南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# DISCRETE MATHEMATICS FOR COMPUTER SCIENCE

Dr. QI WANG

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

# 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*.

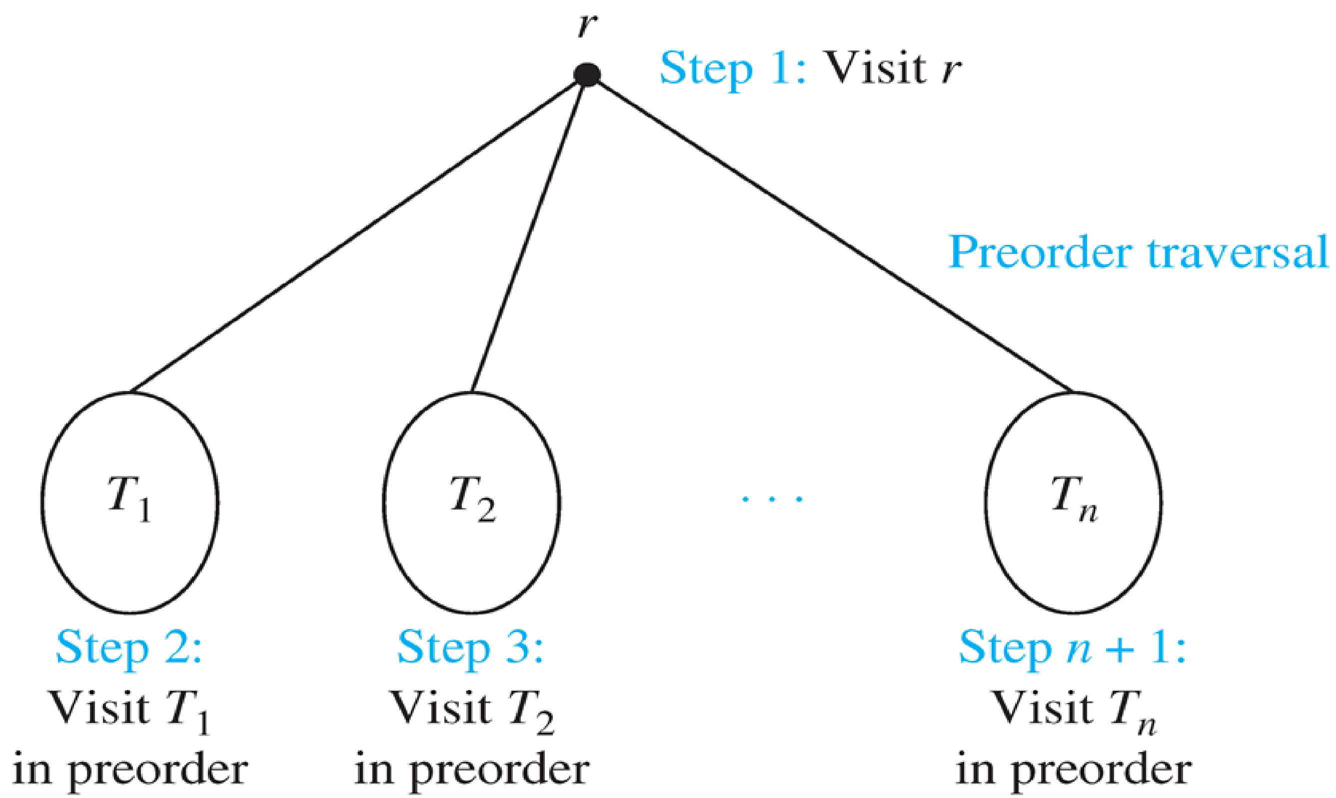
  The three most commonly used traversals are *preorder traversal*, *inorder traversal*, *postorder traversal*.

# 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.

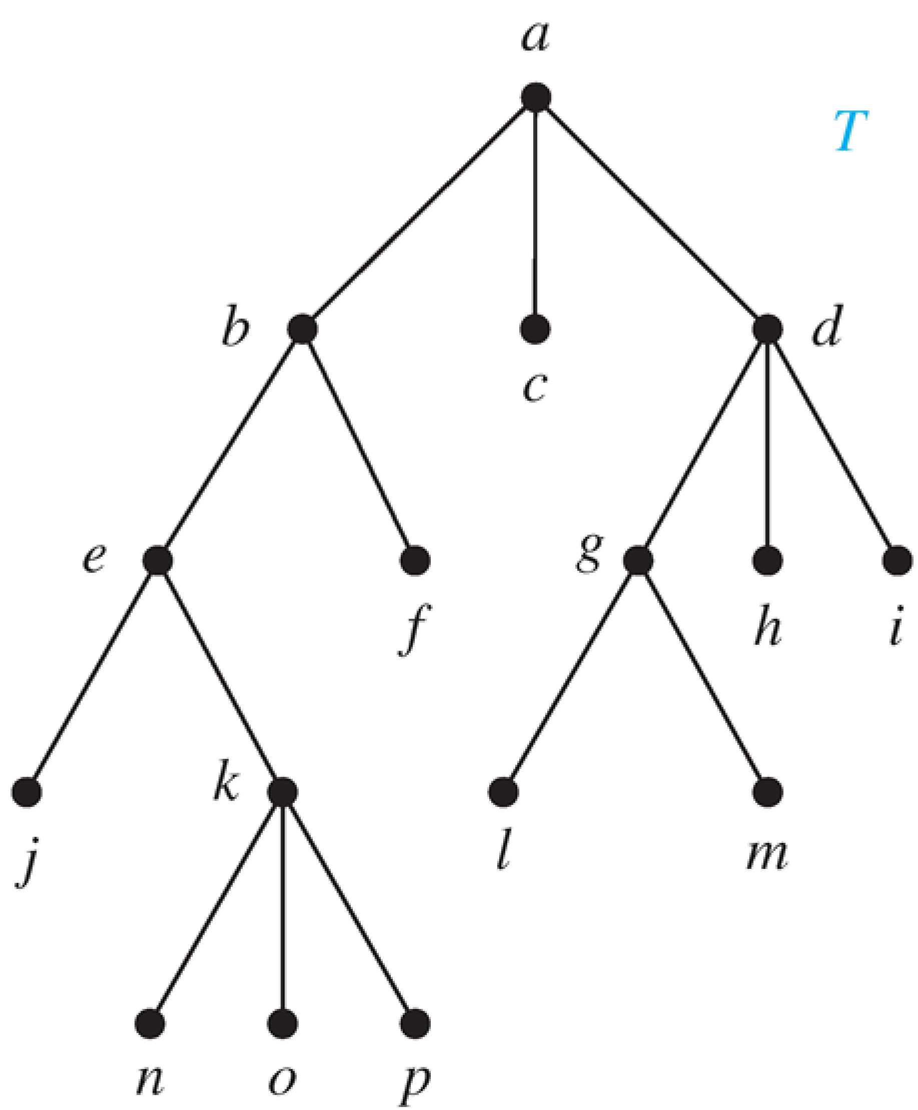
# Preorder Traversal

- **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

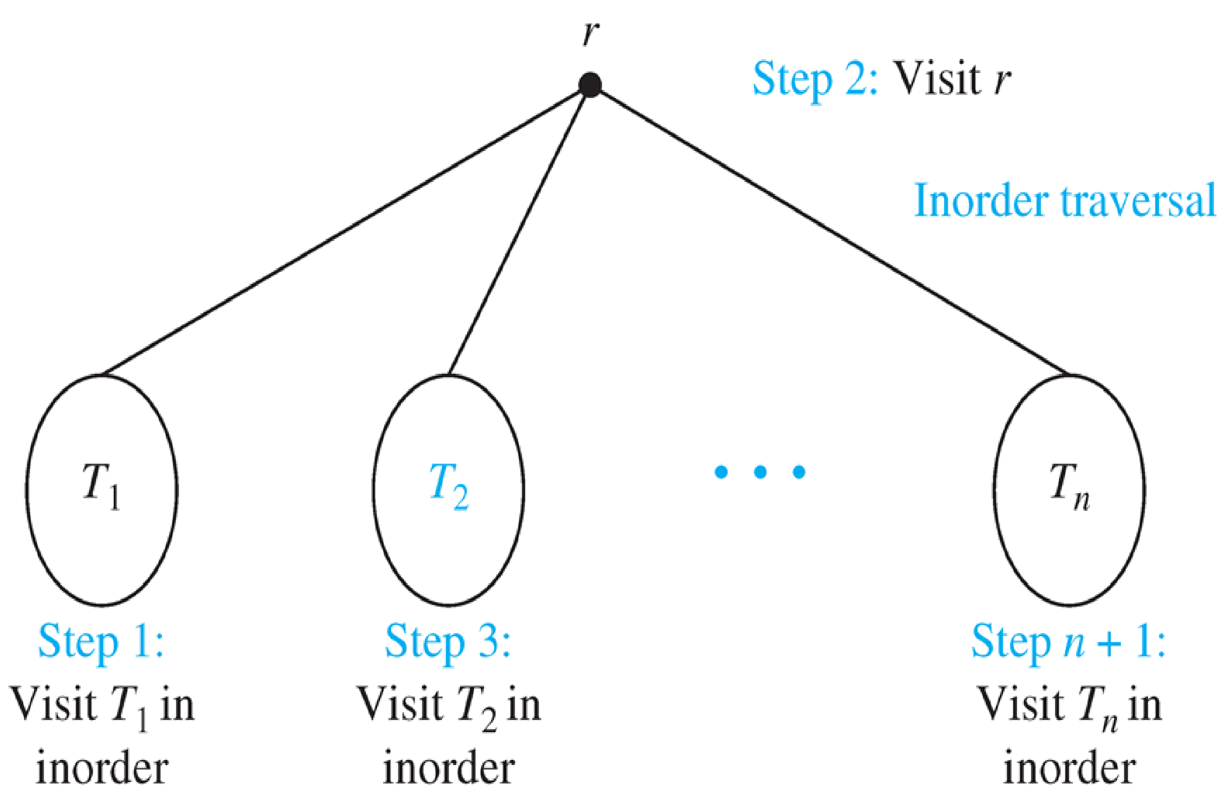
**procedure** *preorder* ($T$: ordered rooted tree)
$r$ := root of $T$
list $r$
**for** each child $c$ of $r$ from left to right
    $T(c)$ := subtree with $c$ as root
    *preorder*($T(c)$)

