Department of
Computer Science and Engineering
计算机科学与工程系

CS215 DISCRETE MATH

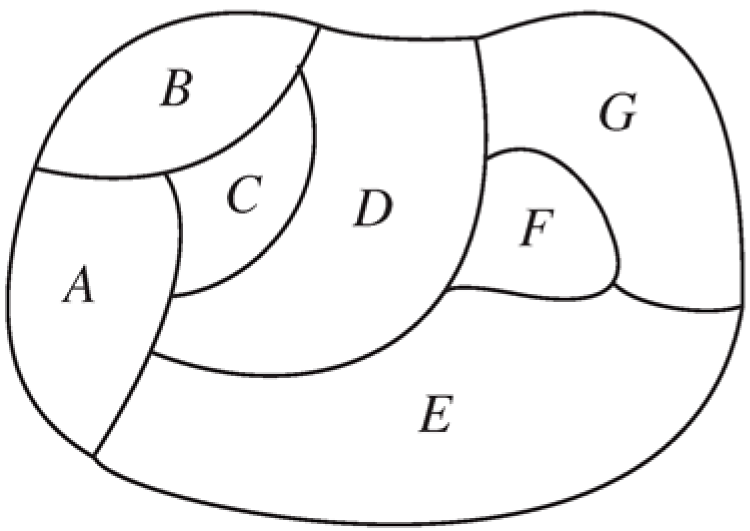
Dr. Qi Wang

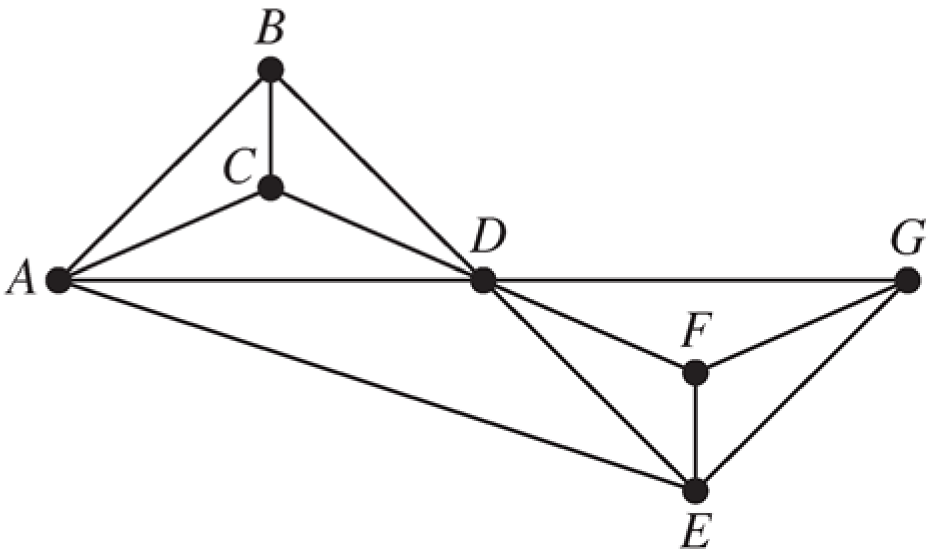
Department of Computer Science and Engineering
Office: Room413, CoE South Tower
Email: wangqi@sustech.edu.cn

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 $\chi(G)$.



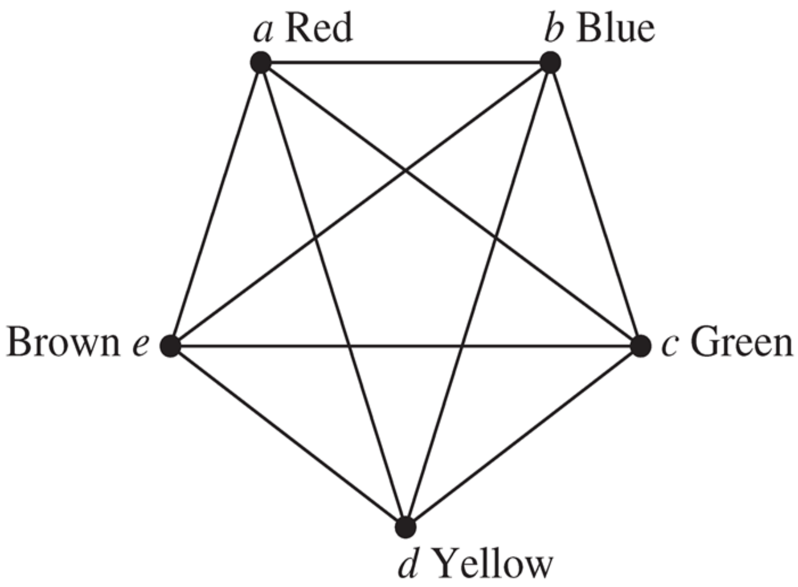


Examples

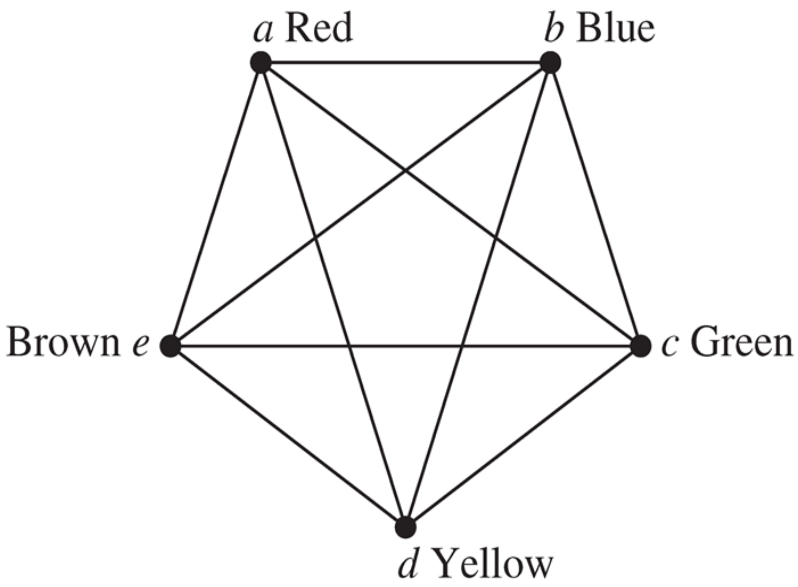
- What is the chromatic number of K_n, $K_{m,n}$, C_n?

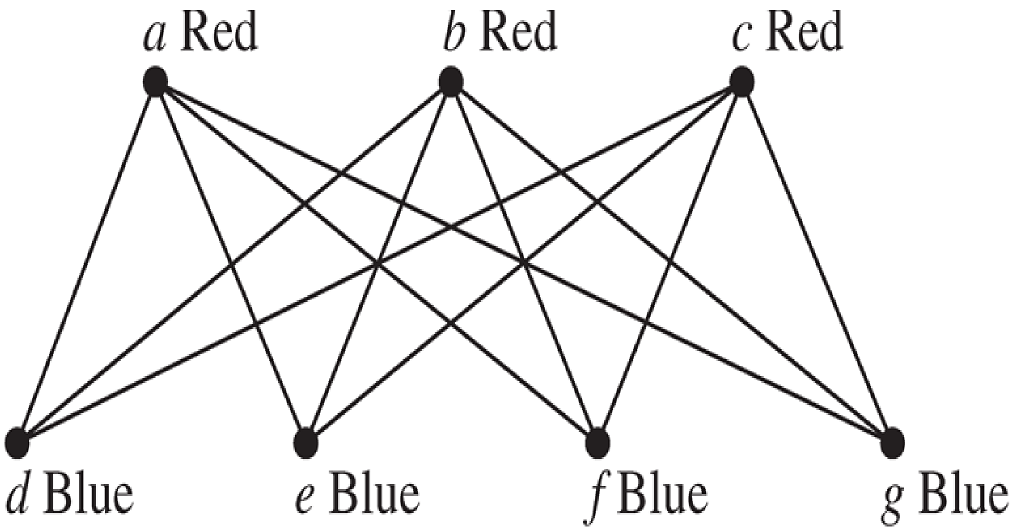
Examples

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



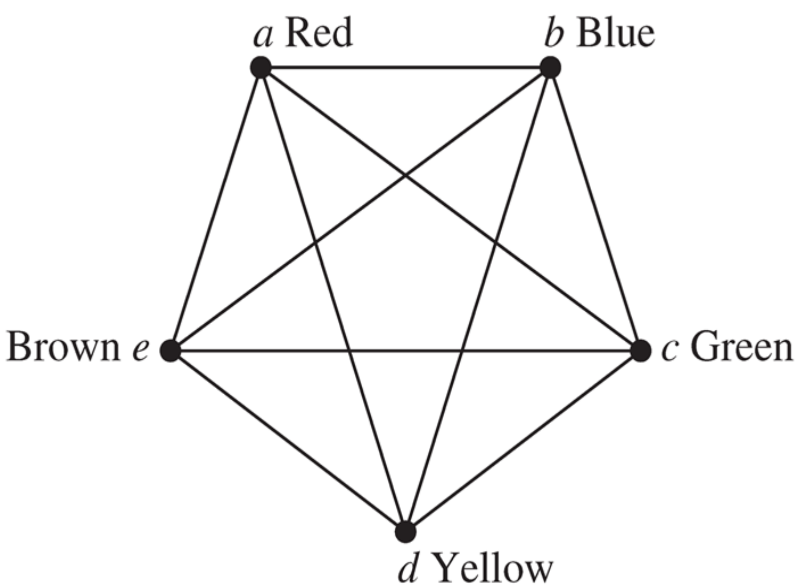
- What is the chromatic number of K_n, $K_{m,n}$, C_n?

