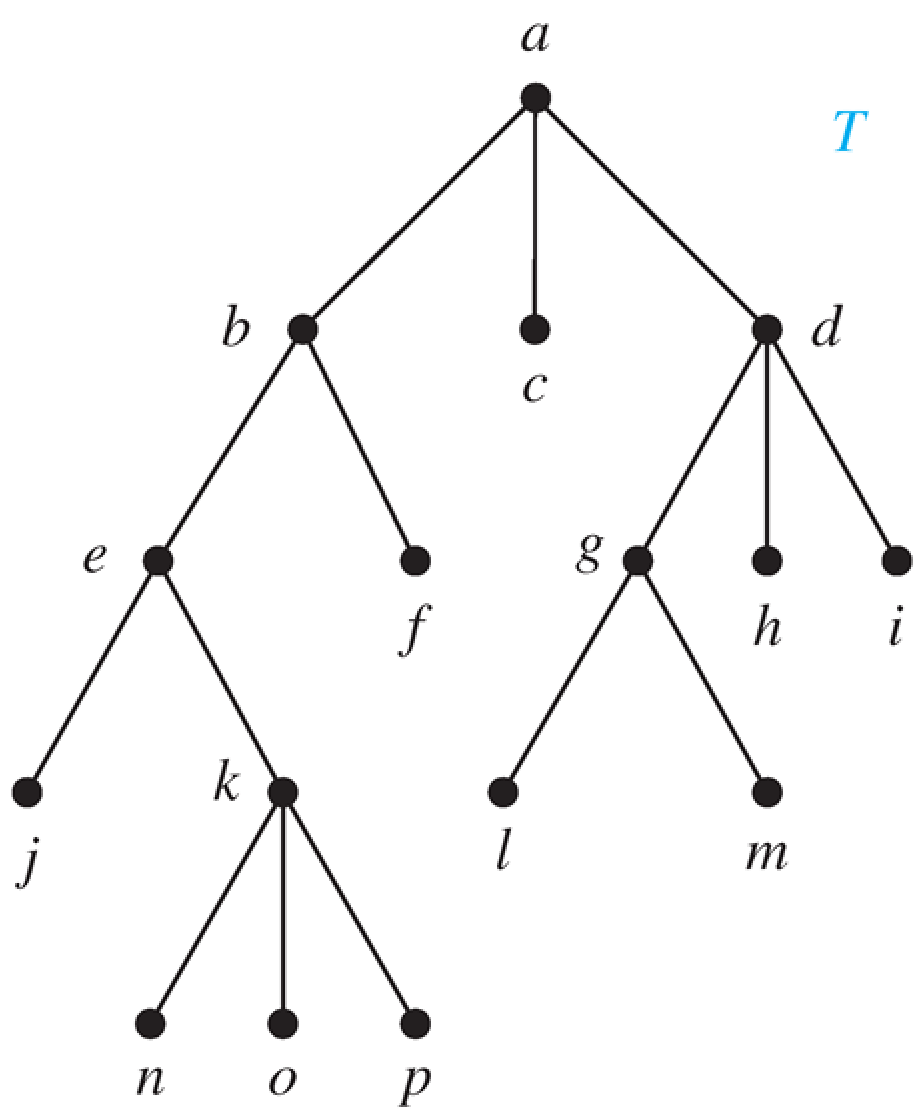
Postorder Traversal

- **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

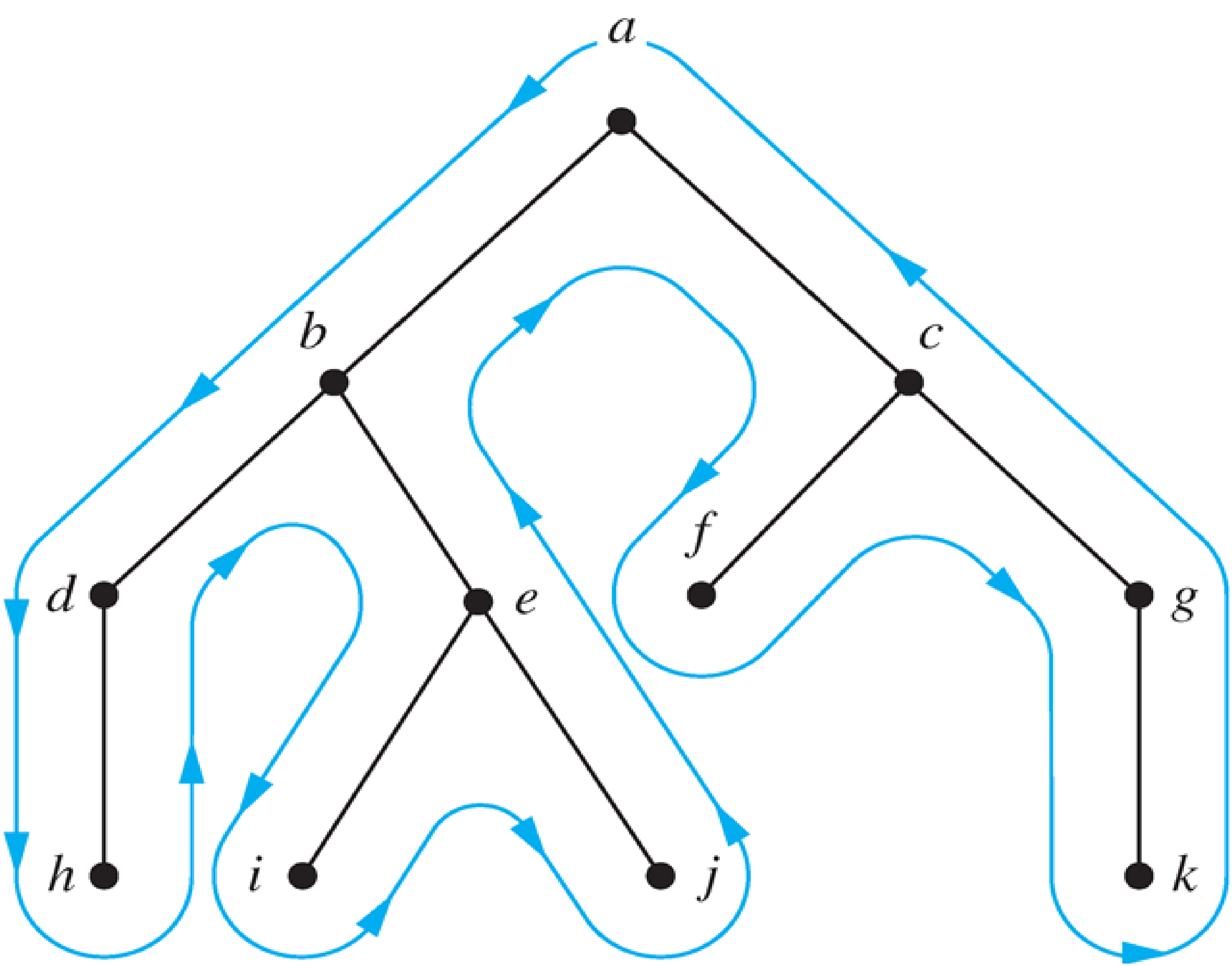
- **Example**



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

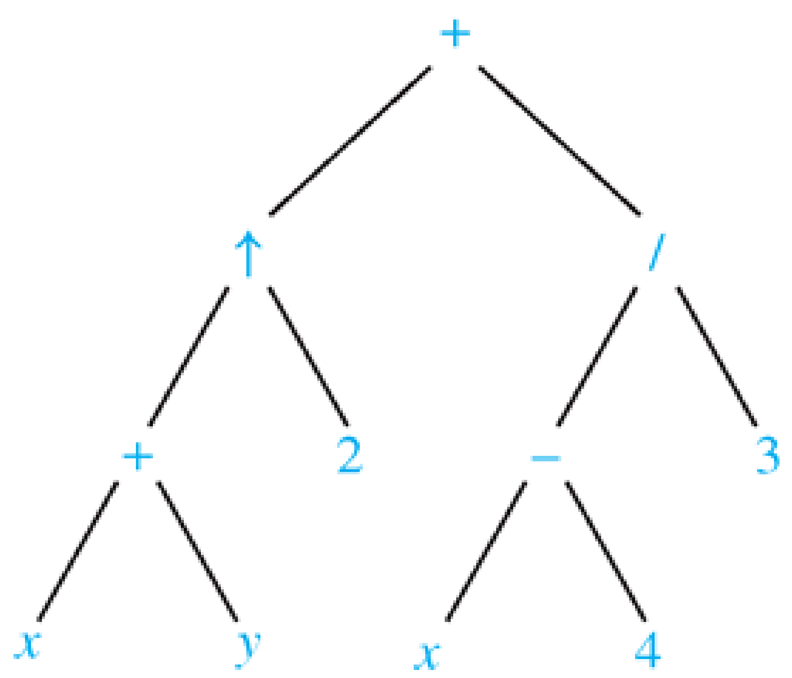
consider the expression $((x + y) \uparrow 2) + ((x - 4)/3)$

Expression Trees

- 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