# 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.

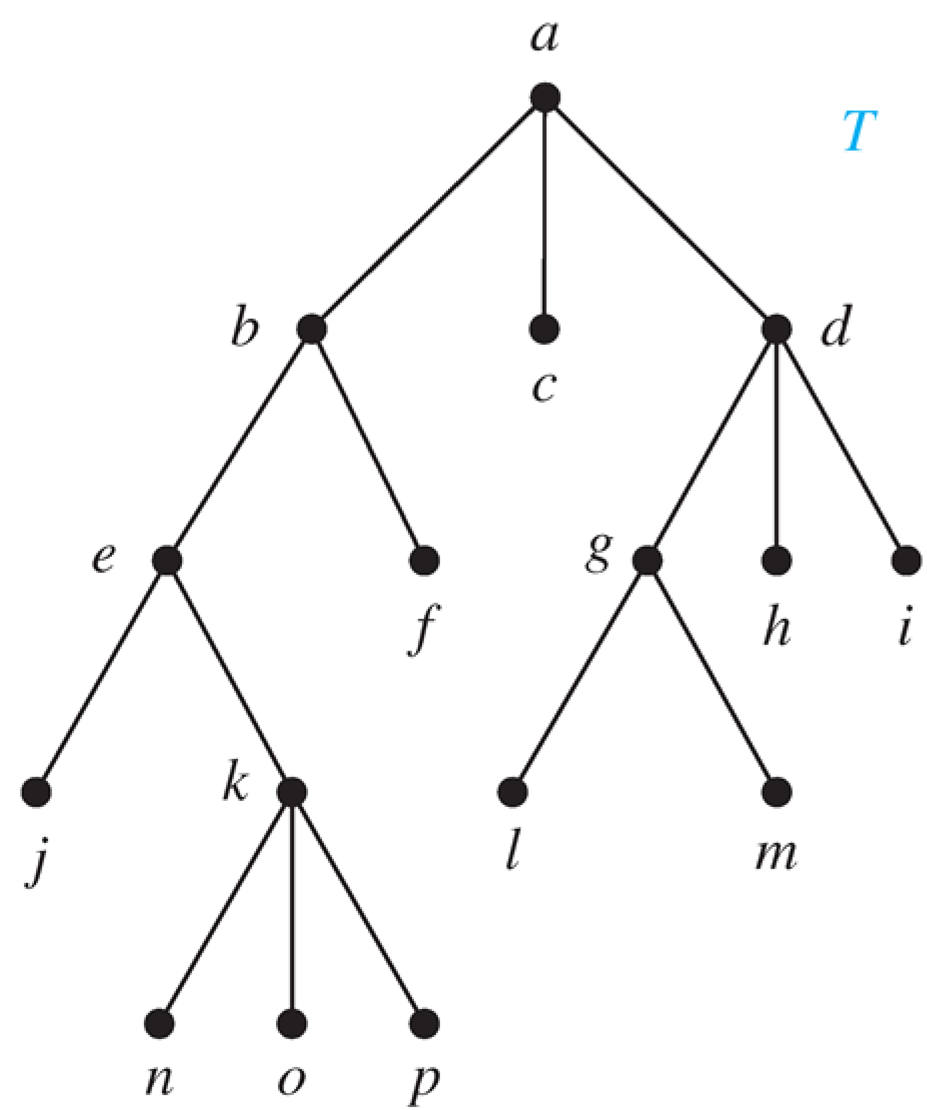
# Inorder Traversal

- **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

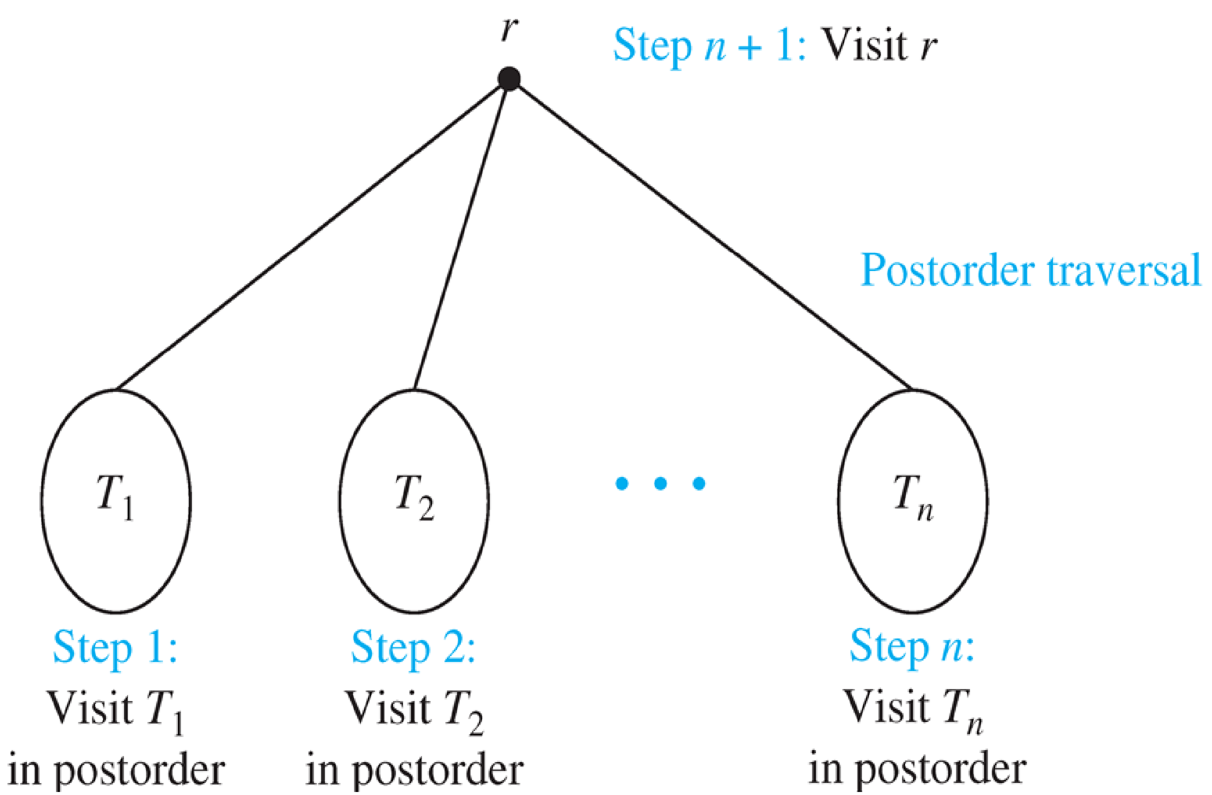
**procedure** *inorder* ($T$: ordered rooted tree)
$r$ := root of $T$
**if** $r$ is a leaf **then** list $r$
**else**
    $l$ := first child of $r$ from left to right
    $T(l)$ := subtree with $l$ as its root
    *inorder*($T(l)$)
    list($r$)
    **for** each child $c$ of $r$ from left to right
 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