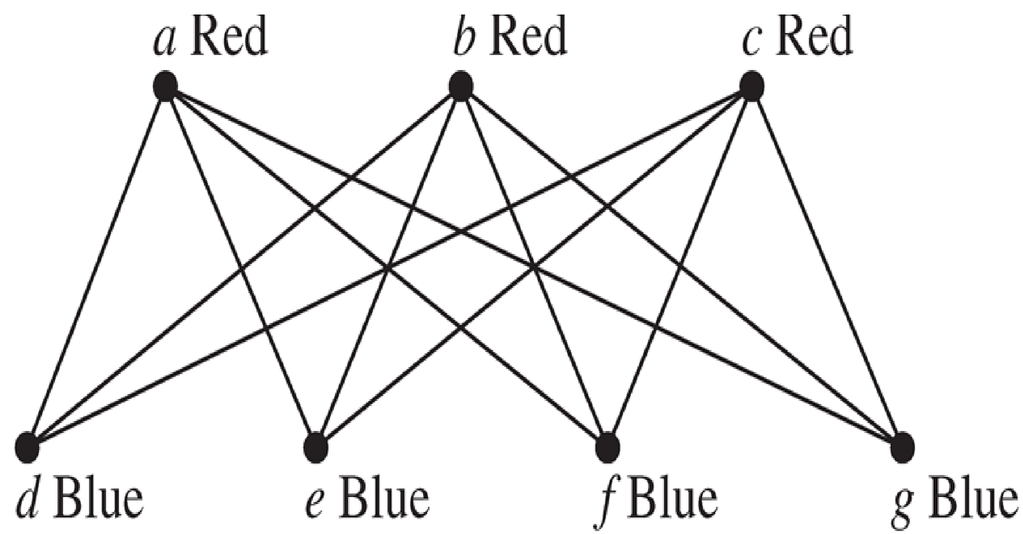


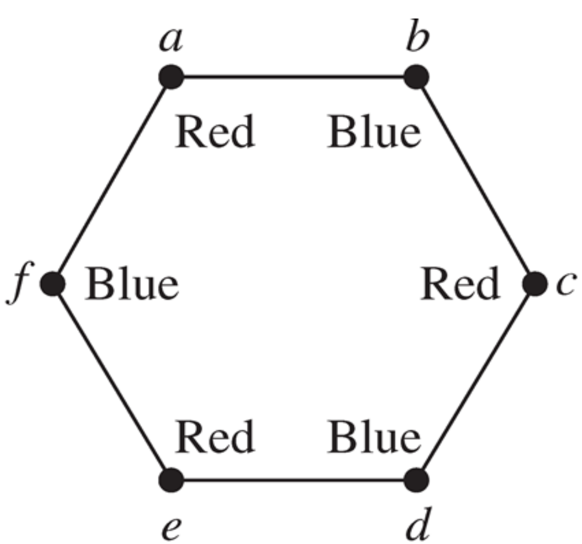


Examples

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

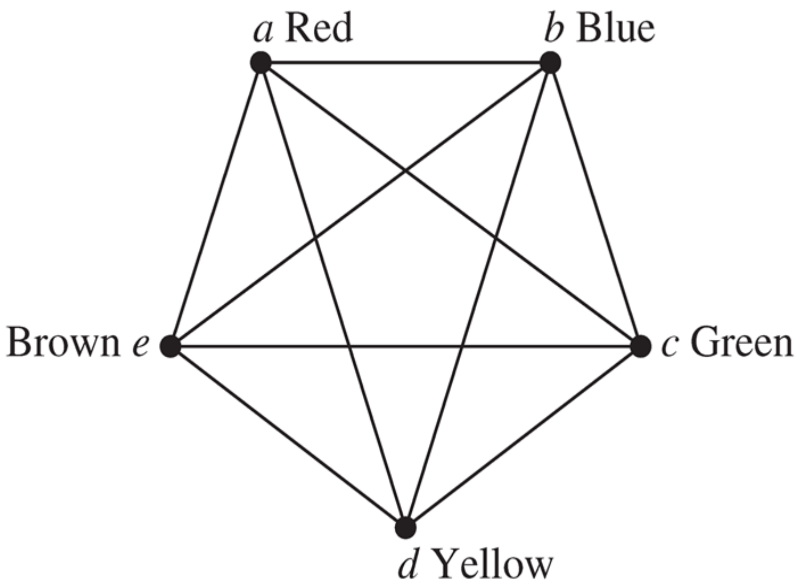


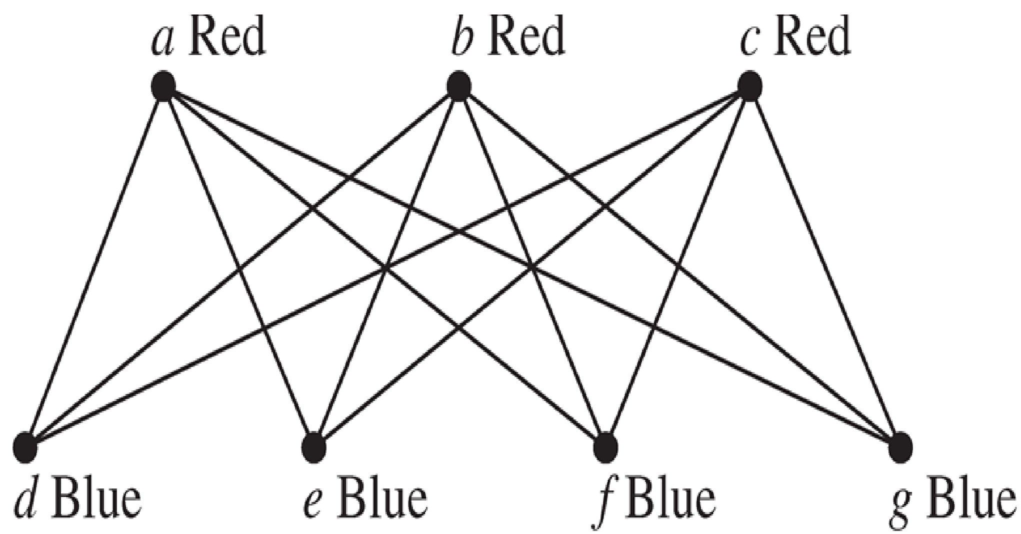


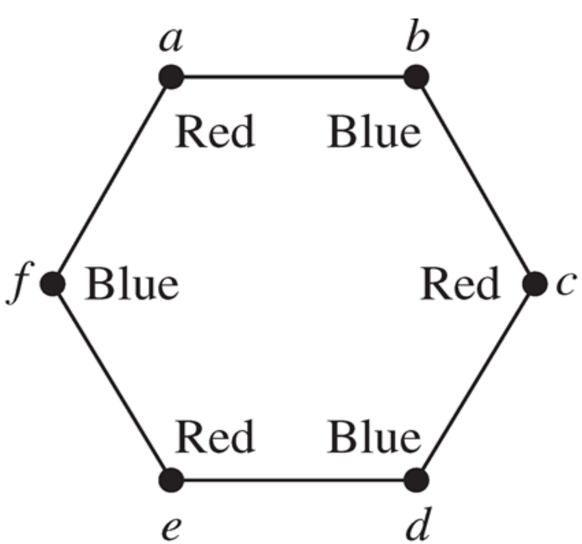


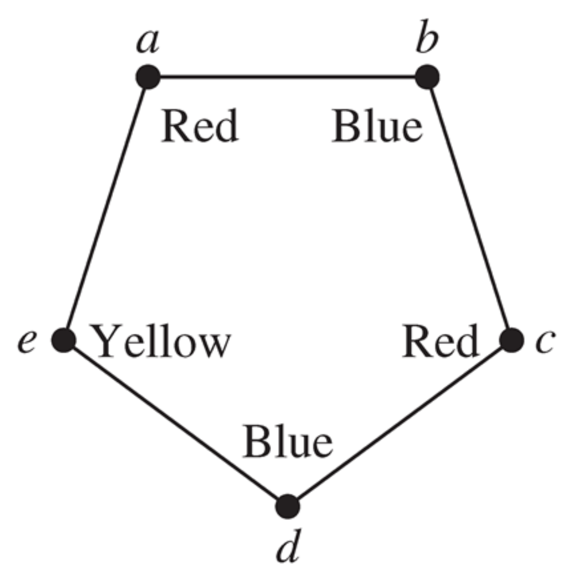
Examples

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





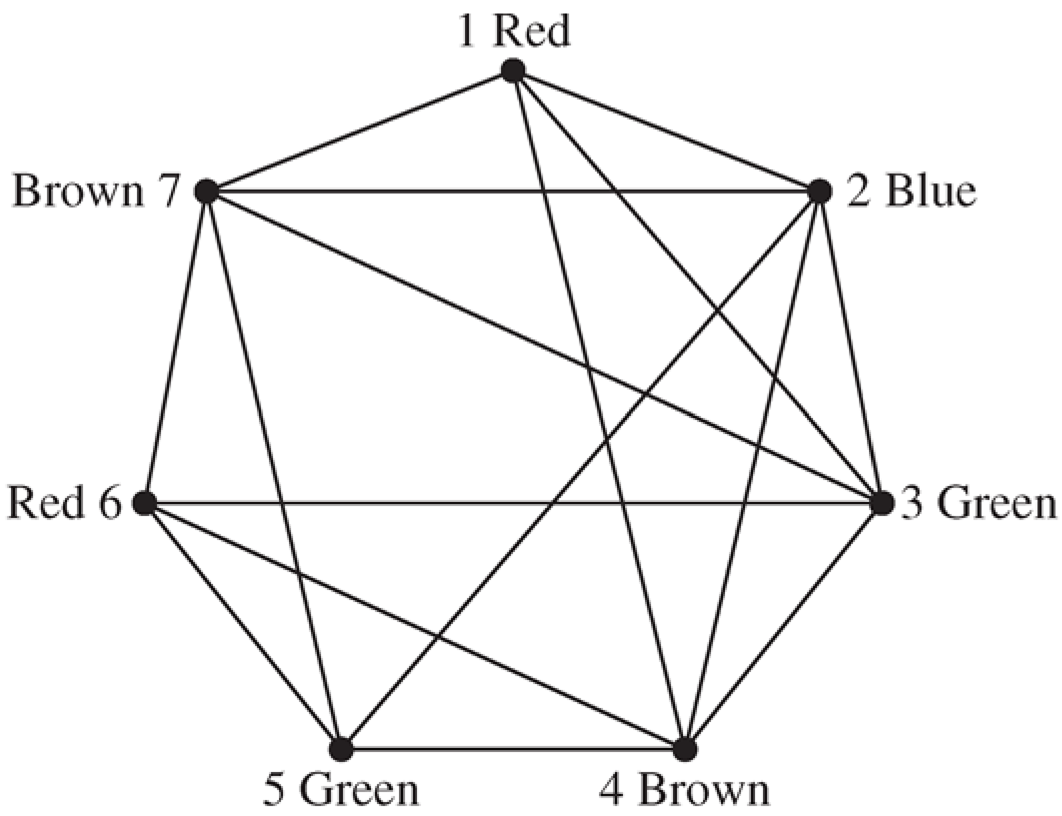




Applications of Graph Coloring

- 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

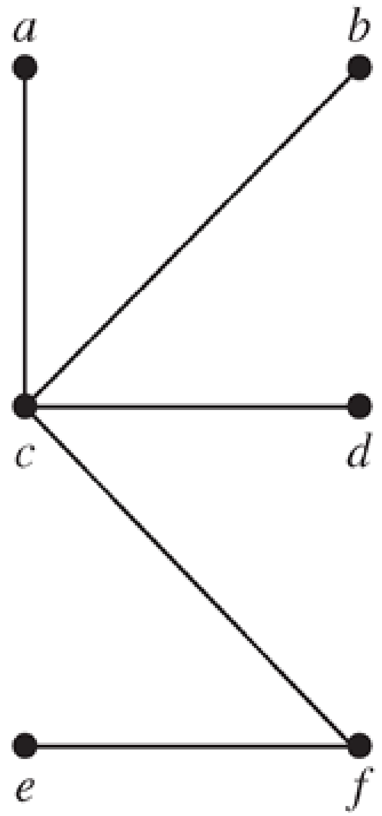
  Television channels 2 through 13 are assigned to stations in North America so that no two stations within 150 miles can operate on the same channel . How can the assignment of channels be modeled by graph coloring?

Graph Coloring $\in$ NPC

Trees

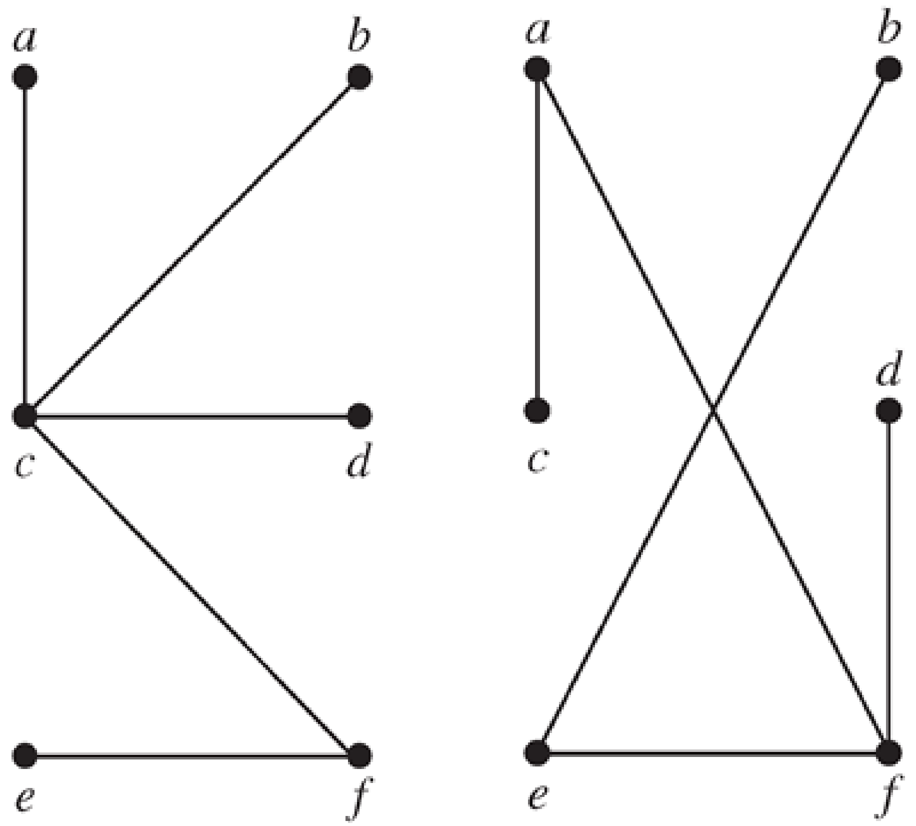
- **Definition** A *tree* is a connected undirected graph with no simple circuits.