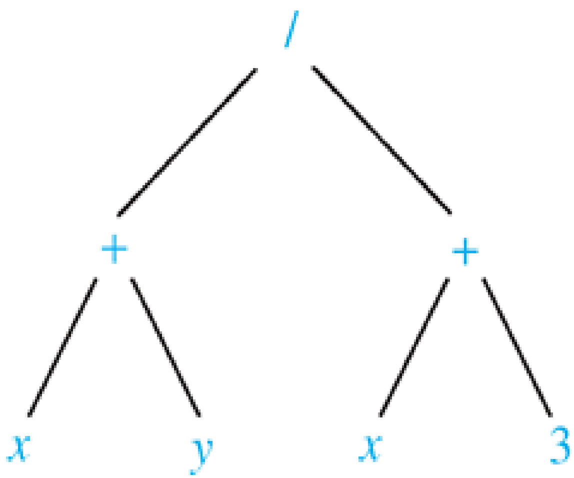
- An inorder traversal of the tree representing an expression produces the original expression when parentheses are included except for unary operation.

  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.

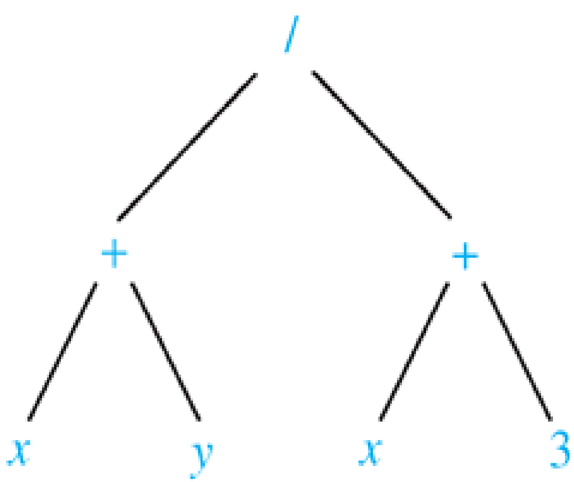
  Why parentheses are needed?

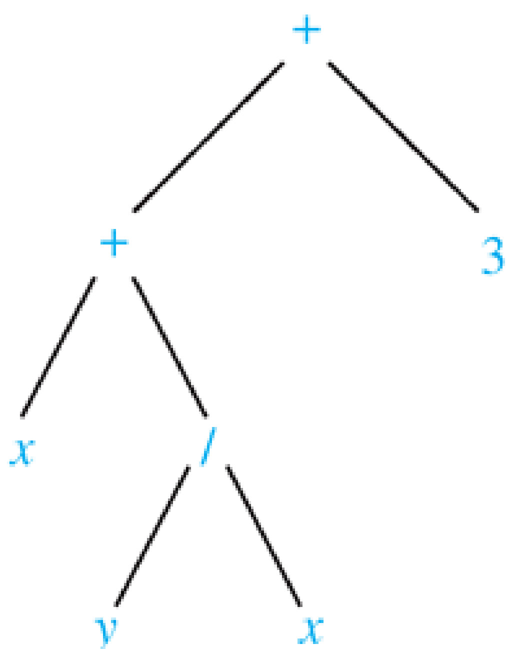


Infix Notation

- 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?