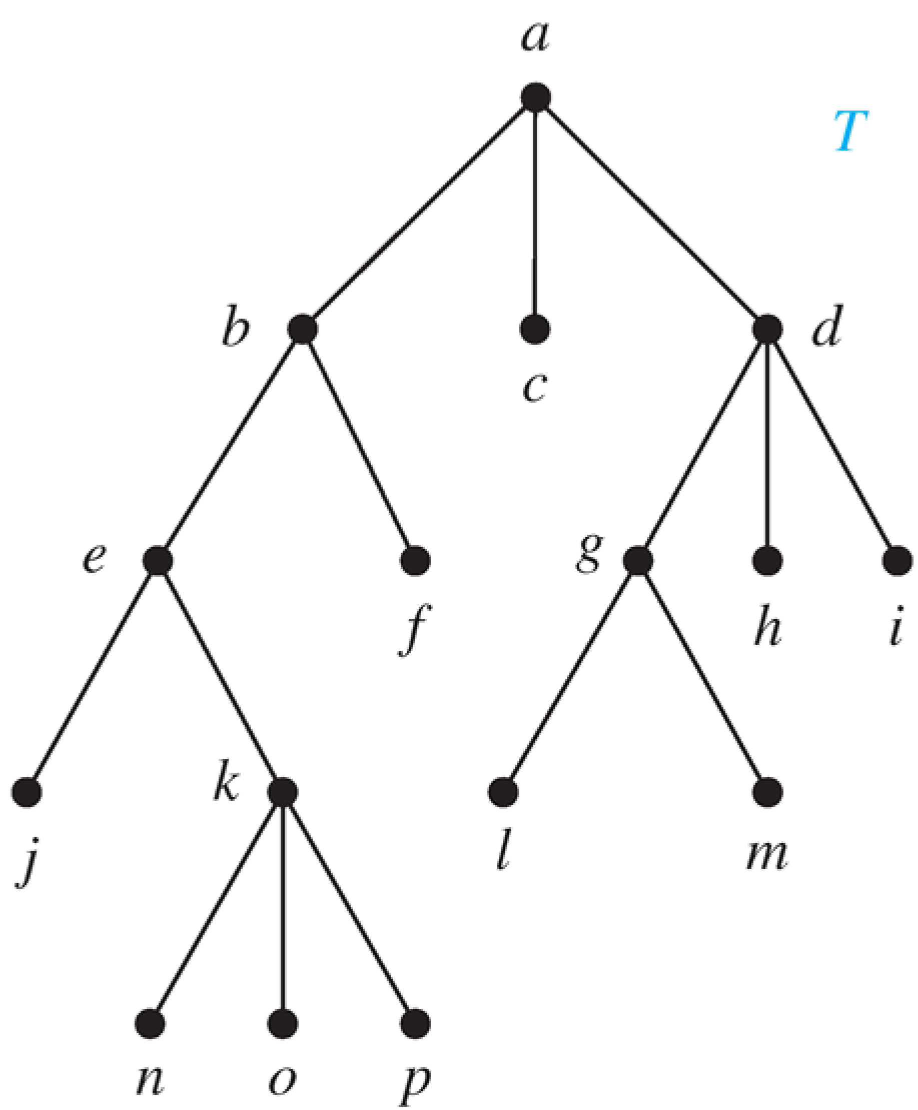
# Postorder Traversal

- **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

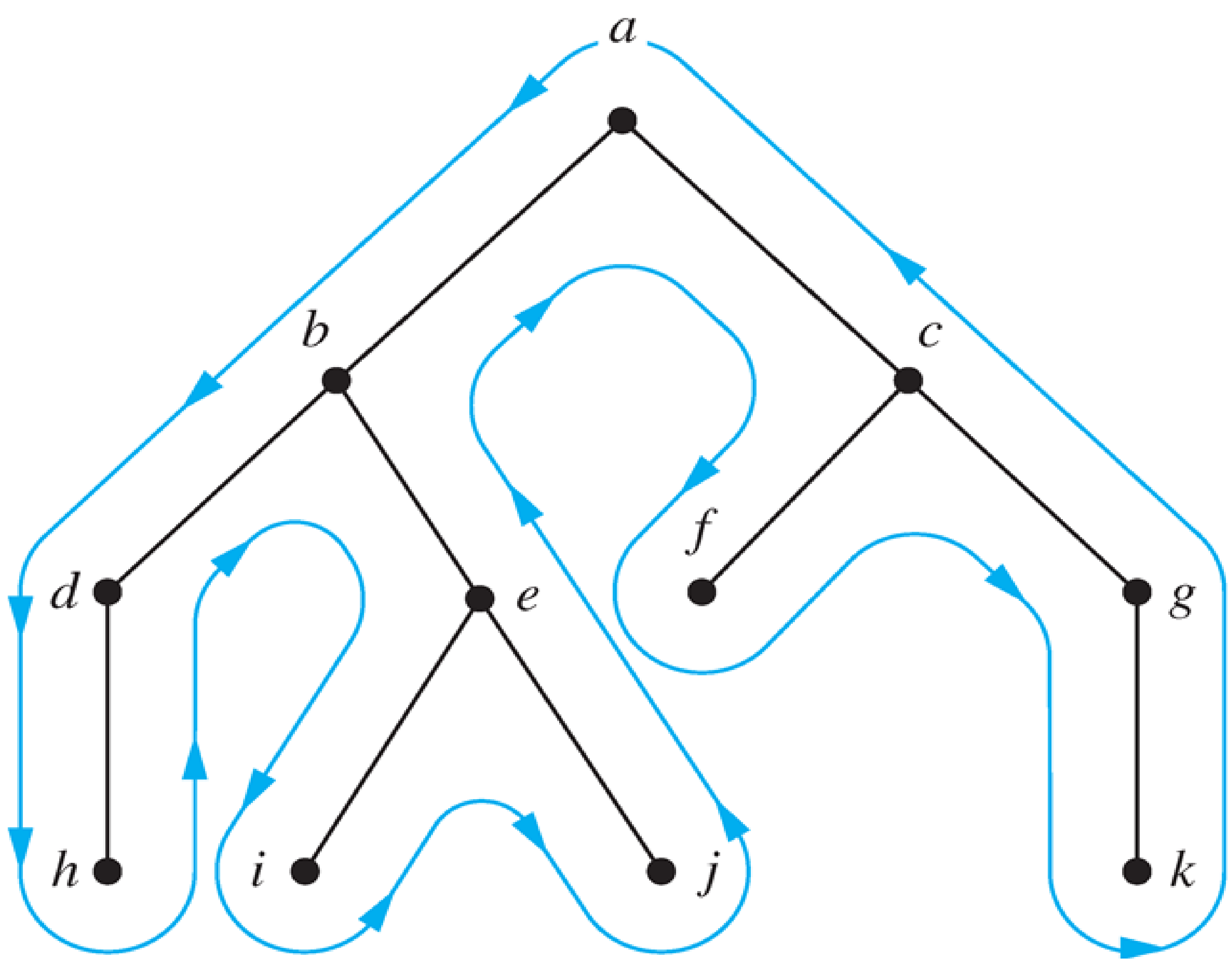
- **Example**



# 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**

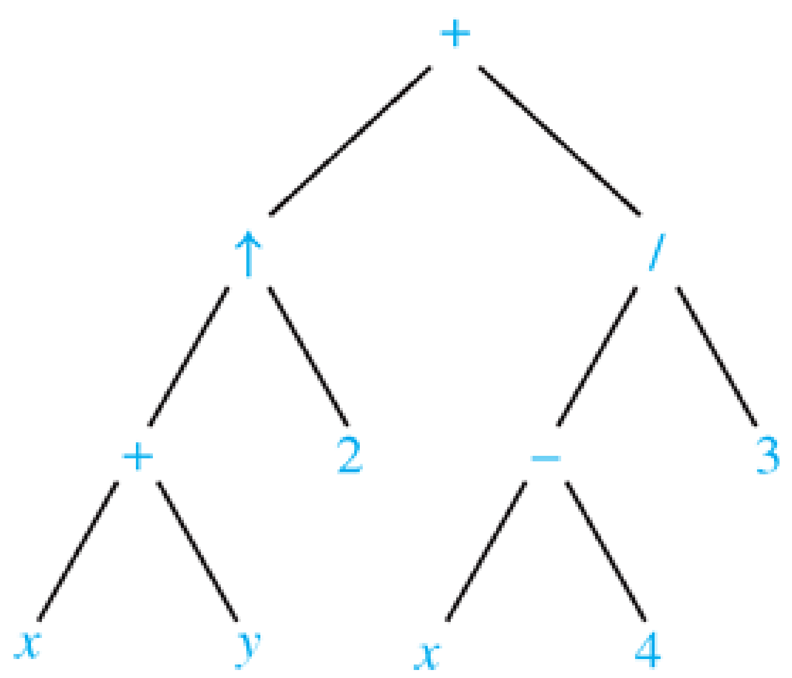
consider the expression $((x+y)\uparrow 2)+((x-4)/3)$

# 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

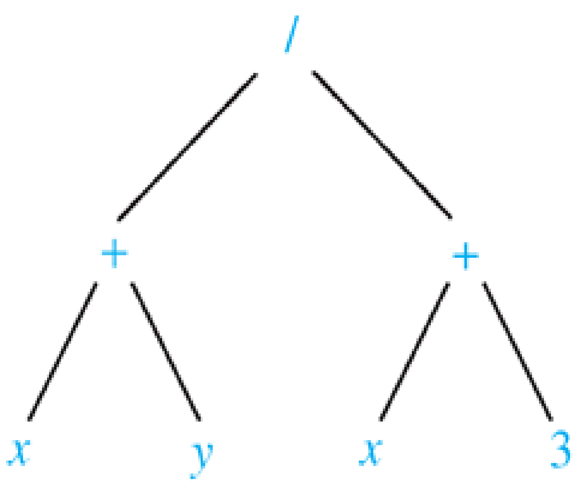
- An inorder traversal of the tree representing an expression produces the original expression when parentheses are included except for unary operation.

  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.

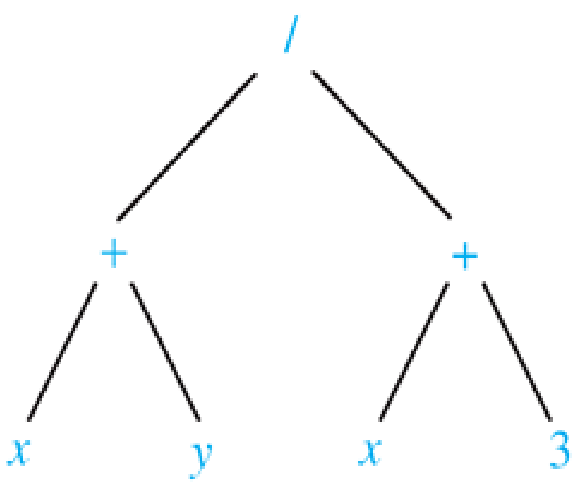
  Why parentheses are needed?

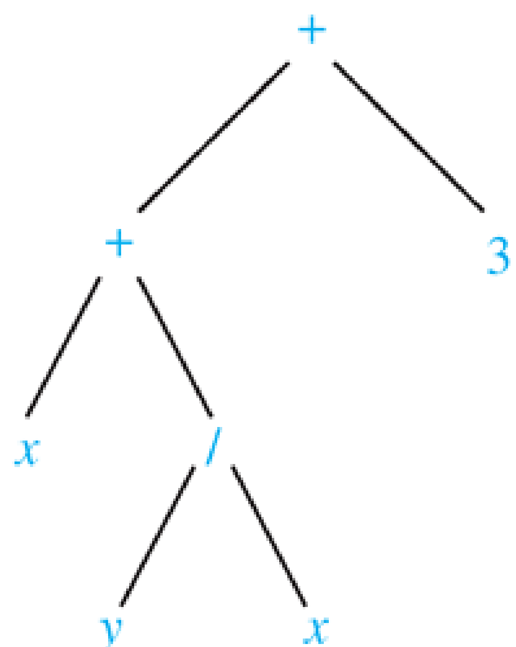


# Infix Notation

- 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?