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



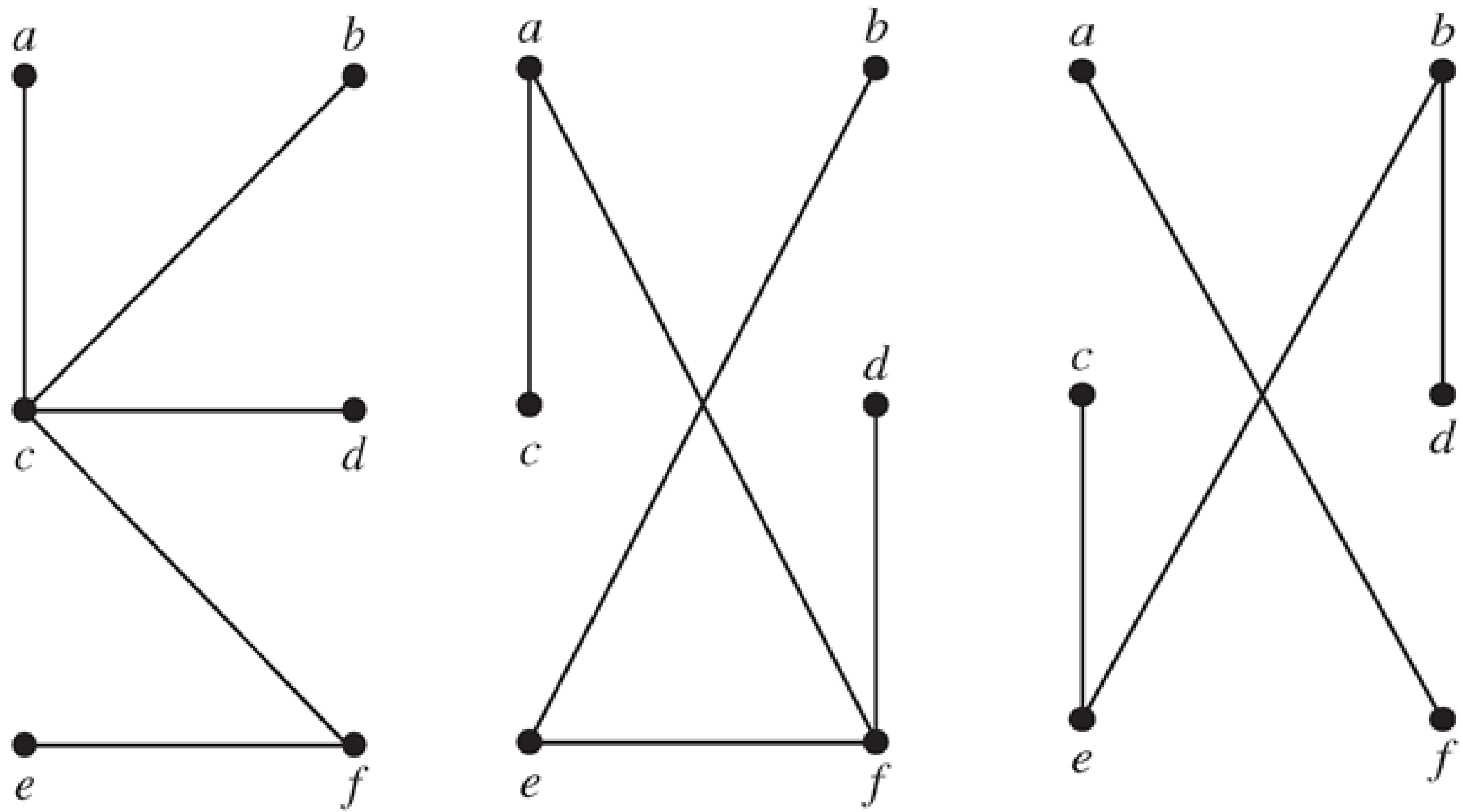
Trees

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



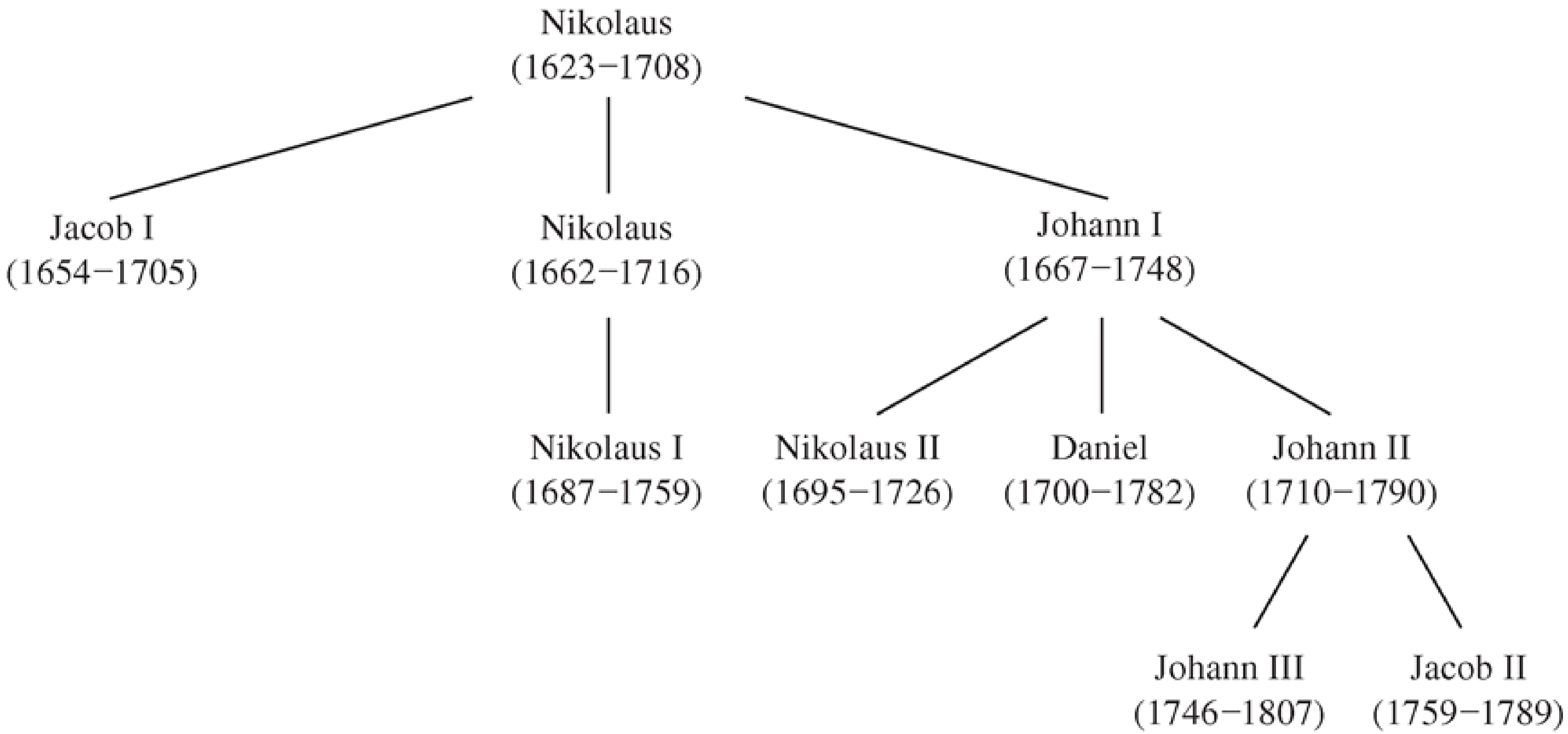
Trees

- **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

- **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

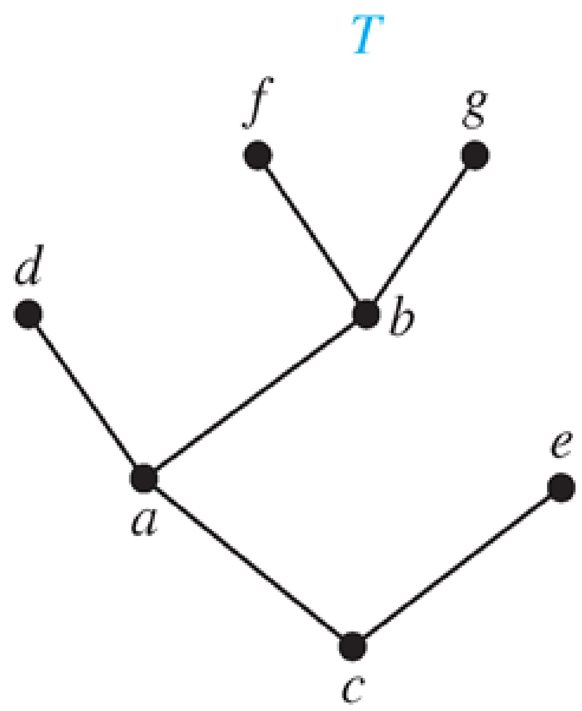
  Two properties of tree: connected, no circuit

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.

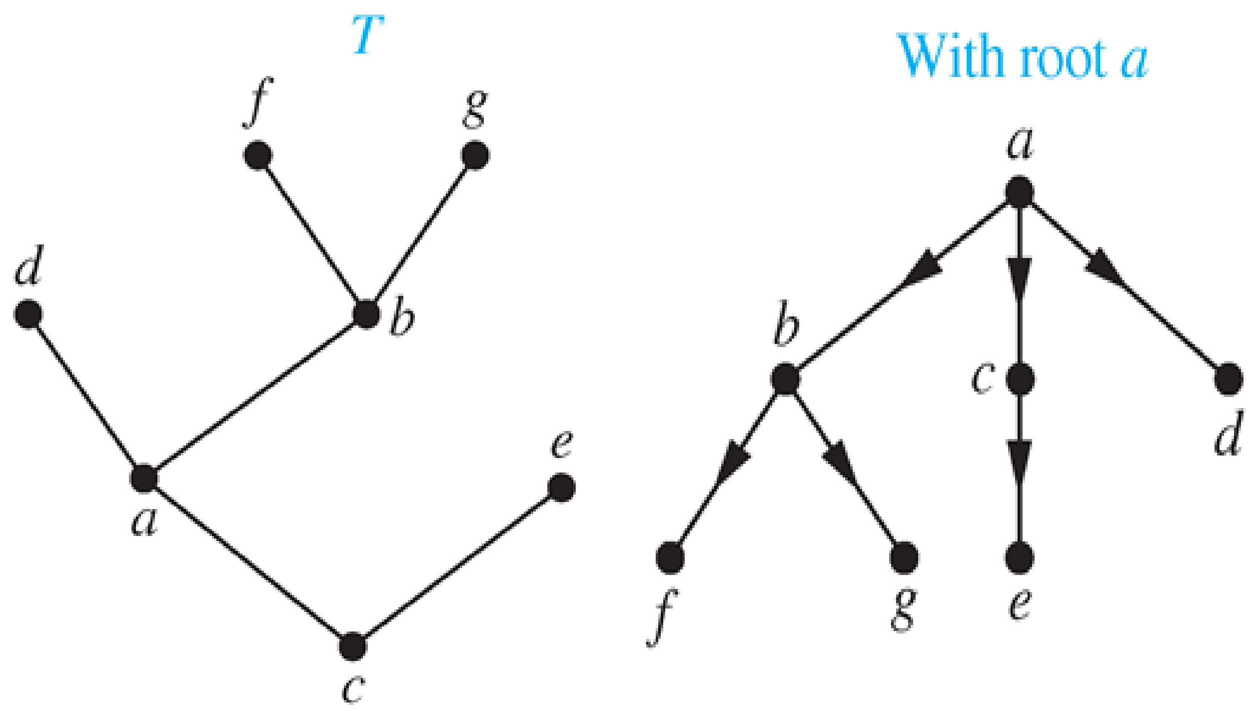
Rooted Trees

- **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.