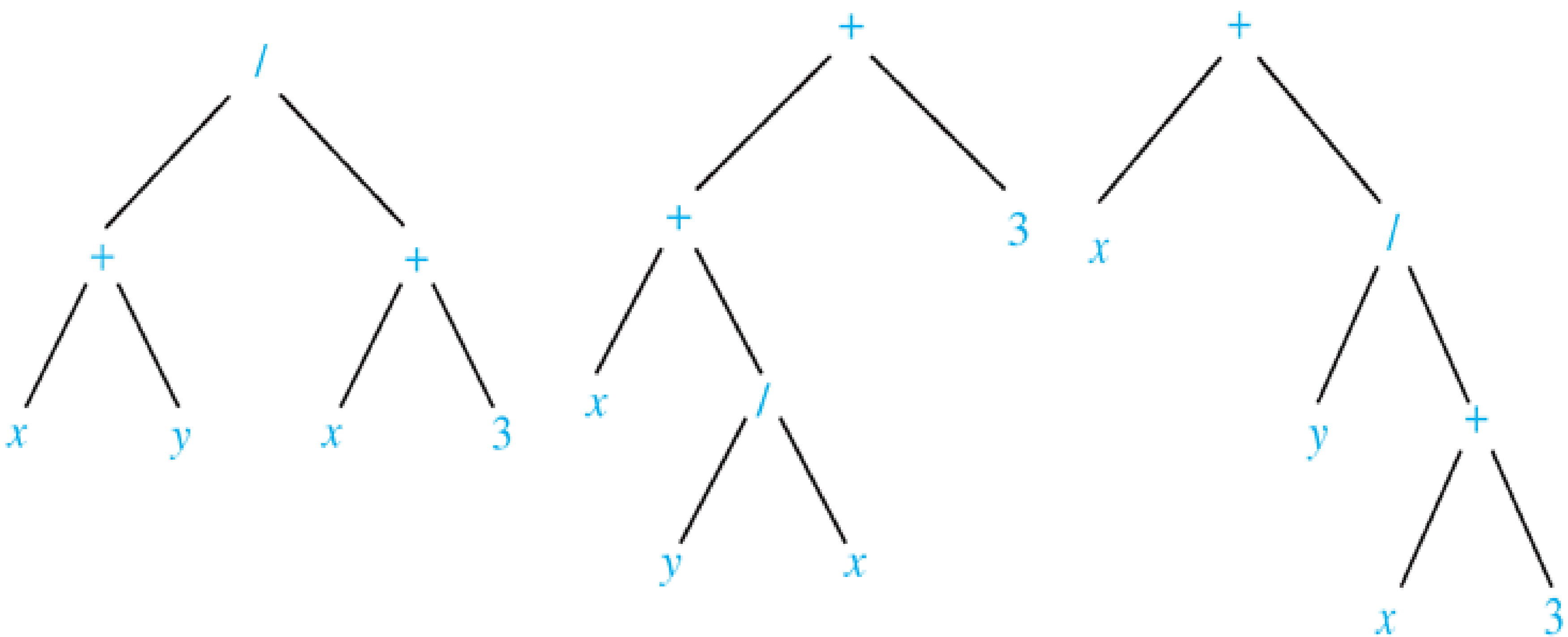




Infix Notation

- 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**

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

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

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

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

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

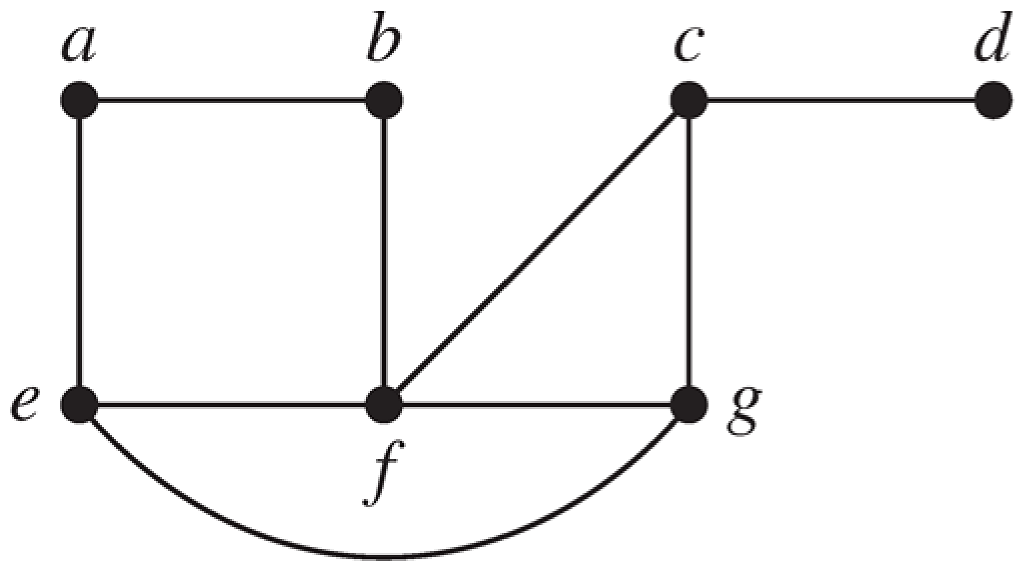
$$\underbrace{1\quad 3\quad +}_{1+3=4}$$

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.