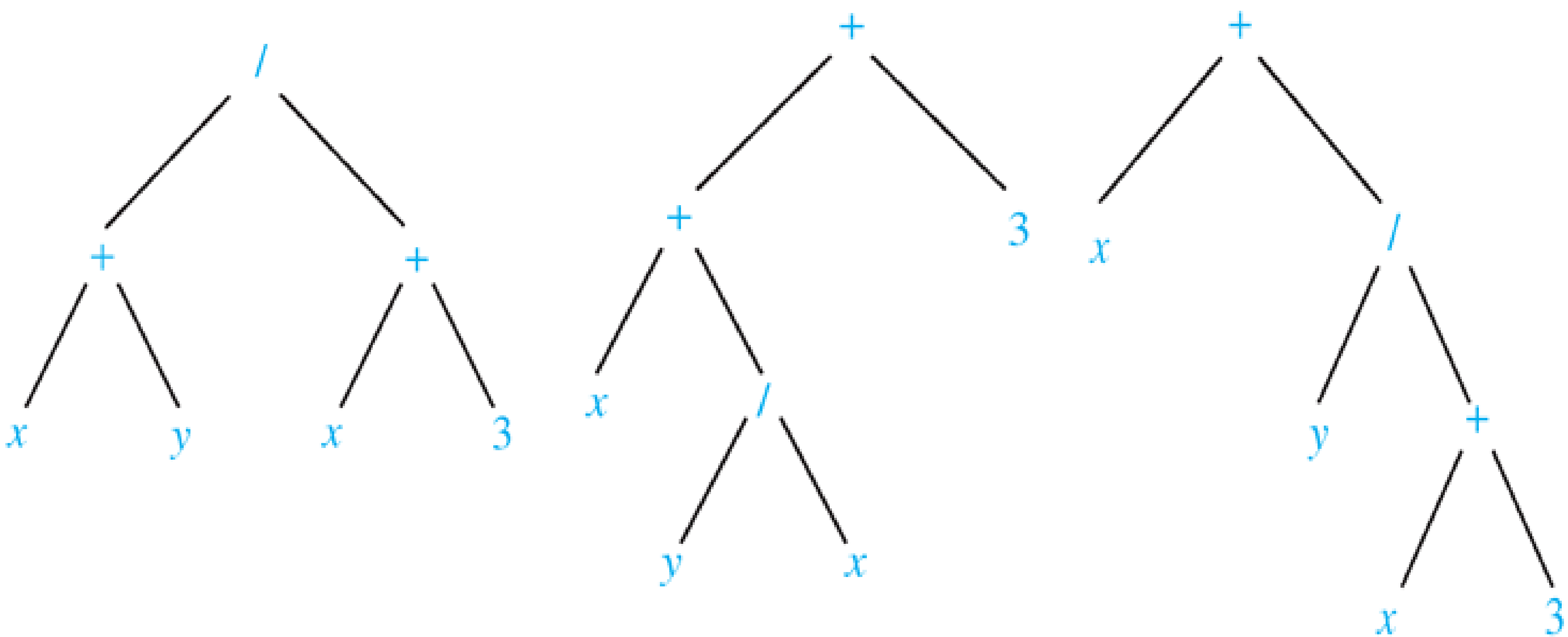


# Infix Notation

- 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**

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

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

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

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

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

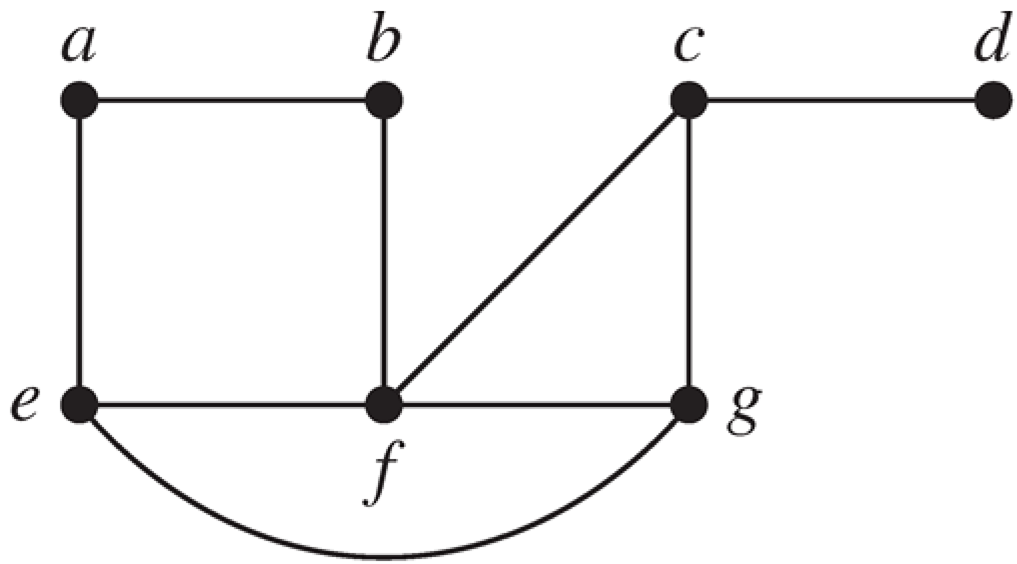
$$\underbrace{1\ \ 3\ \ +}_{1+3=4}$$

# 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 $G$.

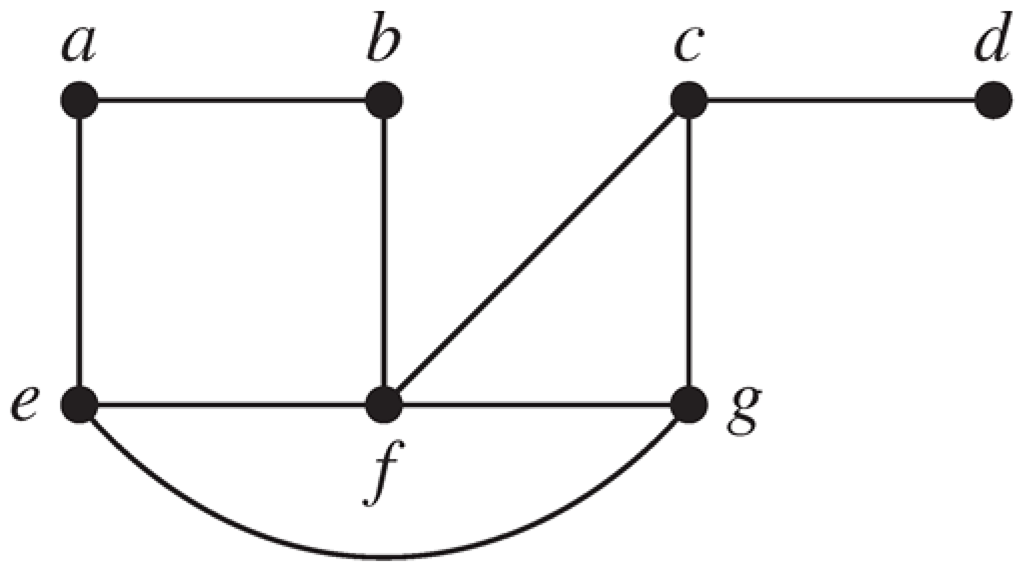
# Spanning Trees

- **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**

- **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.

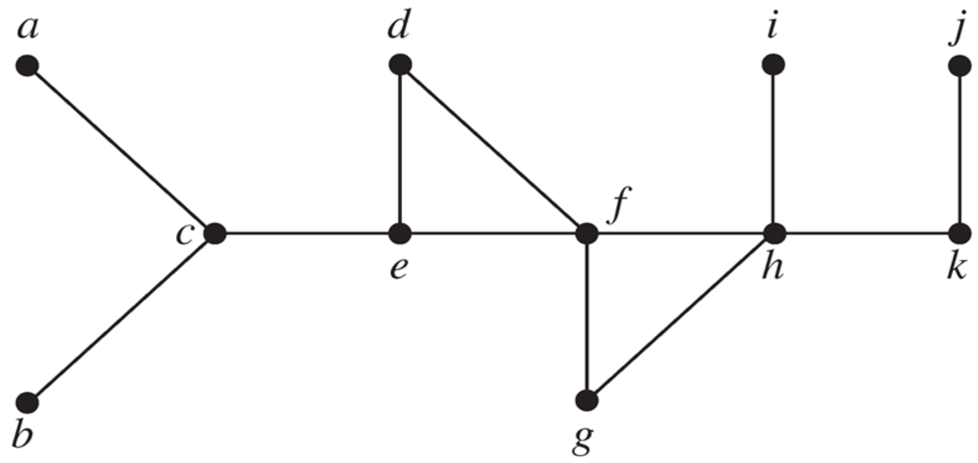
  ◇ First arbitrarily choose a vertex of the graph as the root.

  ◇ 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.