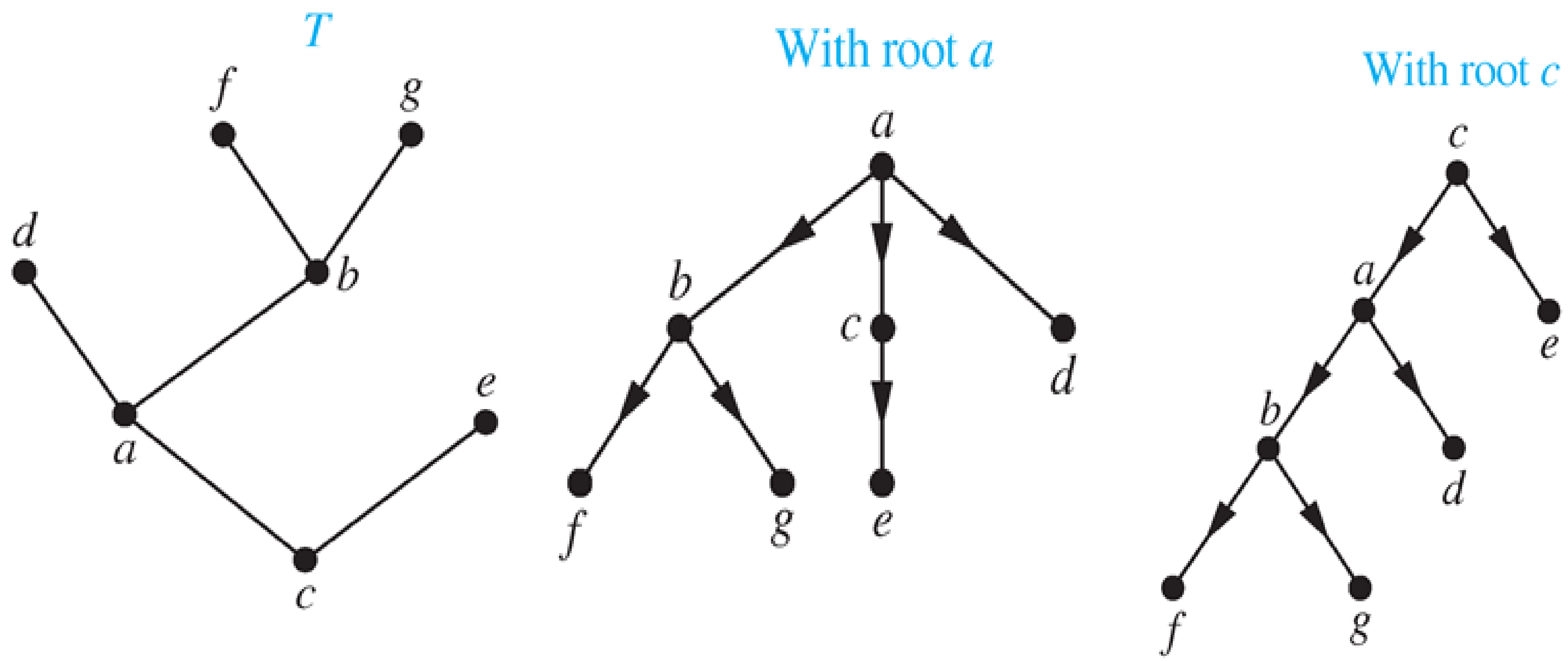


Rooted Trees

- **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

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

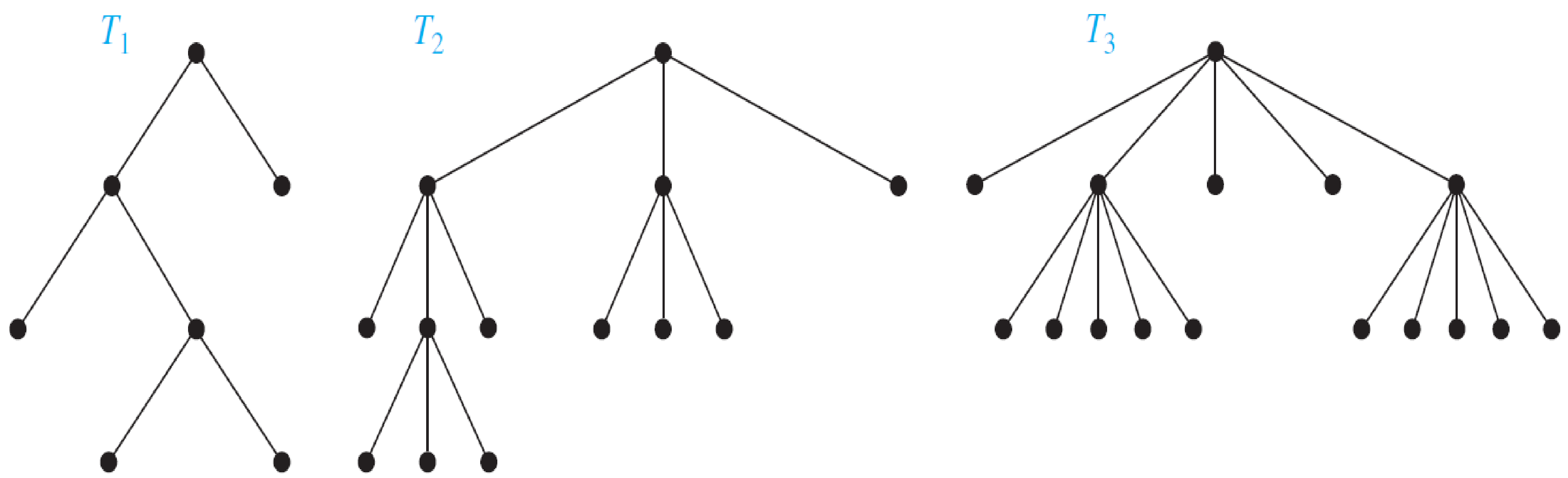
- **Definition** A rooted tree is called an *m-ary tree* if every internal vertex has no more than m children. The tree is called a *full m-ary tree* if every internal vertex has exactly m children. In particular, an m-ary tree with $m = 2$ is called a *binary tree*.

  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*

Full 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*.



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

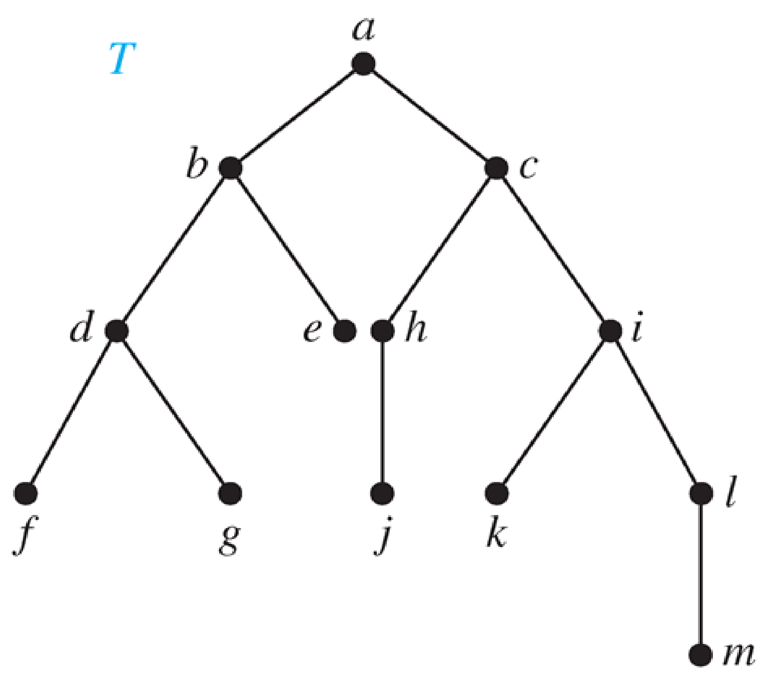
  Proof

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

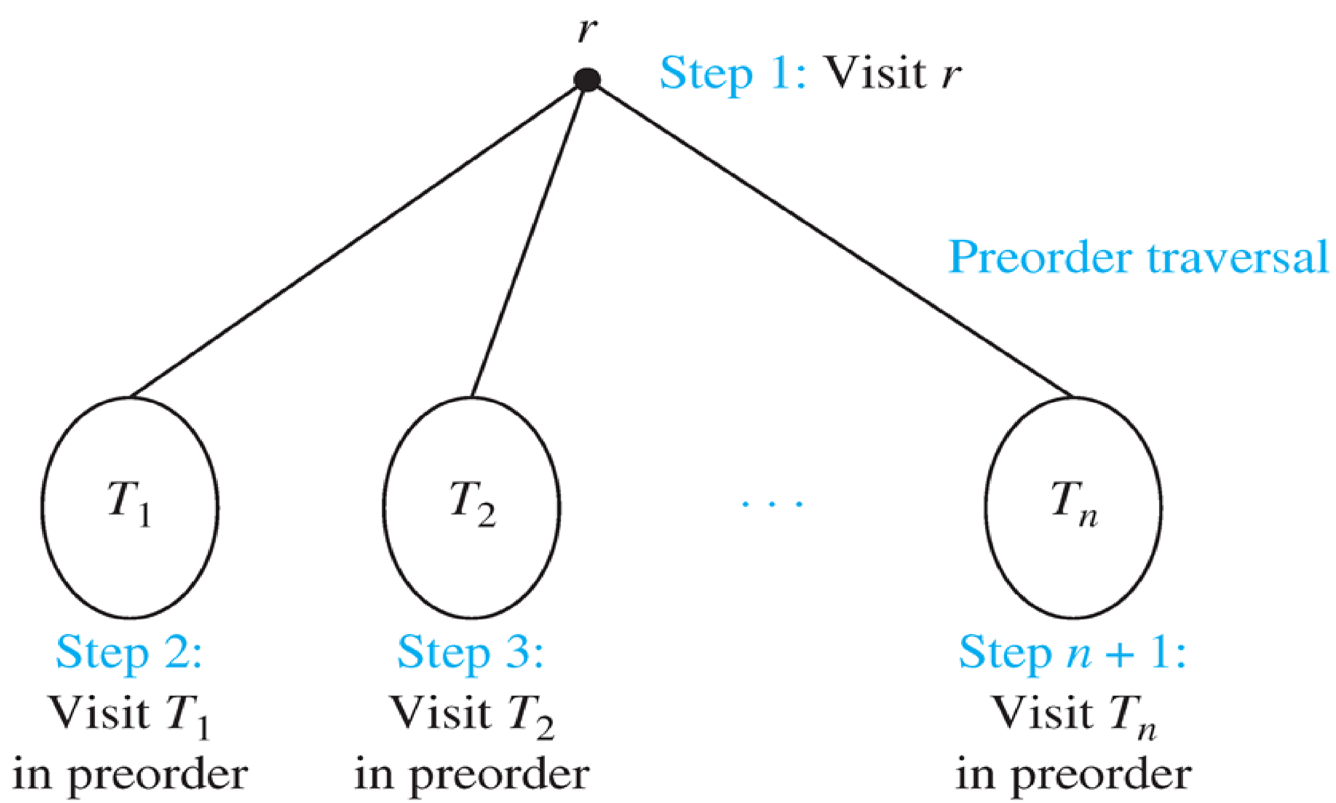
- The procedures for systematically visiting every vertex of an ordered tree are called *traversals*.