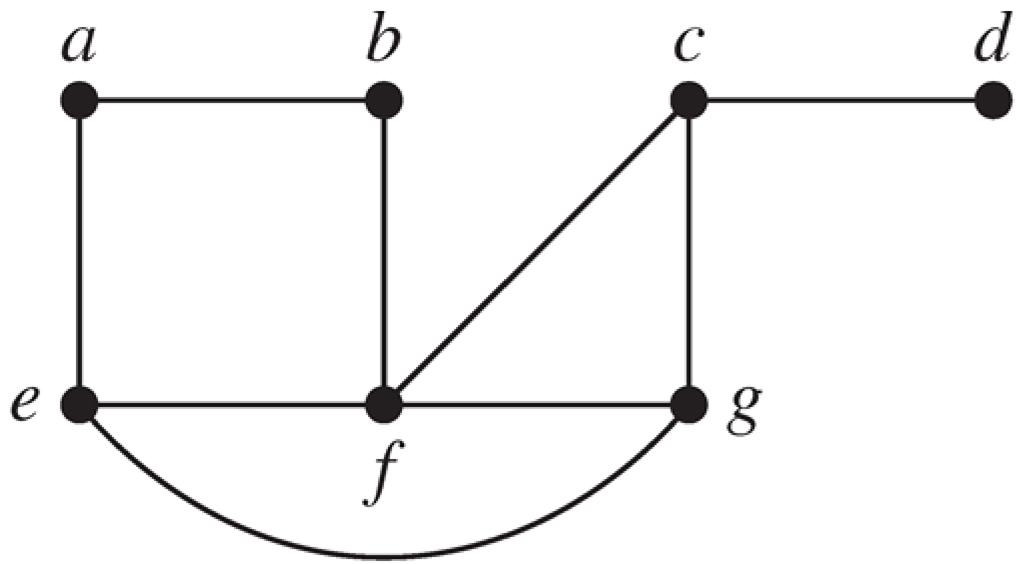
Spanning Trees

- **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.



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.



remove edges to avoid circuits

Spanning Trees

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

Spanning Trees

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

Proof

Spanning Trees

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

Proof

"only if" part

Spanning Trees

- **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.

Spanning Trees

- **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

Spanning Trees

- **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.

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.

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.

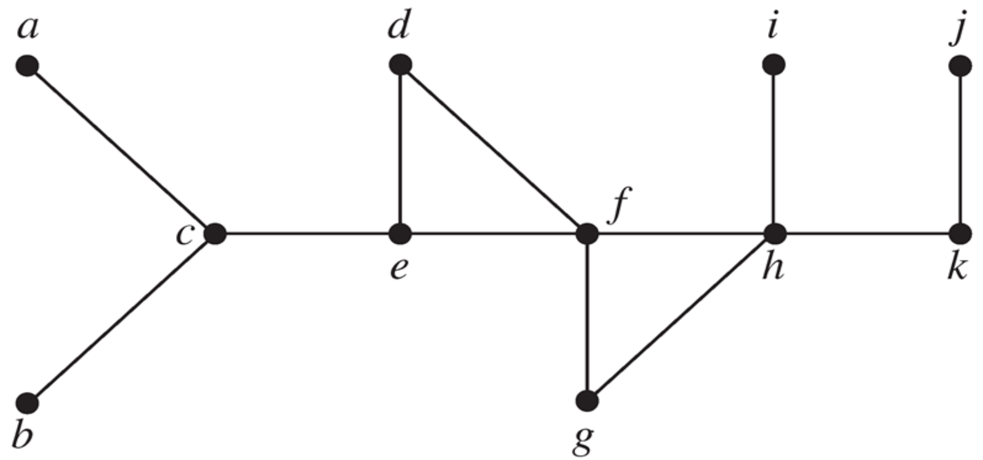
  ◇ First arbitrarily choose a vertex of the graph as the root.

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

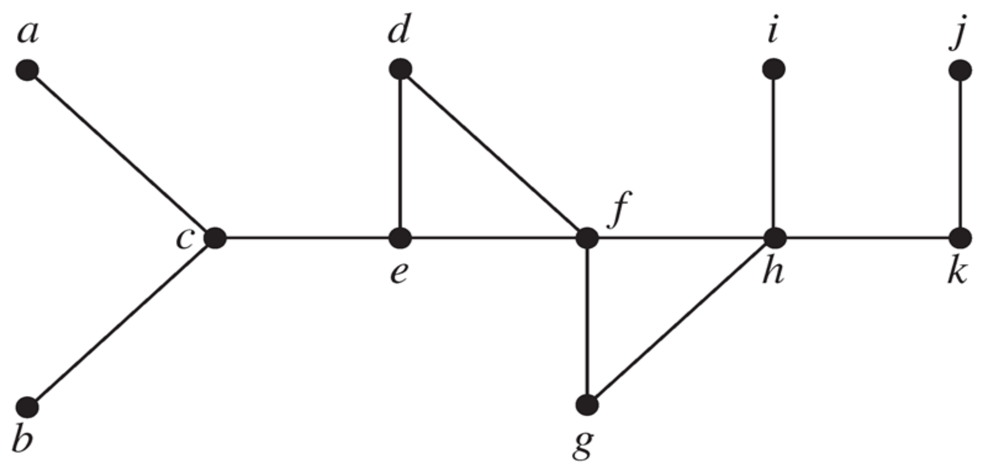
  ◇ If the path goes through all vertices of the graph, the tree is a spanning tree.