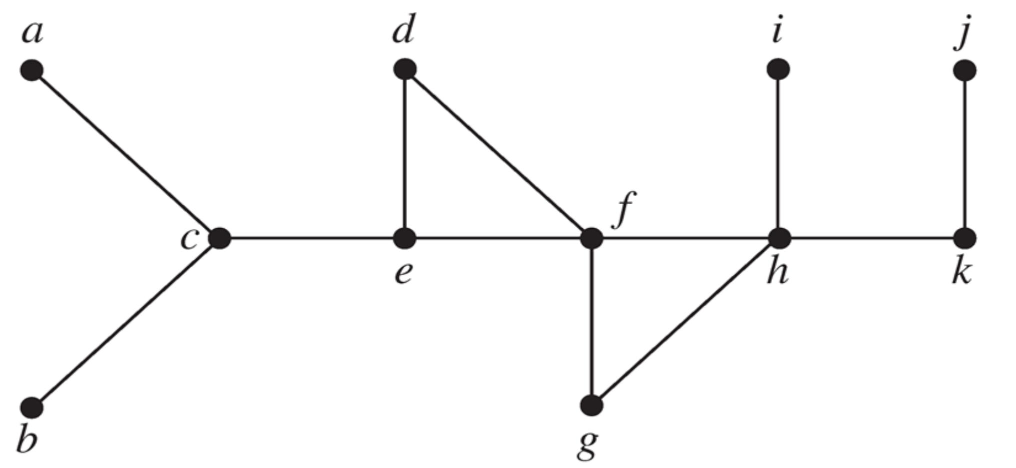
  ◇ Otherwise, move back to some vertex to repeat this procedure (*backtracking*)

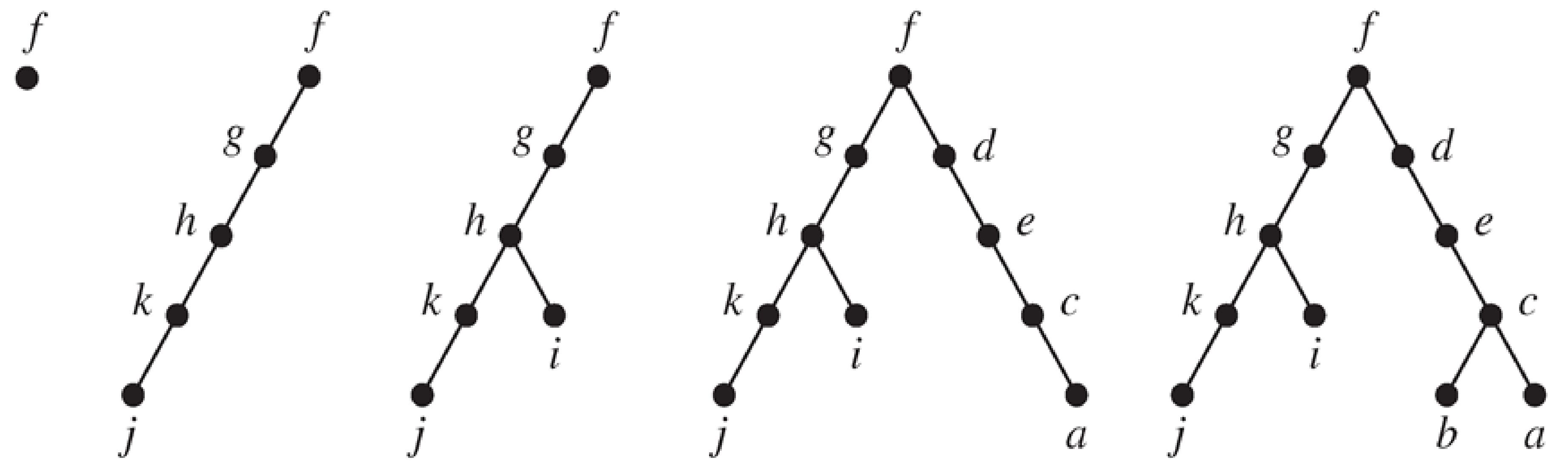
- **Example**



# 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.

# 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.

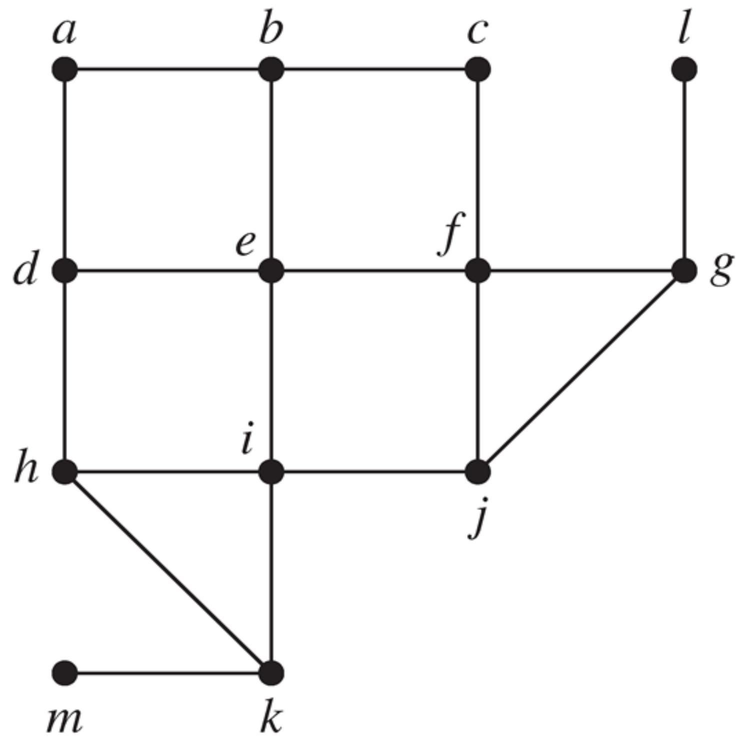
  ◇ Form a path by adding all edges incident to this vertex and the other endpoint of each of these edges

  ◇ 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.