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

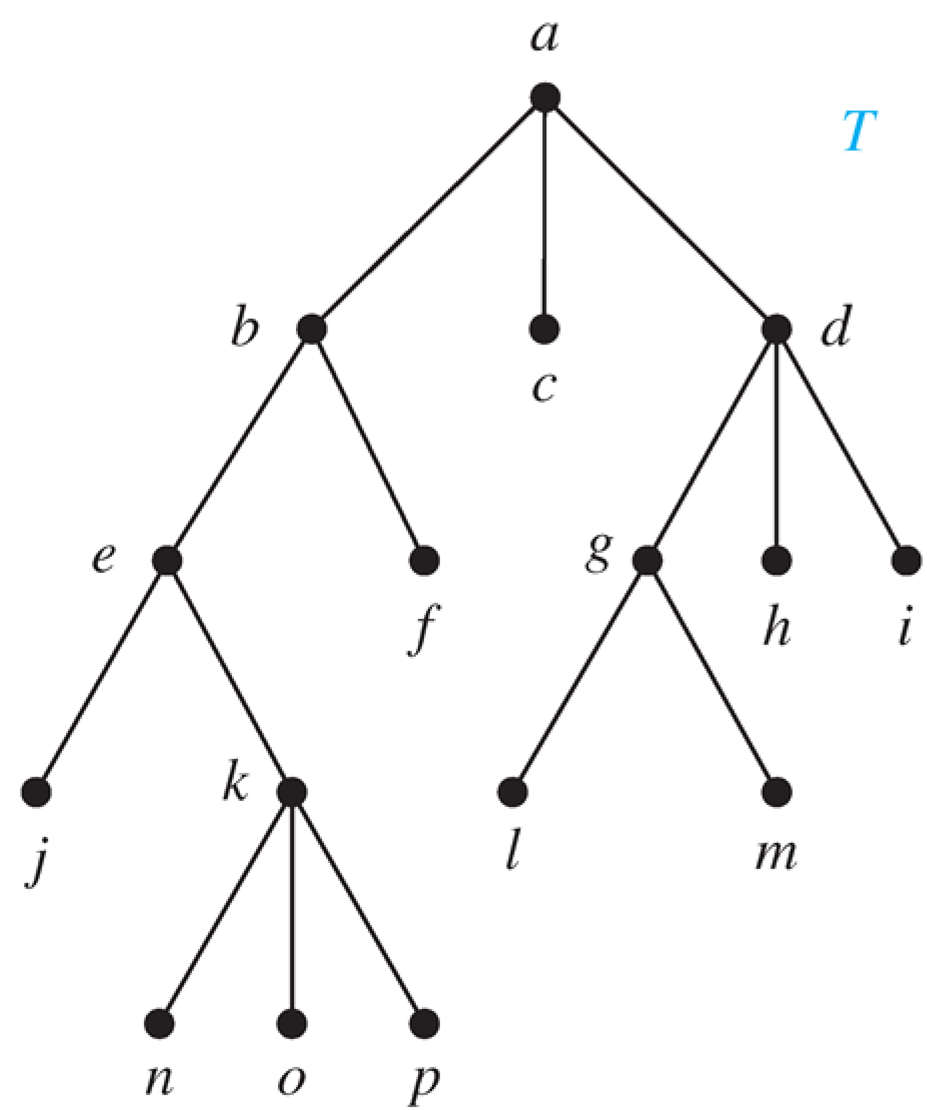
Preorder Traversal

- **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.



- **Example**



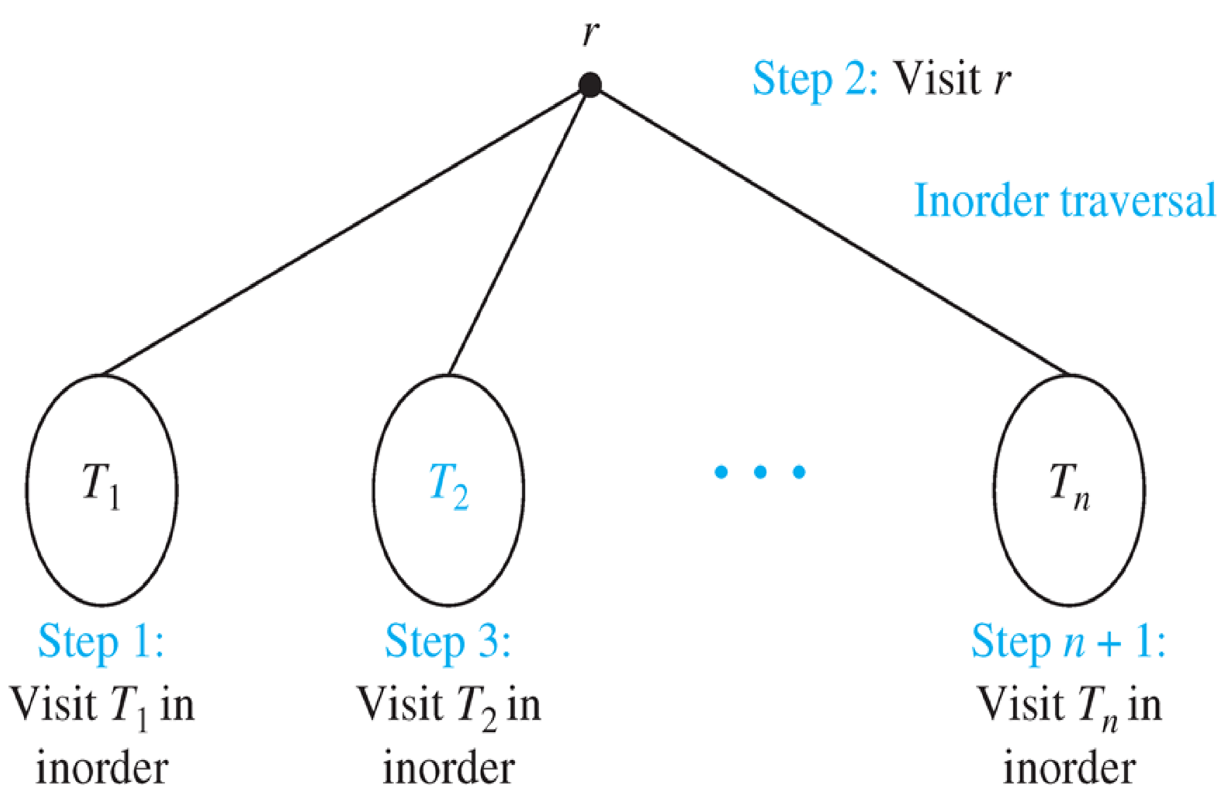
Preorder Traversal

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

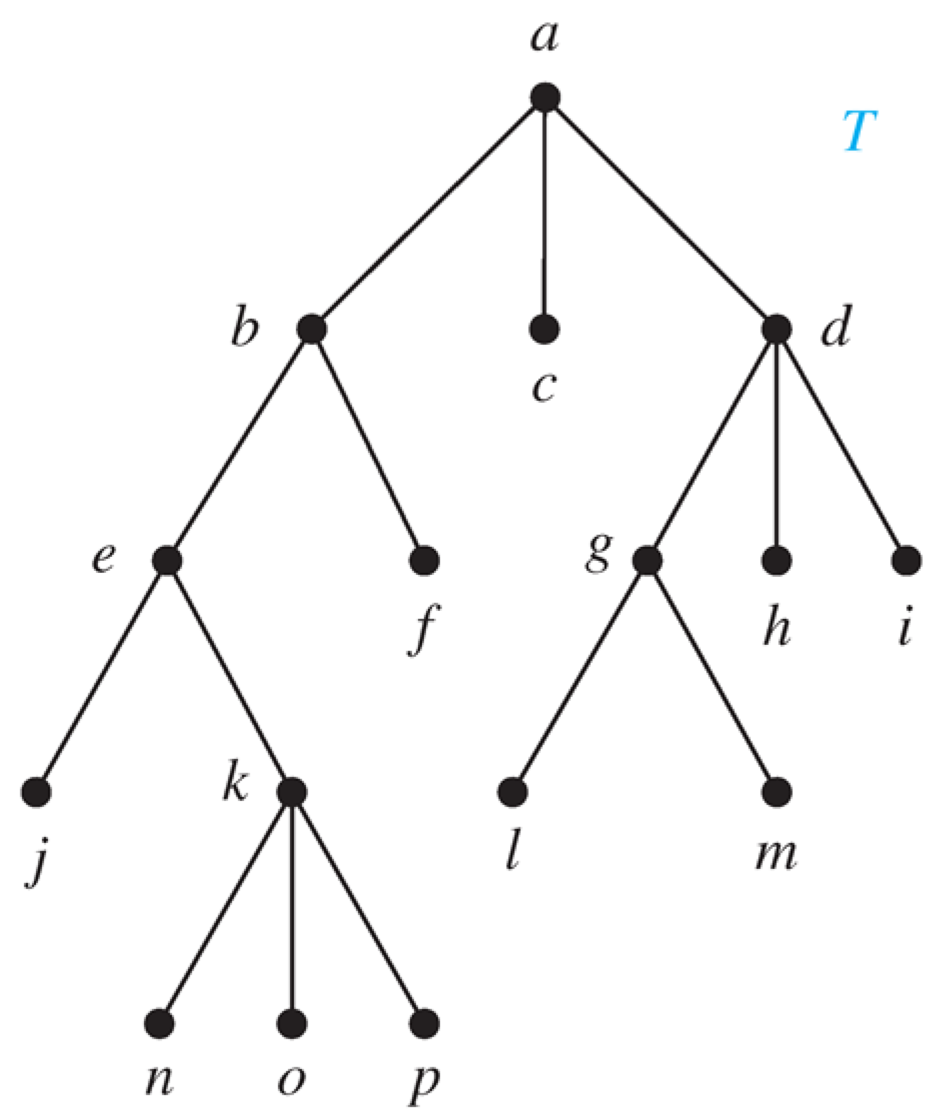
- **Definition** Let T be an ordered rooted tree with root r. If T consists only of r, then r is the *inorder traversal* of T. Otherwise, suppose that $T_1, T_2, \ldots, T_n$ are the subtrees of r from left to right in T. The *inorder traversal* begins by traversing T_1 in inorder, then visiting r, and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