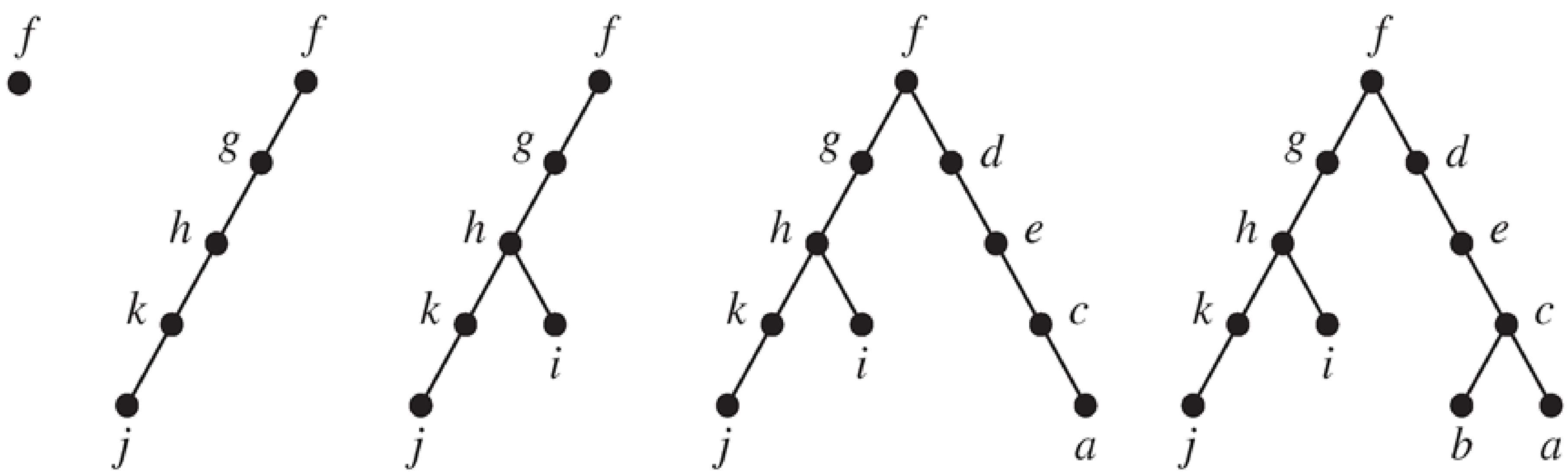
  ◇ Otherwise, move back to some vertex to repeat this procedure (*backtracking*)

- **Example**



- **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.

Breadth-First Search

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

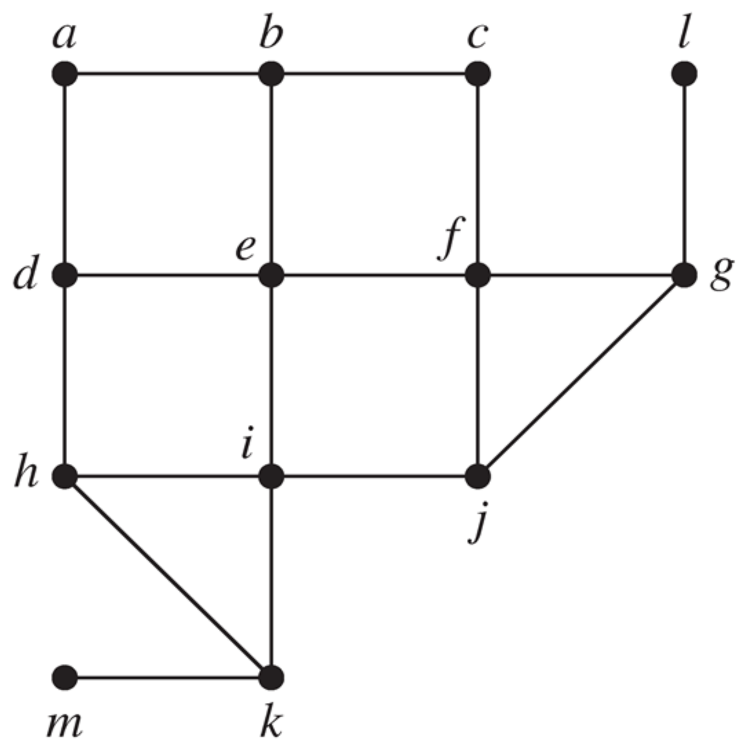
  ◇ First arbitrarily choose a vertex of the graph as the root.

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

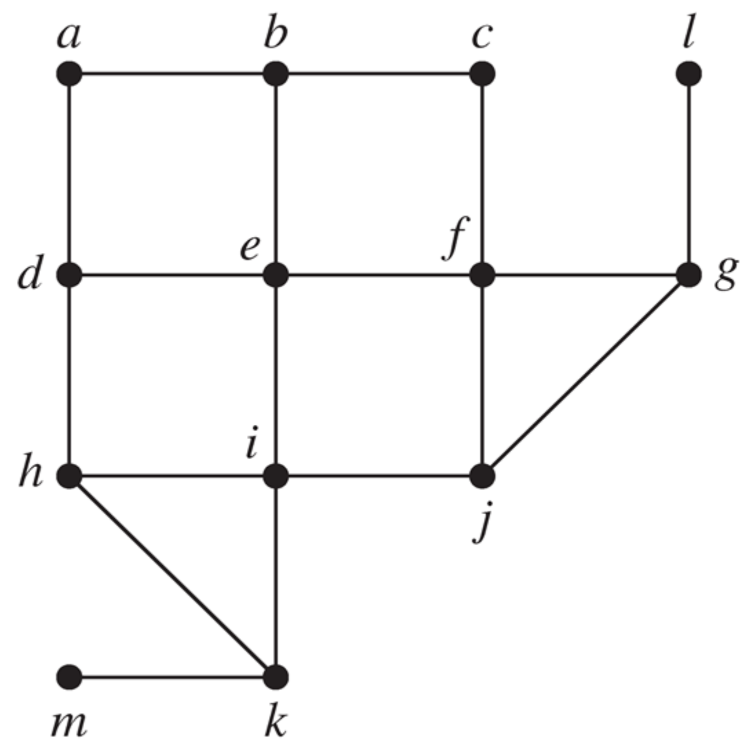
  ◇ For each vertex added at the previous level, add edge incident to this vertex, as long as it does not produce a simple circuit.