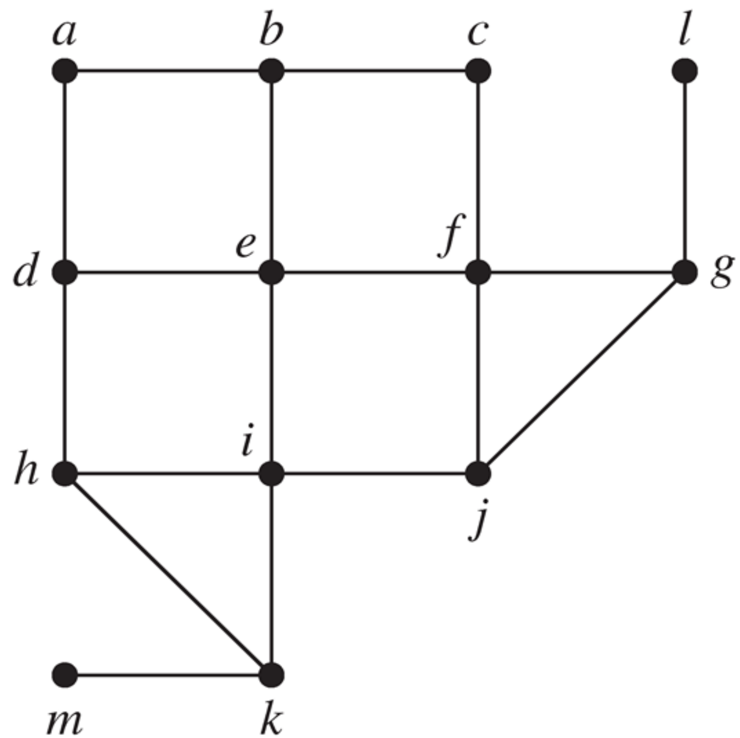
# Breadth-First Search

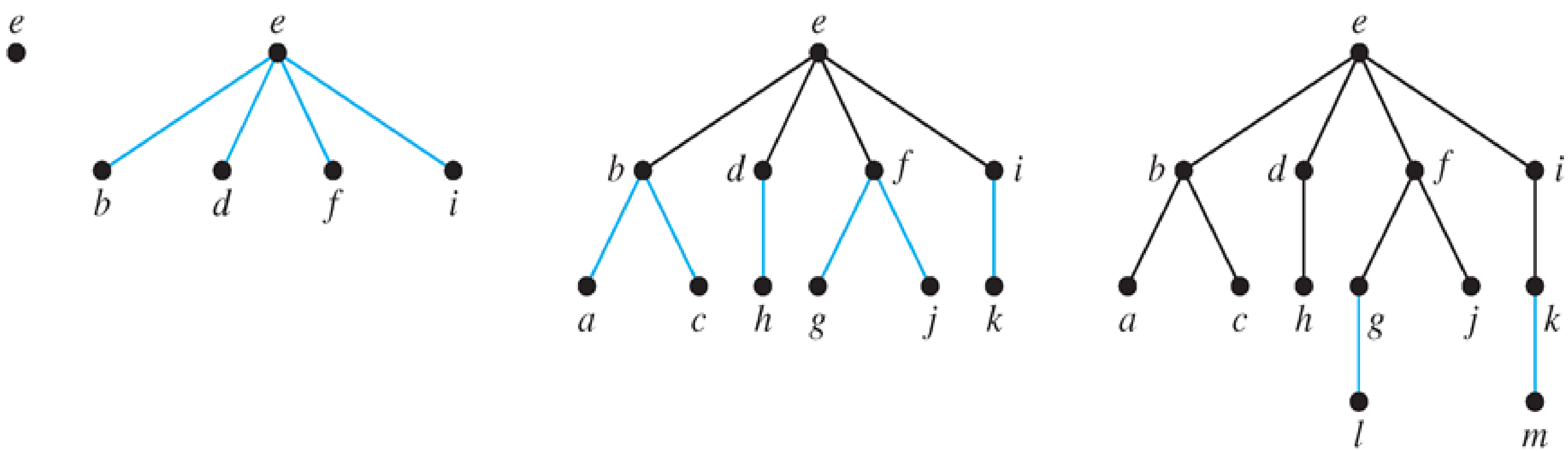
- **Example**



# 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, ...

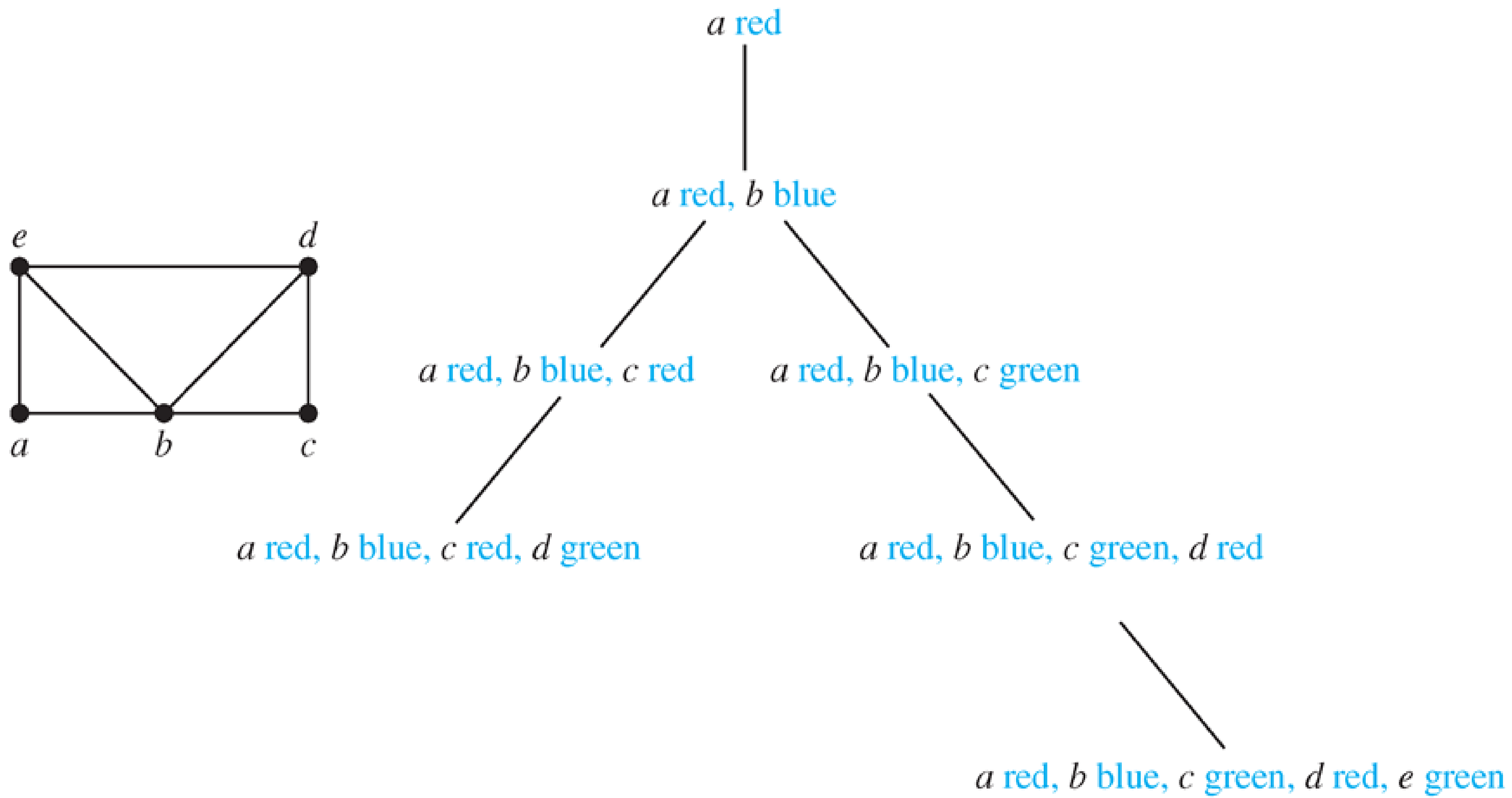
# Applications of DFS, BFS

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

  find shortest paths, determine whether bipartite, ...

  graph coloring, sums of subsets, ...

# Applications of DFS, BFS

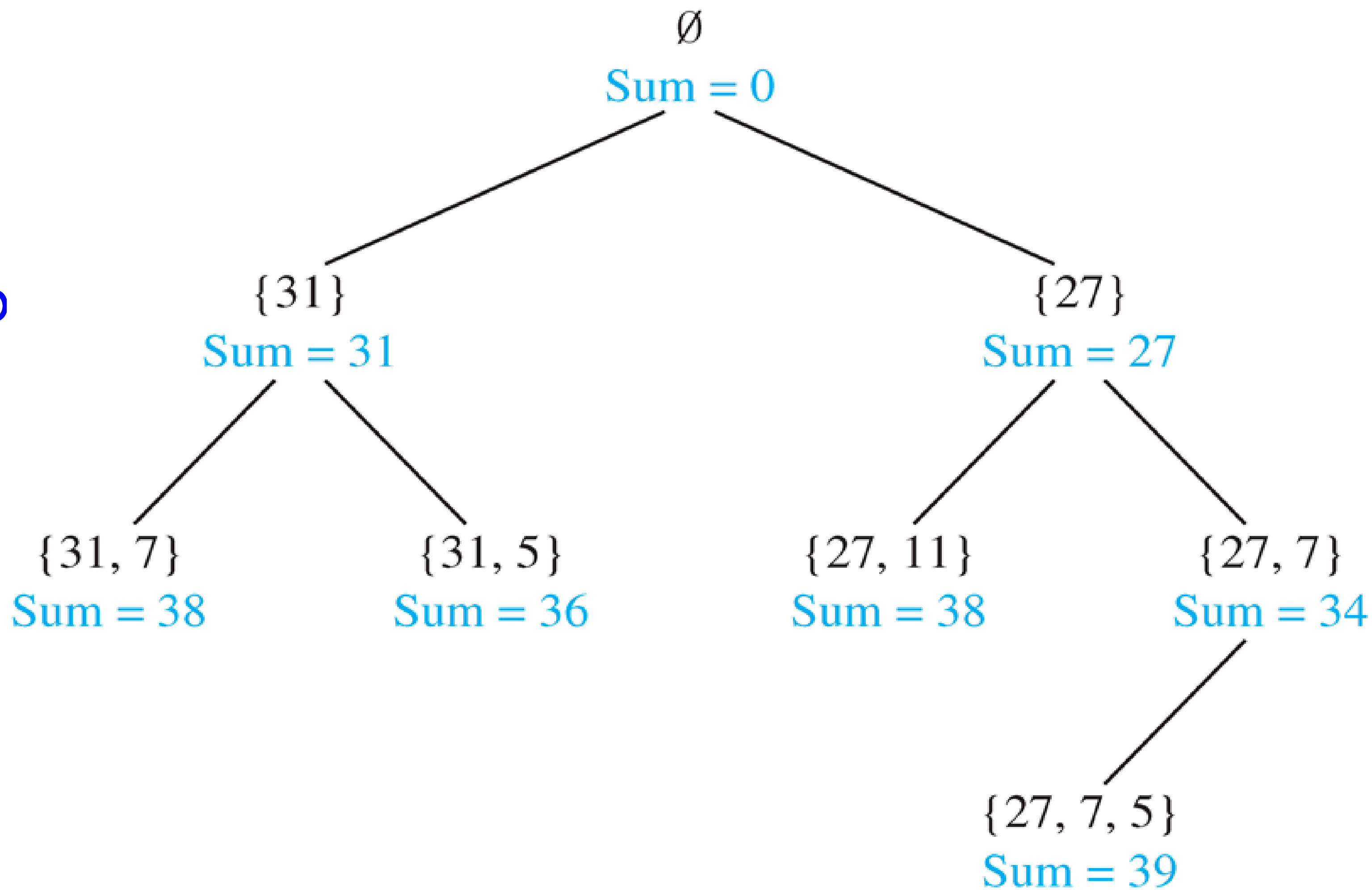


# Applications of DFS, BFS

- find [illegible]es, ...