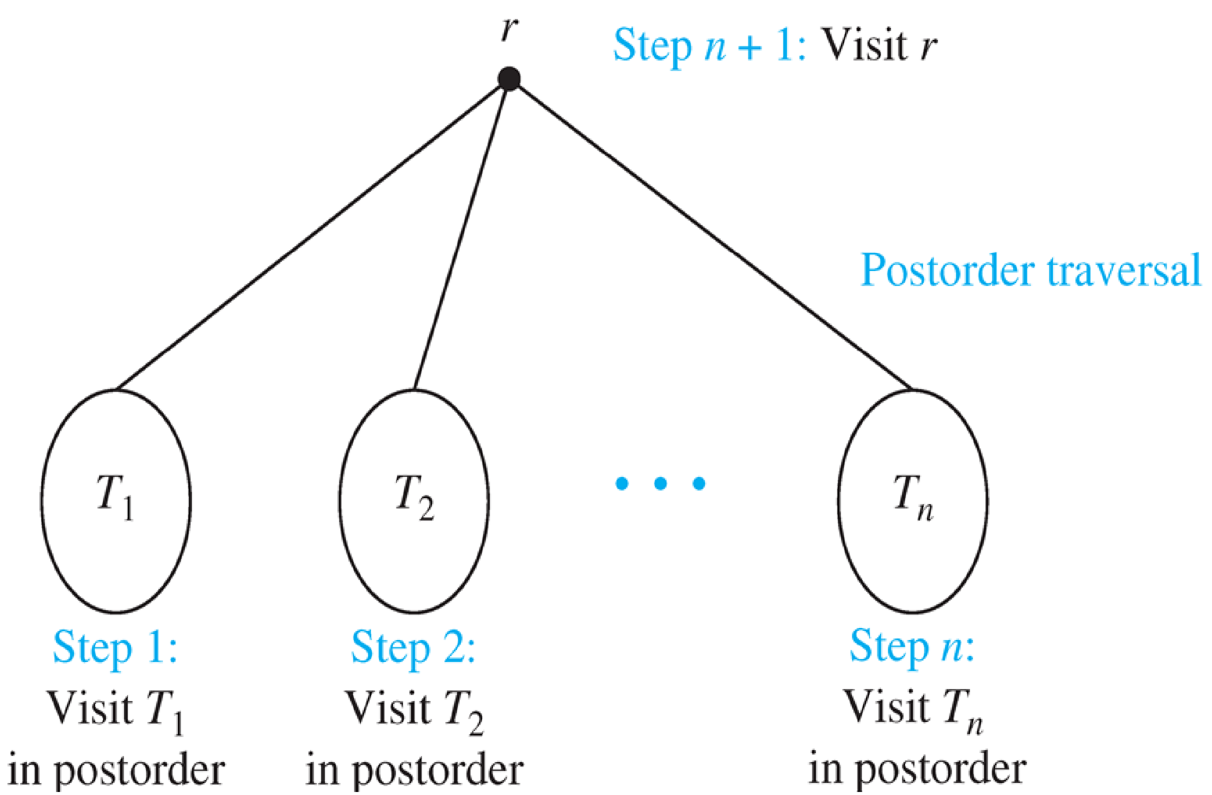
procedure *inorder* (T: ordered rooted tree)
r := root of T
if r is a leaf **then** list r
else
    l := first child of r from left to right
    $T(l)$:= subtree with l as its root
    inorder$(T(l))$
    list(r)
    for each child c of r from left to right
        $T(c)$:= subtree with c as root
  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

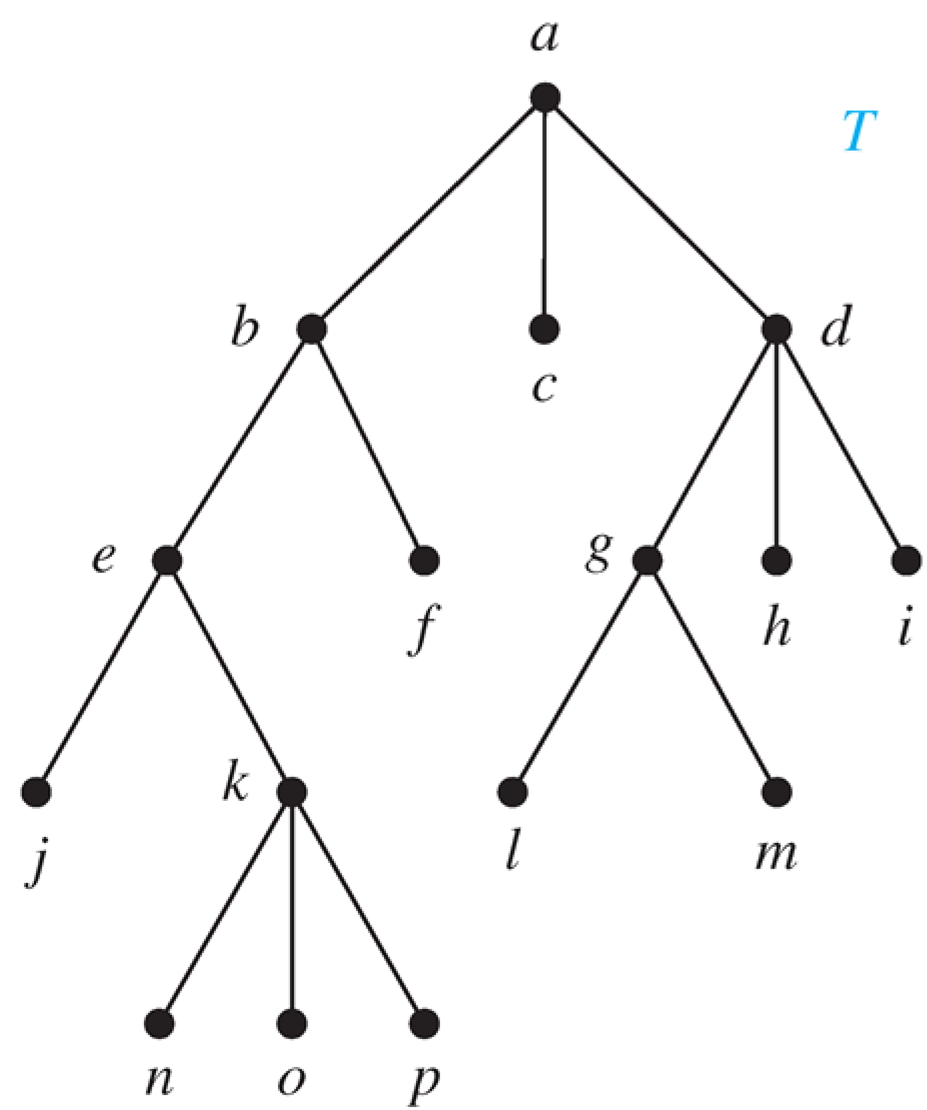
Postorder Traversal

- **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

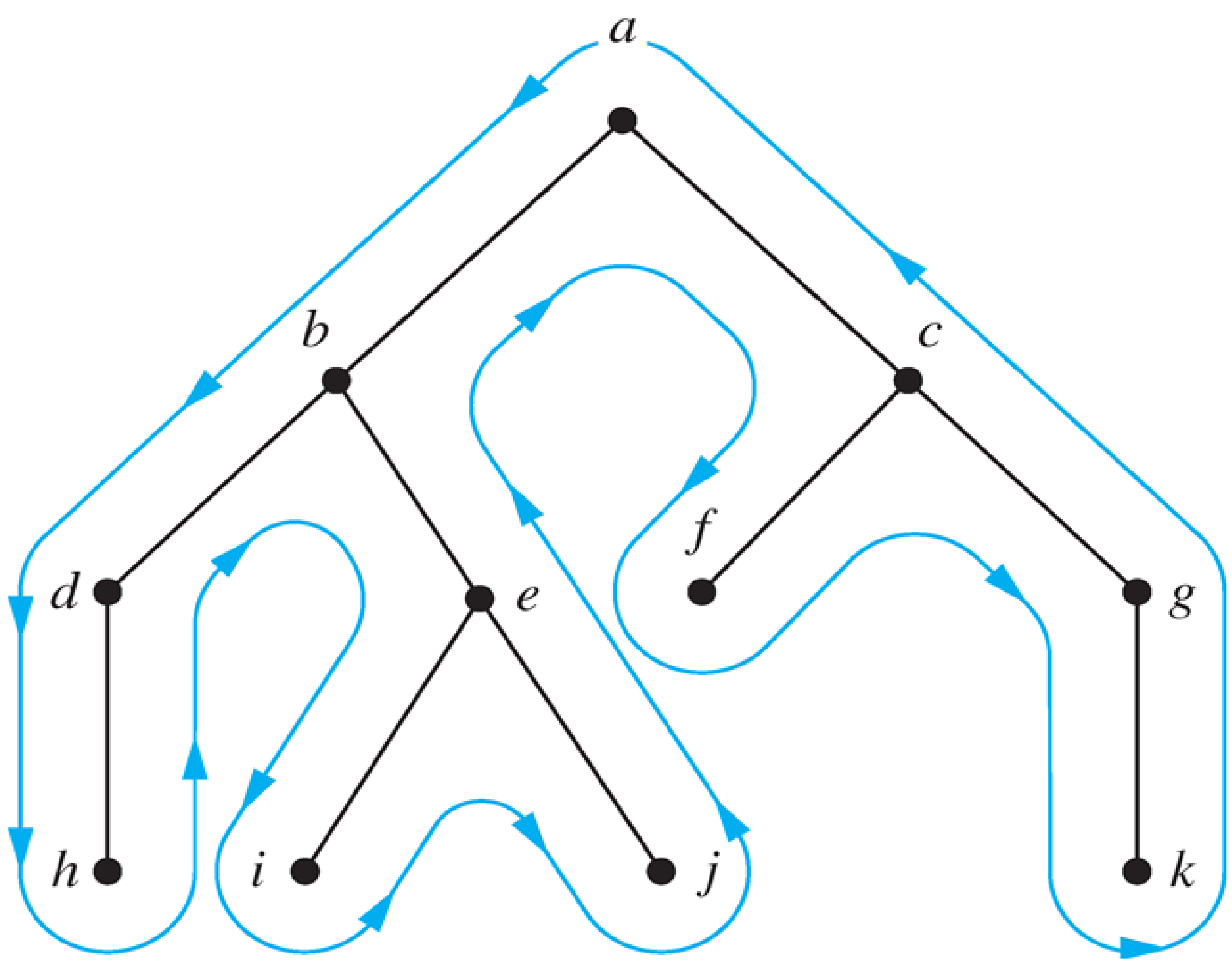
- **Example**



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

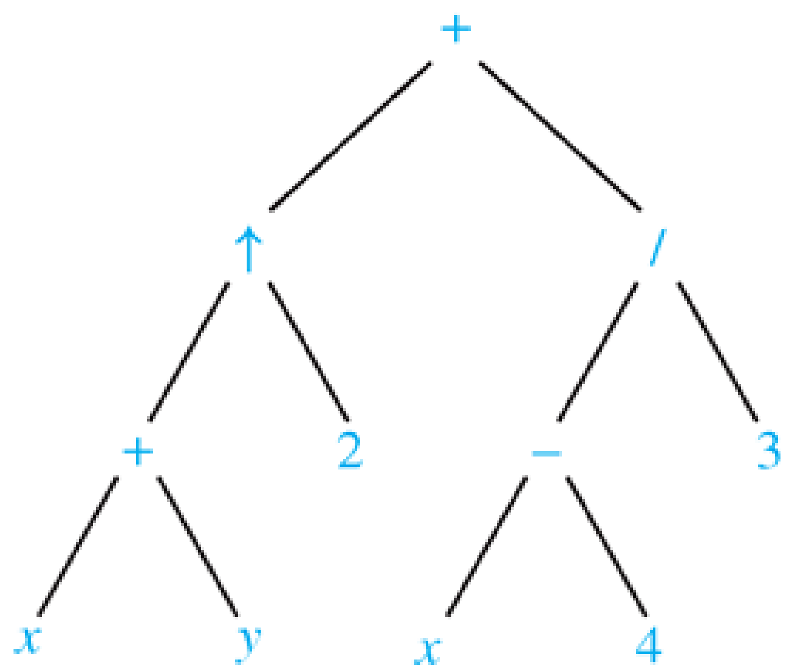
consider the expression $((x + y) \uparrow 2) + ((x - 4)/3)$

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

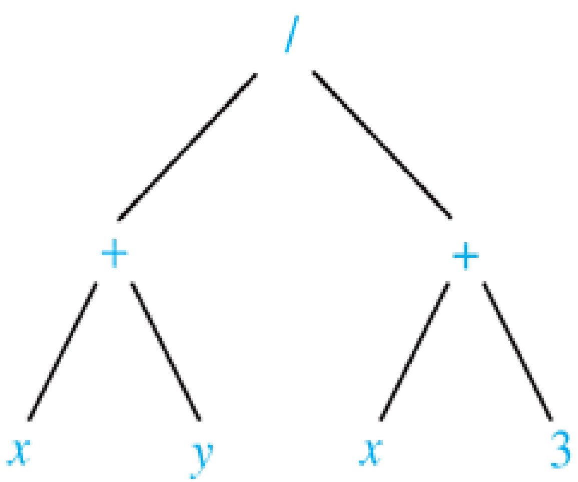
- An inorder traversal of the tree representing an expression produces the original expression when parentheses are included except for unary operation.