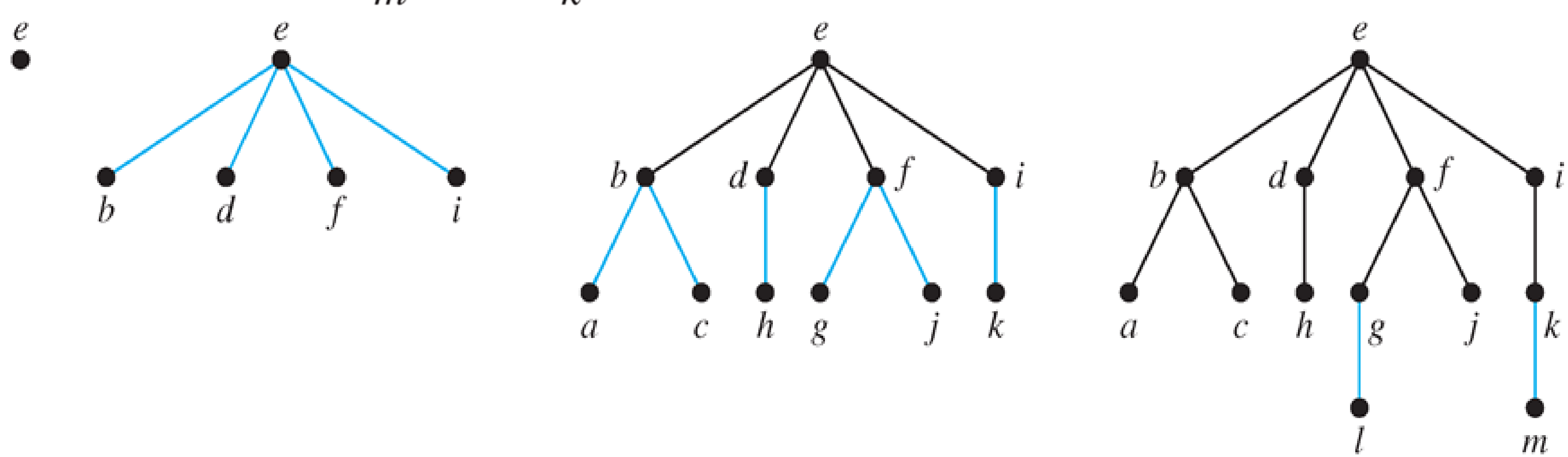
  ◇ Continue in this manner until all vertices have been added.