  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

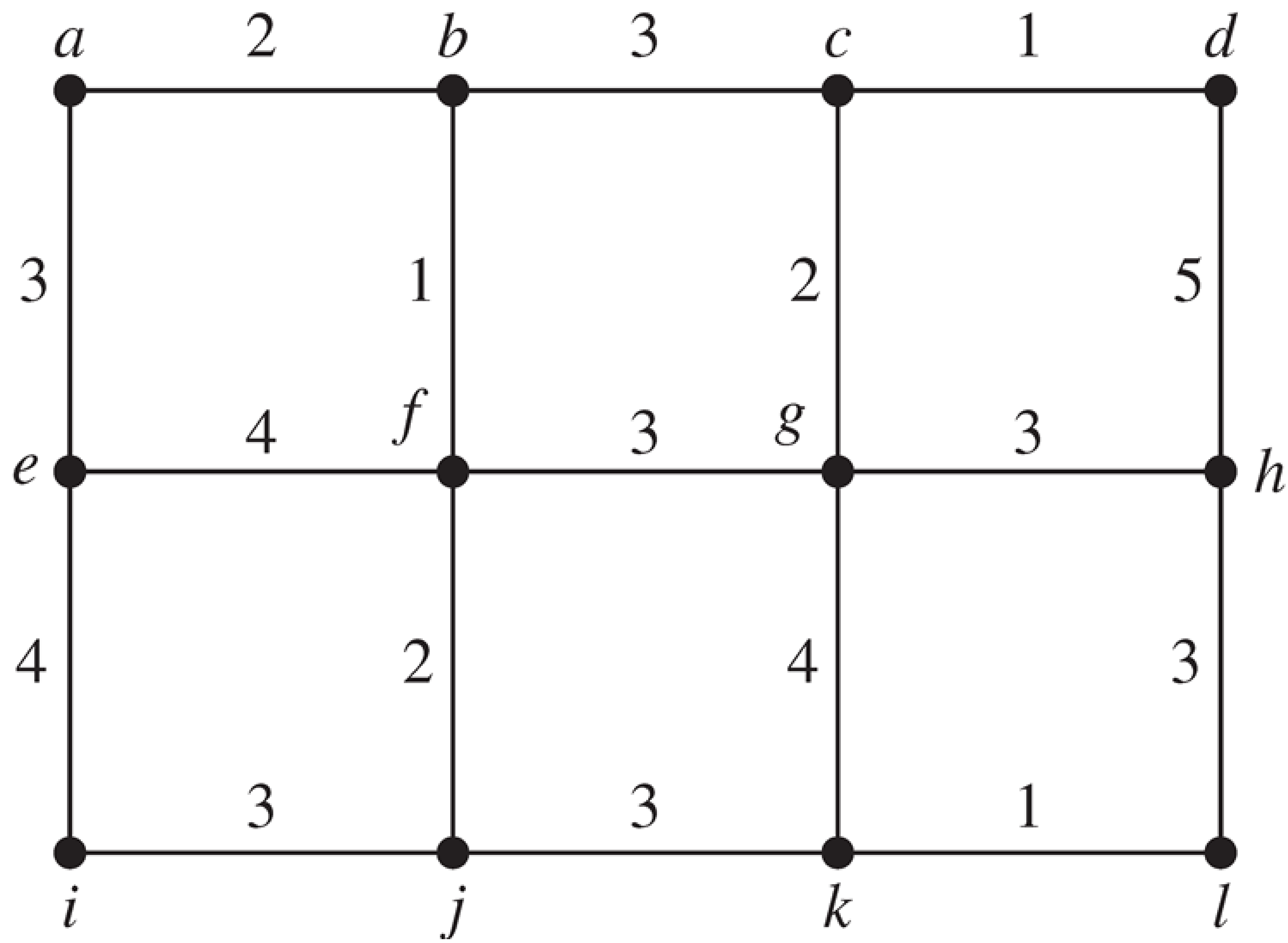
**ALGORITHM 1 Prim's Algorithm.**

**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

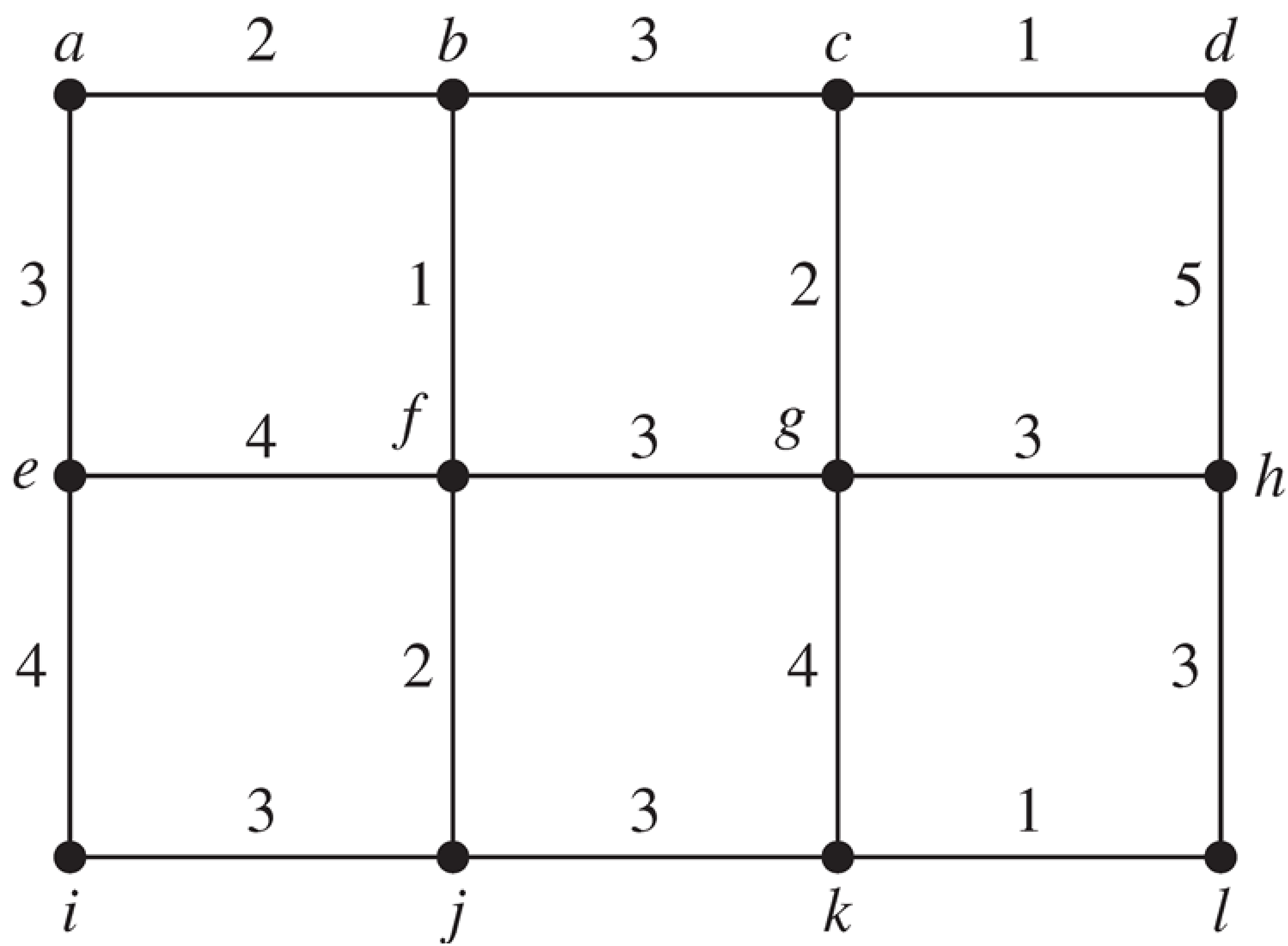
**ALGORITHM 2 Kruskal's Algorithm.**

**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**



# Review

01. Propositional Logic
02. Predicate Logic
03. Mathematical Proofs
04. Sets
05. Functions
06. Complexity of Algorithms
07. Number Theory
08. Cryptography
09. Mathematical Induction
10. Recursion
11. Counting
12. Relation
13. Graphs
14. Tree

# Review

01. Propositional Logic
02. Predicate Logic
03. Mathematical Proofs
04. Sets
05. Functions
06. Complexity of Algorithms
07. Number Theory
08. Cryptography
09. Mathematical Induction
10. Recursion
11. Counting
12. Relation
13. Graphs
14. Tree

Discrete Probability

Groups, Rings and Fields

# Logic

- Logical connectives

# Logic

- Logical connectives

  $\neg p,\ p \vee q,\ p \wedge q,\ p \oplus q,\ p \rightarrow q,\ p \leftrightarrow q$

# Logic

- Logical connectives

  $\neg p,\ p \vee q,\ p \wedge q,\ p \oplus q,\ p \rightarrow q,\ p \leftrightarrow q$

- Logical equivalence

# Logic

- Logical connectives

  $\neg p,\ p \vee q,\ p \wedge q,\ p \oplus q,\ p \rightarrow q,\ p \leftrightarrow q$

- Logical equivalence

  De Morgan's laws, communtative laws, distributive laws, ...

# Logic

- Logical connectives

  $\neg p$, $p \vee q$, $p \wedge q$, $p \oplus q$, $p \rightarrow q$, $p \leftrightarrow q$

- Logical equivalence

  De Morgan's laws, communtative laws, distributive laws, ...

- Predicate logic

  contains variables

# Logic

- Logical connectives

  $\neg p,\ p \vee q,\ p \wedge q,\ p \oplus q,\ p \rightarrow q,\ p \leftrightarrow q$

- Logical equivalence

  De Morgan's laws, communtative laws, distributive laws, ...

- Predicate logic

  contains variables

- Quantified statements

  universal, existential, equivalence

# Methods of Proving Theorems

- Basic methods to prove theorems:
  - ◇ *direct proof*
    - $p \to q$ is proved by showing that if $p$ is true then $q$ follows
  - ◇ *proof by contrapositive*
    - show the contrapositive $\neg q \to \neg p$
  - ◇ *proof by contradiction*
    - show that $(p \wedge \neg q)$ contradicts the assumptions
  - ◇ *proof by cases*
    - give proofs for all possible cases
  - ◇ *proof of equivalence*
    - $p \leftrightarrow q$ is replaced with $(p \to q) \wedge (q \to p)$

# Function

- function?

# Function

- function?

  one-to-one (injective) function?

# Function

- function?

  one-to-one (injective) function?

  onto (surjective) function?

# Function

- function?

  one-to-one (injective) function?

  onto (surjective) function?

  bijective function (one-to-one correspondence)?

# Function

- function?

  one-to-one (injective) function?