  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.

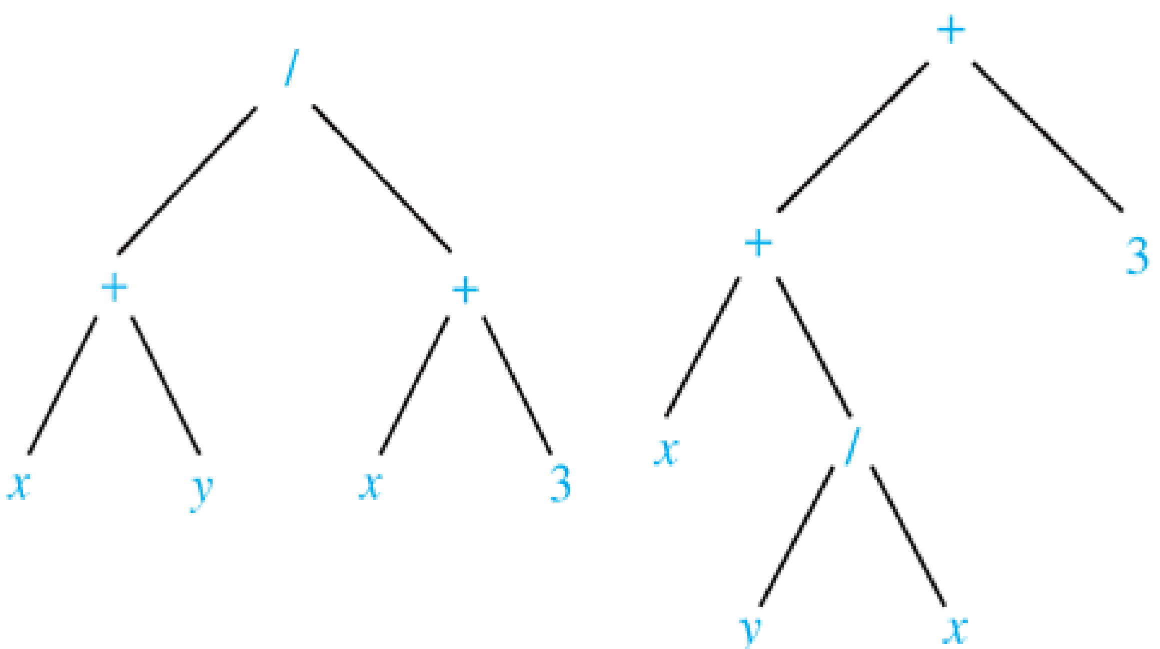
  Why parentheses are needed?



Infix Notation

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

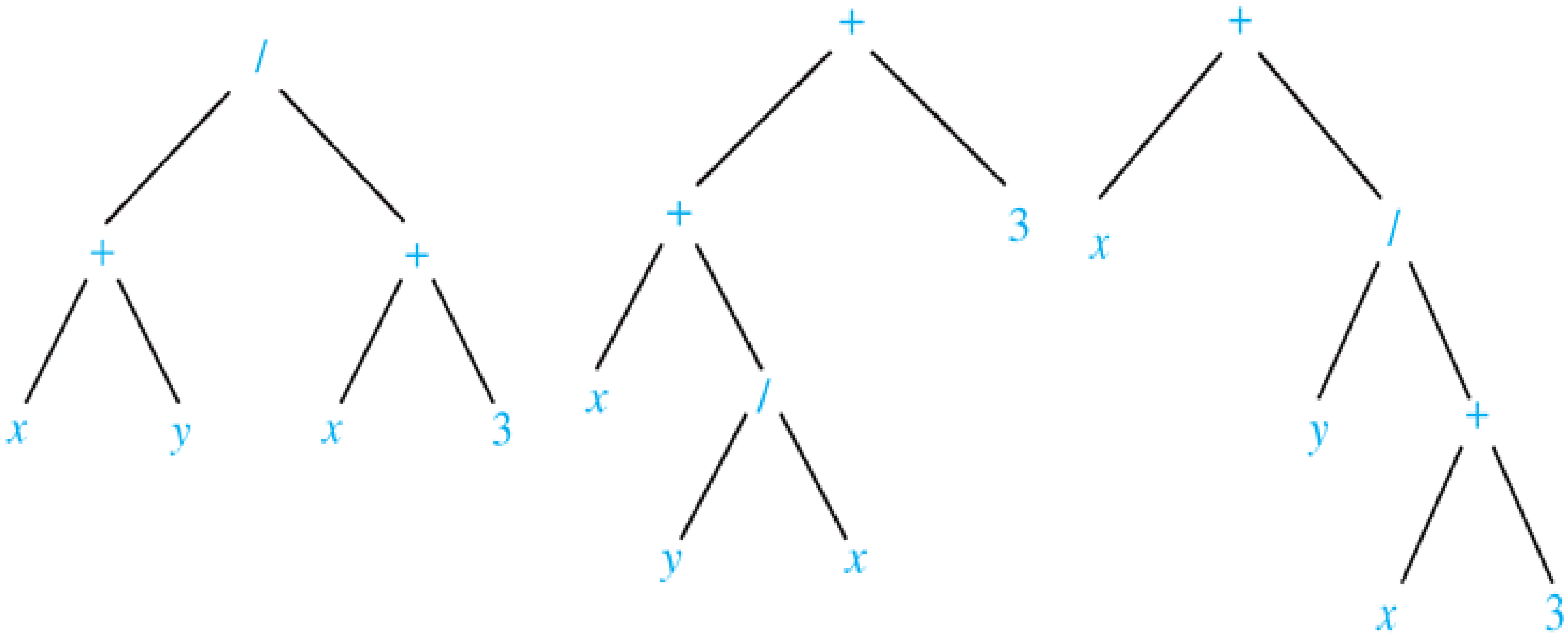
  Why parentheses are needed?



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**

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

- **Example**

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

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

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

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

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

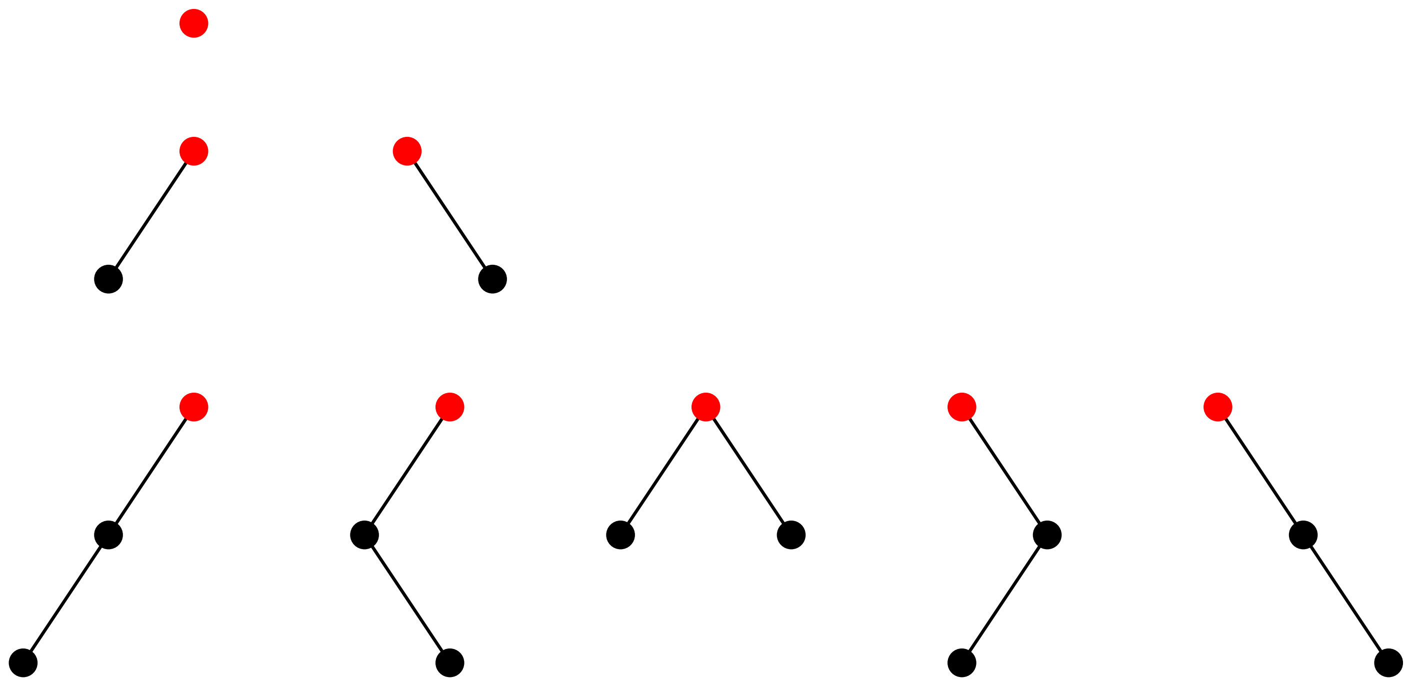
$$\underbrace{1\quad 3\quad +}_{1+3=4}$$

Catalan Numbers

- How many different binary tress are there with n vertices? We denote this number as C_n.

Catalan Numbers

- How many different binary tress are there with n vertices? We denote this number as C_n.