- **Example**



- **Example**





Breadth-First Search

```
procedure BFS(G: connected graph with vertices v_1, v_2, ..., 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, ..., 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, ...

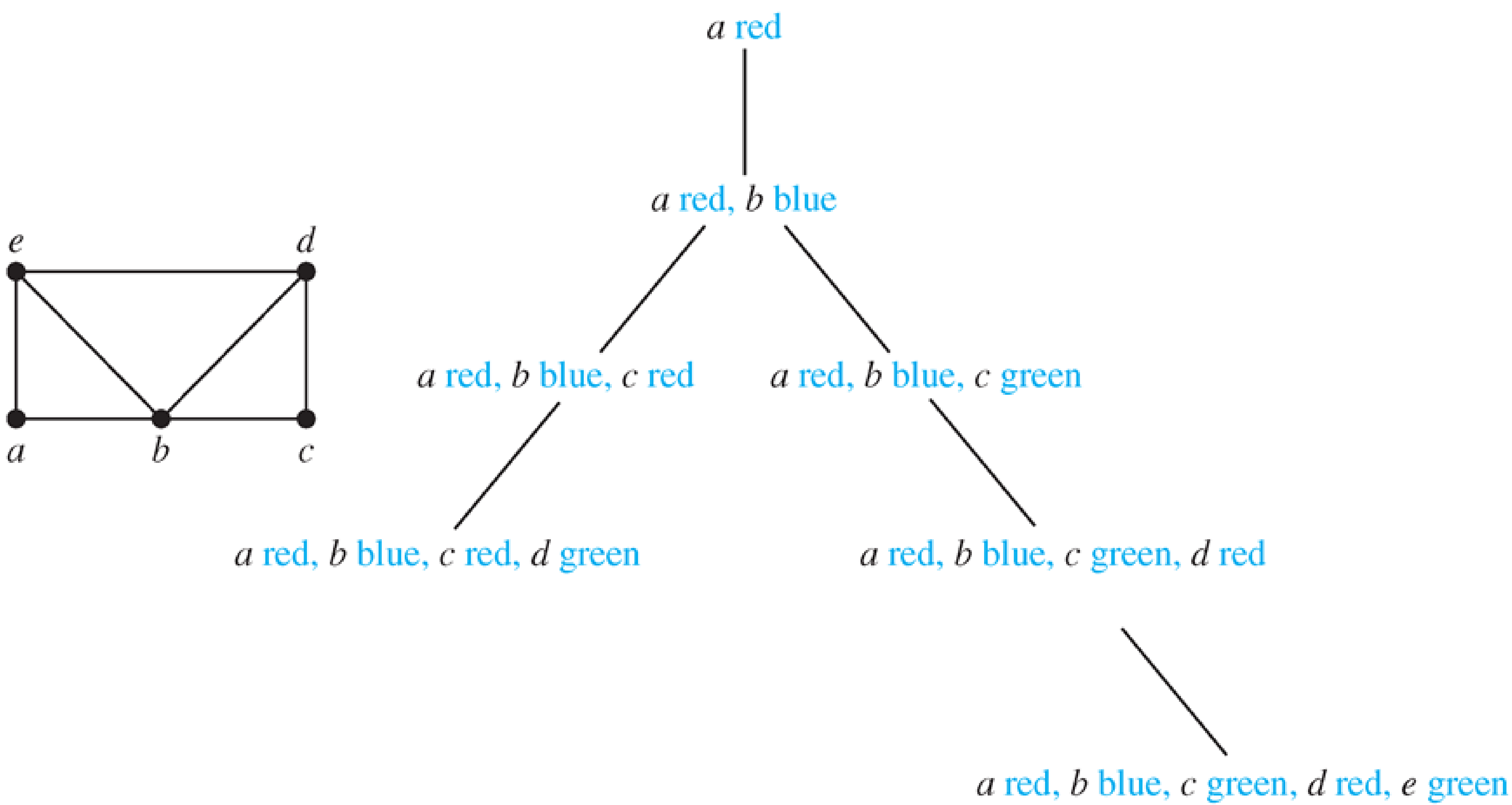
Applications of DFS, BFS

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

  find shortest paths, determine whether bipartite, ...

  graph coloring, sums of subsets, ...

Applications of DFS, BFS

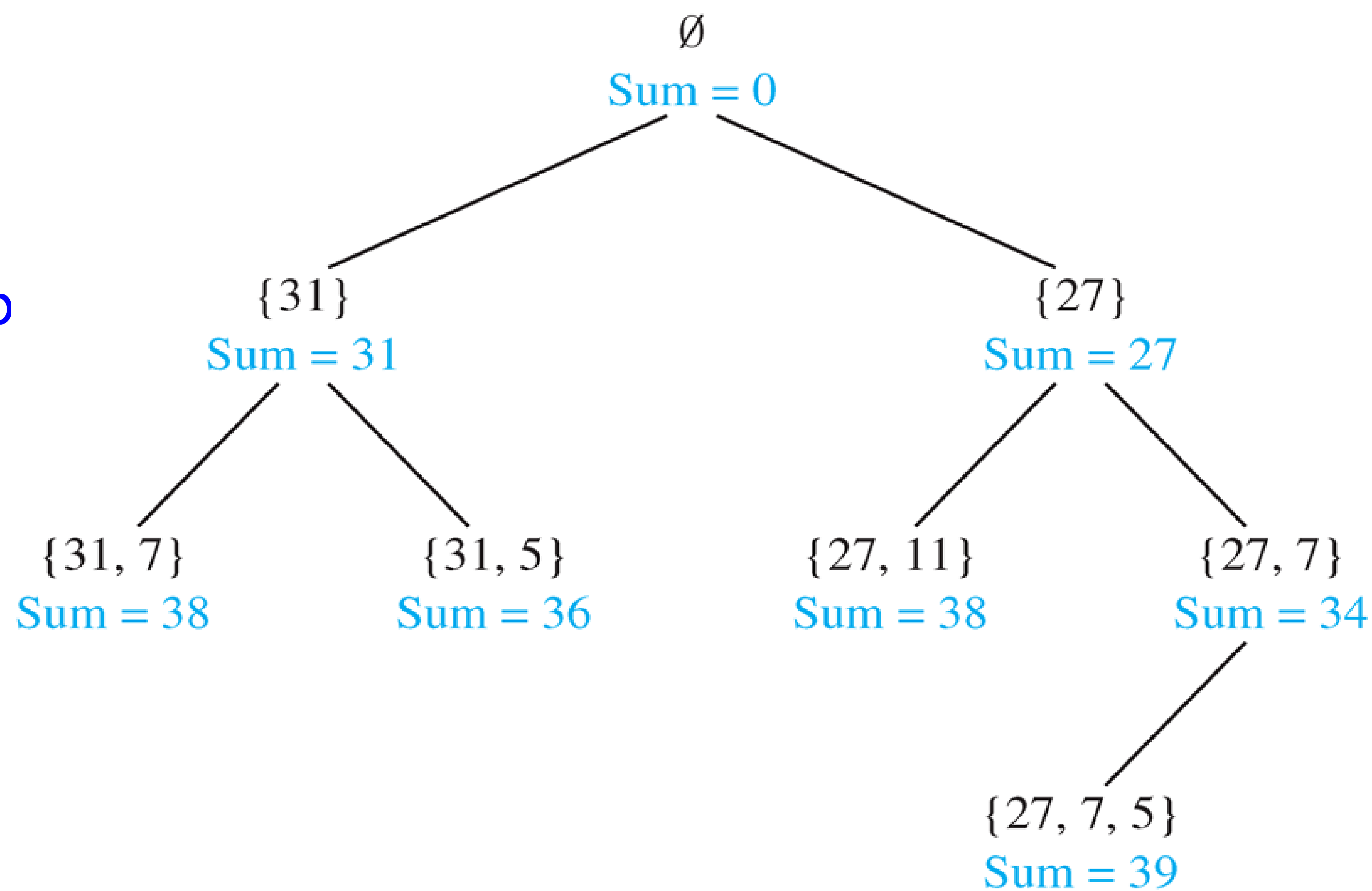