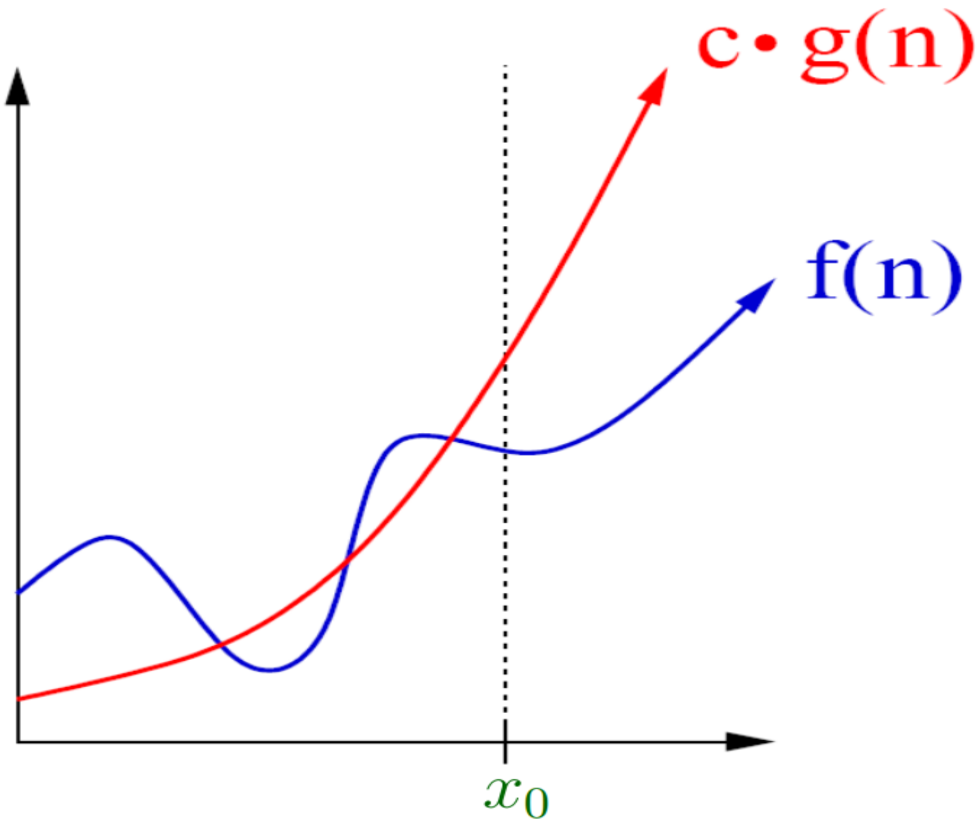
  onto (surjective) function?

  bijective function (one-to-one correspondence)?

- counting the number of such functions?

# Big-$O$ Notation

- Let $f$ and $g$ be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(n) = O(g(n))$ (reads: $f(n)$ is $O$ of $g(n)$), if there exist some positive constants $C$ and $k$ such that $|f(n)| \leq C|g(n)|$, whenever $n > k$.



# Number Theory

- Divisibility

# Number Theory

- Divisibility

  Congruence relation

# Number Theory

- Divisibility

  Congruence relation

  Primes

# Number Theory

- Divisibility

  Congruence relation

  Primes

  GCD and Euclidean Algorithm

# Number Theory

- Divisibility

  Congruence relation

  Primes

  GCD and Euclidean Algorithm

  Modular Inverse

# Number Theory

- Divisibility

  Congruence relation

  Primes

  GCD and Euclidean Algorithm

  Modular Inverse
  - When does an inverse of $a$ modulo $m$ exist?
  - How to find inverses?

# Number Theory

- Divisibility

  Congruence relation

  Primes

  GCD and Euclidean Algorithm

  Modular Inverse

    When does an inverse of $a$ modulo $m$ exist?

    How to find inverses?

  Chinese Remainder Theorem

# Number Theory

- Divisibility

  Congruence relation

  Primes

  GCD and Euclidean Algorithm

  Modular Inverse

    When does an inverse of $a$ modulo $m$ exist?

    How to find inverses?

  Chinese Remainder Theorem

  Back substitution

# Number Theory

- Divisibility

  Congruence relation

  Primes

  GCD and Euclidean Algorithm

  Modular Inverse

  When does an inverse of $a$ modulo $m$ exist?

  How to find inverses?

  Chinese Remainder Theorem

  Back substitution

$$\begin{aligned} x &\equiv 2 \pmod 3 \\ x &\equiv 3 \pmod 5 \\ x &\equiv 2 \pmod 7 \end{aligned}$$

- Fermat's Little Theorem

# Cryptography

- Fermat's Little Theorem

  Euler's Theorem

- Fermat's Little Theorem

  Euler's Theorem

  Primitive roots, multiplicative order

# Cryptography

- Fermat's Little Theorem

  Euler's Theorem

  Primitive roots, multiplicative order

  RSA cryptosystem

  DLP, Diffie-Hellman protocol

# Mathematical Induction

- A *typical* proof by mathematical induction, showing that a statement $P(n)$ is true for all integers $n \geq b$ consists of three steps:

# Mathematical Induction

- A *typical* proof by mathematical induction, showing that a statement $P(n)$ is true for all integers $n \geq b$ consists of three steps:

  1. We show that $P(b)$ is true. – Base Step

# Mathematical Induction

- A *typical* proof by mathematical induction, showing that a statement $P(n)$ is true for all integers $n \geq b$ consists of three steps:

1. We show that $P(b)$ is true. – Base Step

2. We then, $\forall n > b$, show either

$$(*) \qquad P(n-1) \to P(n)$$

or

$$(**) \qquad P(b) \land P(b+1) \land \cdots \land P(n-1) \to P(n)$$

# Mathematical Induction

- A *typical* proof by mathematical induction, showing that a statement $P(n)$ is true for all integers $n \geq b$ consists of three steps:

  1. We show that $P(b)$ is true. – Base Step
  2. We then, $\forall n > b$, show either

     $$(*) \qquad P(n-1) \rightarrow P(n)$$

     or

     $$(**) \qquad P(b) \wedge P(b+1) \wedge \cdots \wedge P(n-1) \rightarrow P(n)$$

     We need to make the inductive hypothesis of either $P(n-1)$ or $P(b) \wedge P(b+1) \wedge \cdots \wedge P(n-1)$.We then use $(*)$ or $(**)$ to derive $P(n)$.

# Mathematical Induction

- A *typical* proof by mathematical induction, showing that a statement $P(n)$ is true for all integers $n \geq b$ consists of three steps:

1. We show that $P(b)$ is true. – Base Step
2. We then, $\forall n > b$, show either

$$(*) \qquad P(n-1) \to P(n)$$

or

$$(**) \qquad P(b) \wedge P(b+1) \wedge \cdots \wedge P(n-1) \to P(n)$$

   We need to make the inductive hypothesis of either $P(n-1)$ or $P(b) \wedge P(b+1) \wedge \cdots \wedge P(n-1)$.We then use $(*)$ or $(**)$ to derive $P(n)$.

3. We conclude on the basis of the principle of mathematical induction that $P(n)$ is true for all $n \geq b$.

# Recurrence

- Iterating a recurrence

# Recurrence

- Iterating a recurrence

  bottom up or top down

# Recurrence

- Iterating a recurrence

  bottom up or top down

  prove by induction, complexity, ...

# Counting

- The sum rule and product rule

- The sum rule and product rule

  The Inclusion-Exclusion Principle

# Counting

- The sum rule and product rule

  The Inclusion-Exclusion Principle

  The Pigeonhole Principle

# Counting

- The sum rule and product rule

  The Inclusion-Exclusion Principle

  The Pigeonhole Principle

  **Theorem** If $N$ is a positive integer and $k$ is an integer with $1 \leq k \leq n$, then there are
  $P(n, k) = n(n-1)(n-2)\cdots(n-k+1)$
  $k$-element permutations with $n$ distinct elements.

# Counting

- The sum rule and product rule

  The Inclusion-Exclusion Principle

  The Pigeonhole Principle

  **Theorem** If $N$ is a positive integer and $k$ is an integer with $1 \leq k \leq n$, then there are
  $P(n, k) = n(n-1)(n-2)\cdots(n-k+1)$
  $k$-element permutations with $n$ distinct elements.

  $P(n, 3) = 3! \cdot C(n, 3)$

# Counting

- The sum rule and product rule

  The Inclusion-Exclusion Principle

  The Pigeonhole Principle

  **Theorem** If $N$ is a positive integer and $k$ is an integer with $1 \leq k \leq n$, then there are $P(n, k) = n(n-1)(n-2)\cdots(n-k+1)$ $k$-element permutations with $n$ distinct elements.

  $$P(n, 3) = 3! \cdot C(n, 3)$$

  Pascal's Triangle, Identity

# Counting

- The sum rule and product rule

  The Inclusion-Exclusion Principle

  The Pigeonhole Principle

  **Theorem** If $N$ is a positive integer and $k$ is an integer with $1 \leq k \leq n$, then there are $P(n,k) = n(n-1)(n-2)\cdots(n-k+1)$ $k$-element permutations with $n$ distinct elements.

  $$P(n,3) = 3! \cdot C(n,3)$$

  Pascal's Triangle, Identity

  The Binomial Theorem, Trinomial

# Binary Relations

- Properties of relations

# Binary Relations

- Properties of relations

  Representing relations

# Binary Relations

- Properties of relations

  Representing relations

  Closures on relations

# Binary Relations

- Properties of relations

  Representing relations

  Closures on relations

  Equivalence relation

  **Definition** A relation $R$ on a set $A$ is called an *equivalence relation* if it is reflexive, symmetric, and transitive.

# Binary Relations

- Properties of relations

  Representing relations

  Closures on relations

  Equivalence relation

  **Definition** A relation $R$ on a set $A$ is called an *equivalence relation* if it is reflexive, symmetric, and transitive.

  Partial ordering

# Binary Relations

- Properties of relations

  Representing relations

  Closures on relations

  Equivalence relation

  **Definition** A relation $R$ on a set $A$ is called an *equivalence relation* if it is reflexive, symmetric, and transitive.

  Partial ordering

  **Definition** A relation $R$ on a set $A$ is called a *partial ordering* if it is reflexive, antisymmetric, and transitive.

- Basic concepts

# Graphs & Trees

- Basic concepts

  connected graph, simple graph, isomophism, chromatic number, Euler circuit, Hamilton circuit, shortest path, bipartite graph, complete graph, special graphs ($K_n$, $K_{m,n}$, $C_n$, $W_n$ ), m-ary tree, tree traversal, spanning tree ...

# Next Lecture

- the last lecture ...