- 

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.

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.

  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

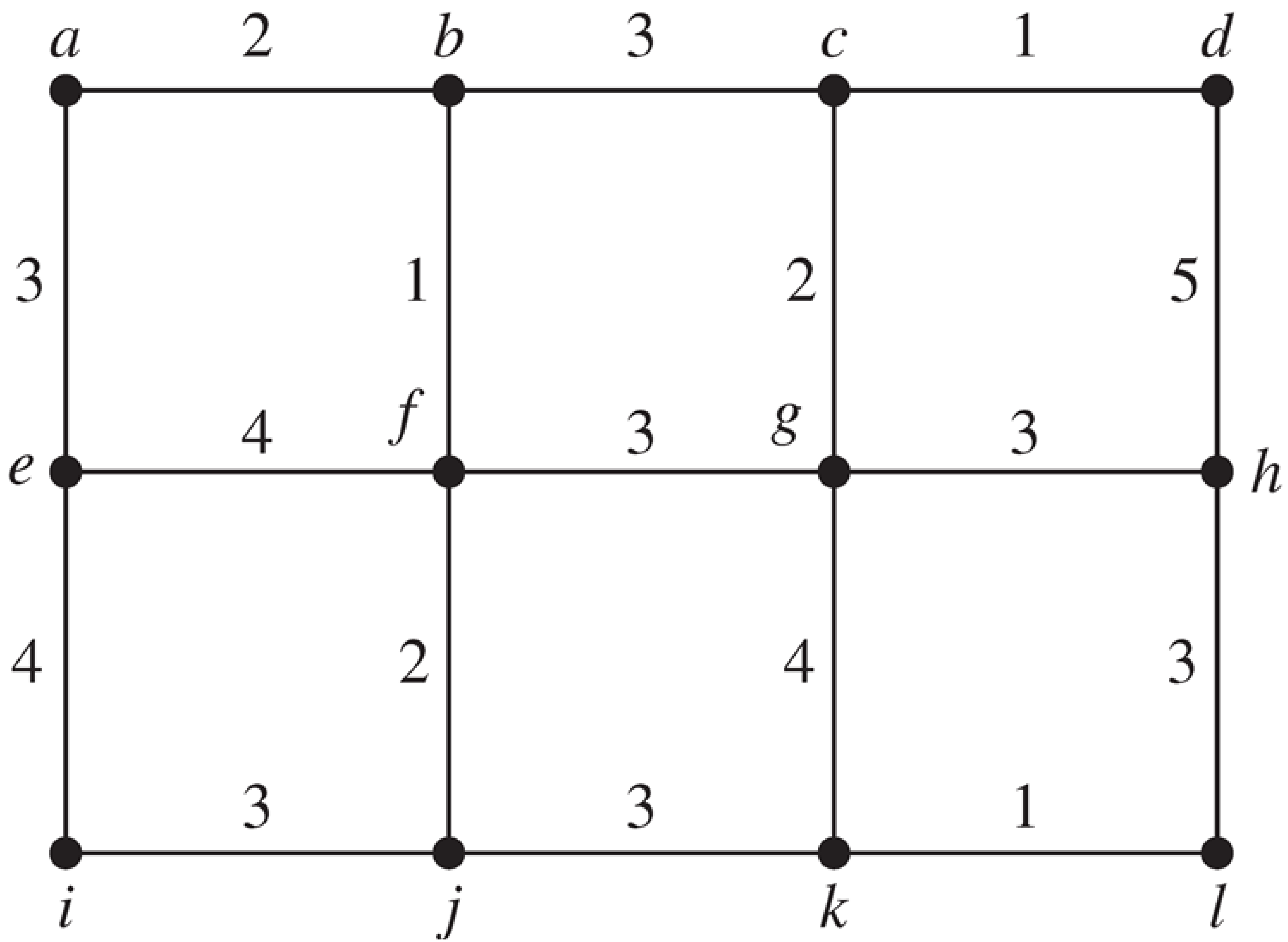
ALGORITHM 1 Prim's Algorithm.

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: $e \log v$

Prim's Algorithm

- **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

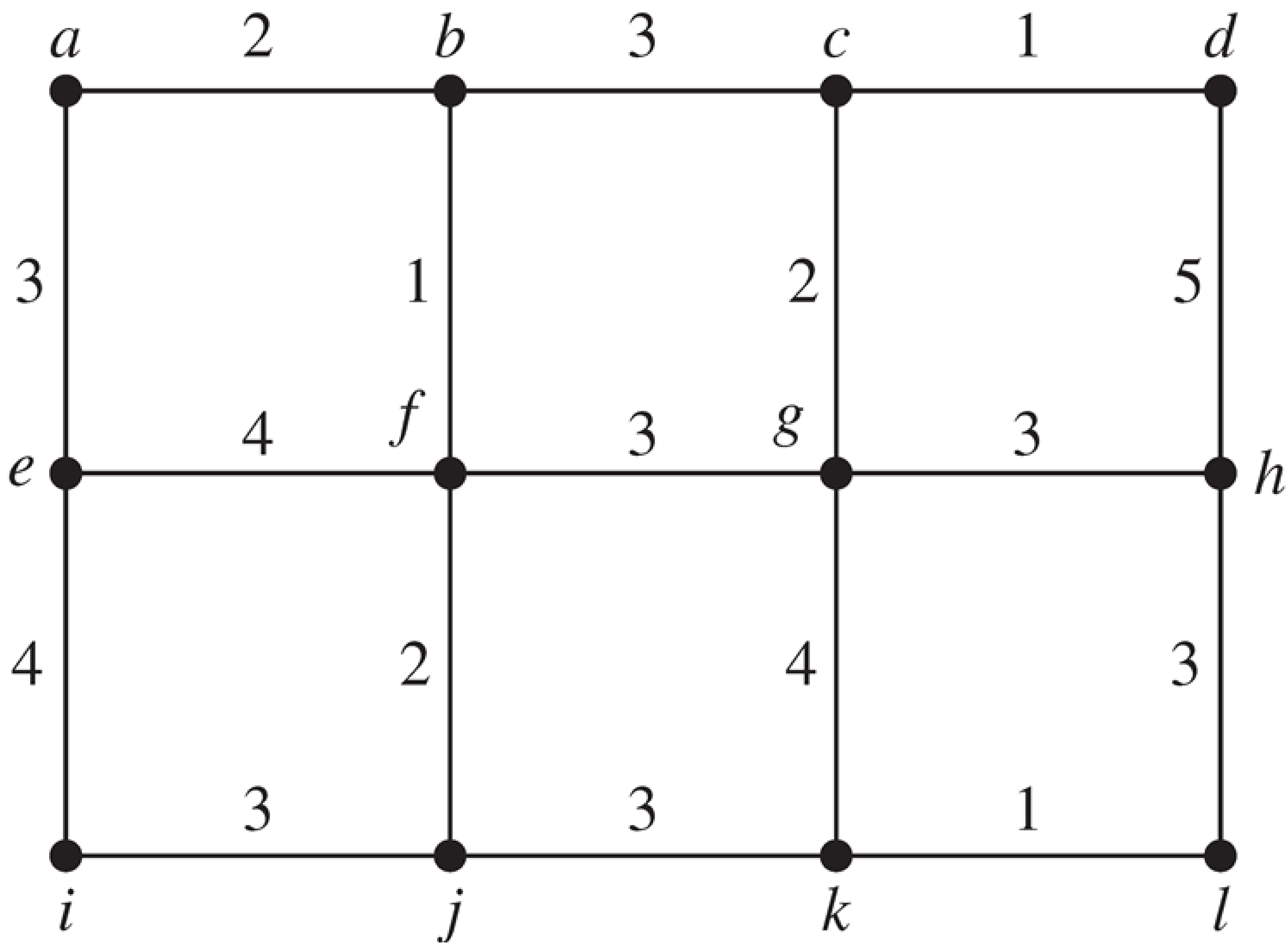
ALGORITHM 2 Kruskal's Algorithm.

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 ...