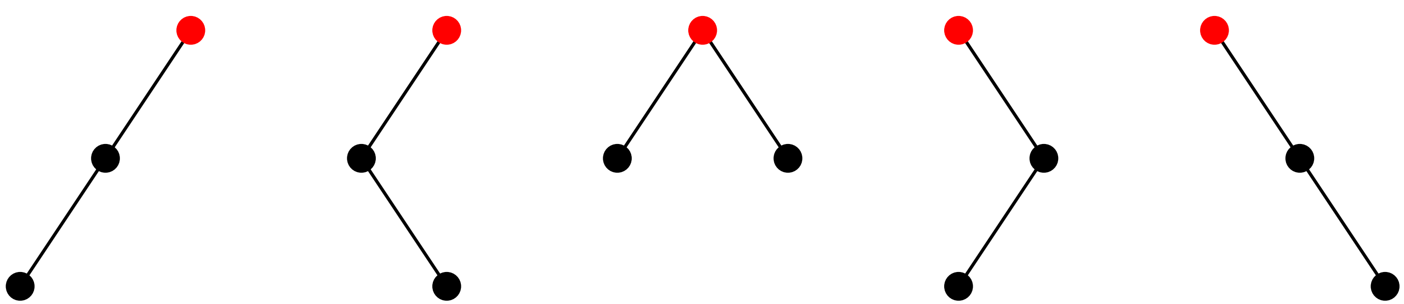


Catalan Numbers

- How many different binary tress are there with n vertices? We denote this number as C_n.

$$C_0 = C_1 = 1$$

$$C_2 = 2$$



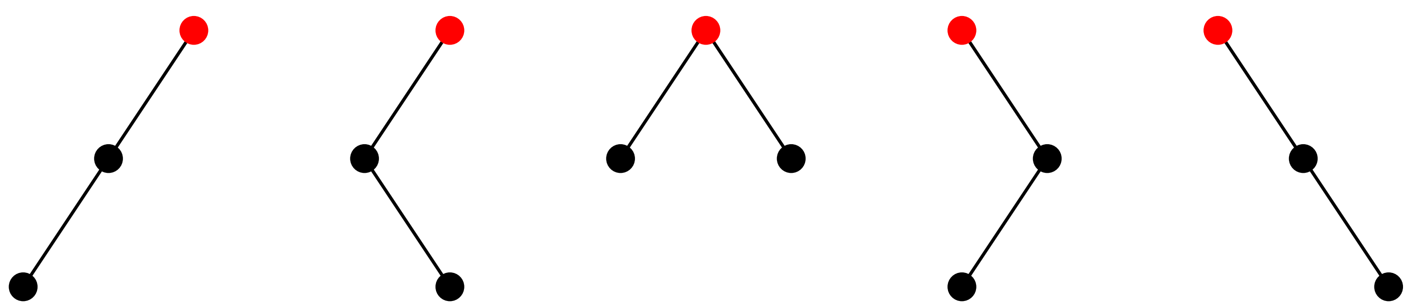
$$C_3 = 5$$

Catalan Numbers

- How many different binary tress are there with n vertices? We denote this number as C_n.

$$C_0 = C_1 = 1$$

$$C_2 = 2$$



$$C_3 = 5$$

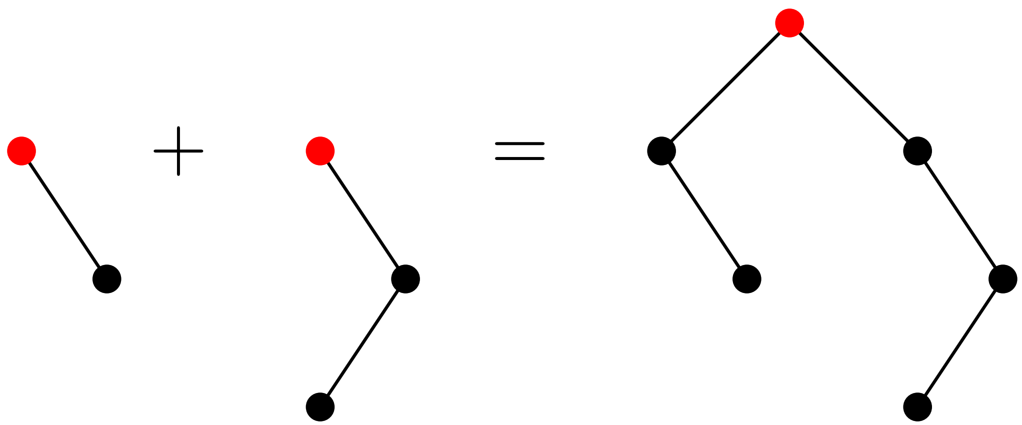
How to find a formula for C_n?

Catalan Numbers

- We first give an important *observation* on the recursive relation.

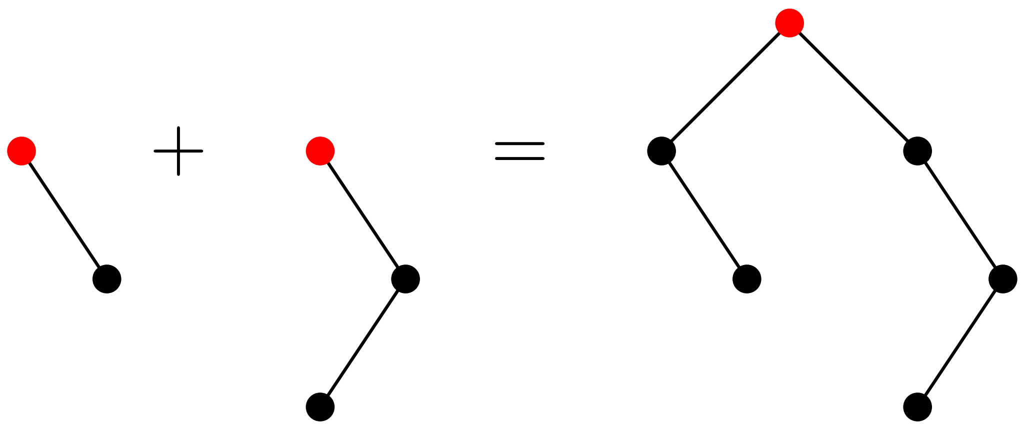
Catalan Numbers

- We first give an important *observation* on the recursive relation.
  Any nonempty rooted binary tree
  $=$ two smaller binary trees (possibly empty) $+$ one extra root



Catalan Numbers

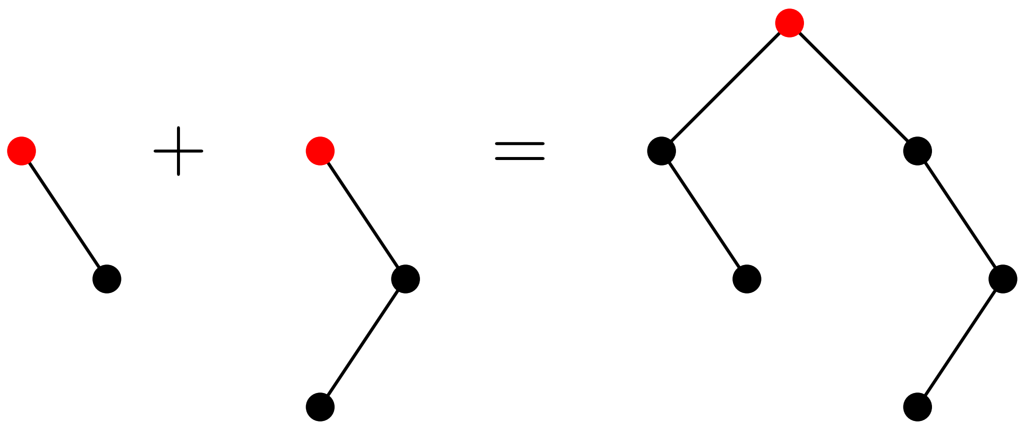
- We first give an important *observation* on the recursive relation.
  Any nonempty rooted binary tree
  = two smaller binary trees (possibly empty) + one extra root



We have $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$

Catalan Numbers

- We first give an important *observation* on the recursive relation.
  Any nonempty rooted binary tree
  $=$ two smaller binary trees (possibly empty) $+$ one extra root



We have $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$

For example, $C_3 = C_0 C_2 + C_1 C_1 + C_2 C_0 = 1 * 2 + 1 * 1 + 2 * 1 = 5.$

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

Catalan Numbers: Using Generating Functions

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

  The coefficient of x^n in f^2 is: $[x^n]_{f^2} = \sum_{i=0}^{n} C_i C_{n-i}$,

  since the following is the sum of all possible terms of x^n

  $$C_0 \cdot C_n x^n + C_1 x \cdot C_{n-1} x^{n-1} + \cdots + C_n x^n \cdot C_0.$$

Catalan Numbers: Using Generating Functions

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

  The coefficient of x^n in f^2 is: $[x^n]_{f^2} = \sum_{i=0}^{n} C_i C_{n-i}$,

  since the following is the sum of all possible terms of x^n

  $$C_0 \cdot C_n x^n + C_1 x \cdot C_{n-1} x^{n-1} + \cdots + C_n x^n \cdot C_0.$$

  Since $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$, we have

  $$f^2 = \sum_{n=0}^{\infty} C_{n+1} x^n.$$

Catalan Numbers: Using Generating Functions

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

  The coefficient of x^n in f^2 is: $[x^n]_{f^2} = \sum_{i=0}^{n} C_i C_{n-i}$,

  since the following is the sum of all possible terms of x^n

  $$C_0 \cdot C_n x^n + C_1 x \cdot C_{n-1} x^{n-1} + \cdots + C_n x^n \cdot C_0.$$

  Since $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$, we have

  $$f^2 = \sum_{n=0}^{\infty} C_{n+1} x^n.$$

  Then we have $xf^2 + 1 = f$, which gives $f = \frac{1 \pm \sqrt{1-4x}}{2x}$ for $x \neq 0$.

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

  The coefficient of x^n in f^2 is: $[x^n]_{f^2} = \sum_{i=0}^{n} C_i C_{n-i}$,

  since the following is the sum of all possible terms of x^n

  $$C_0 \cdot C_n x^n + C_1 x \cdot C_{n-1} x^{n-1} + \cdots + C_n x^n \cdot C_0.$$

  Since $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$, we have

  $$f^2 = \sum_{n=0}^{\infty} C_{n+1} x^n.$$

  Then we have $xf^2 + 1 = f$, which gives $f = \frac{1 \pm \sqrt{1-4x}}{2x}$ for $x \neq 0$.

  When $x \to 0$, $\frac{1+\sqrt{1-4x}}{2x} \to \infty$ and $\frac{1-\sqrt{1-4x}}{2x} \to 1$.

  Since $f(0) = 1 = C_0$, we know $f = \frac{1-\sqrt{1-4x}}{2x}$.

Catalan Numbers: Using Generating Functions

- Let $f(x) = \sum_{i=0}^{\infty} C_i x^i$. We now consider f^2.

  The coefficient of x^n in f^2 is: $[x^n]_{f^2} = \sum_{i=0}^{n} C_i C_{n-i}$,

  since the following is the sum of all possible terms of x^n

  $$C_0 \cdot C_n x^n + C_1 x \cdot C_{n-1} x^{n-1} + \cdots + C_n x^n \cdot C_0.$$

  Since $C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$, we have

  $$f^2 = \sum_{n=0}^{\infty} C_{n+1} x^n.$$

  Then we have $xf^2 + 1 = f$, which gives $f = \frac{1 \pm \sqrt{1-4x}}{2x}$ for $x \neq 0$.

  When $x \to 0$, $\frac{1+\sqrt{1-4x}}{2x} \to \infty$ and $\frac{1-\sqrt{1-4x}}{2x} \to 1$.

  Since $f(0) = 1 = C_0$, we know $f = \frac{1-\sqrt{1-4x}}{2x}$.

  C_n – the coefficient of x^n in the expansion of f.

Catalan Numbers

- $f = \frac{1-\sqrt{1-4x}}{2x}$, by the extended Binomial Theorem,

$$\sqrt{1-4x} = (1+(-4x))^{1/2} = \sum_{n=0}^{\infty} \binom{1/2}{n} (-4x)^n.$$

- $f = \frac{1-\sqrt{1-4x}}{2x}$, by the extended Binomial Theorem,

$$\sqrt{1-4x} = (1+(-4x))^{1/2} = \sum_{n=0}^{\infty} \binom{1/2}{n} (-4x)^n.$$

$$\frac{1-\sqrt{1-4x}}{2x} = \sum_{n=1}^{\infty} -\frac{1}{2}\binom{1/2}{n}(-4)^n x^{n-1} = \sum_{n=0}^{\infty} -\frac{1}{2}\binom{1/2}{n+1}(-4)^{n+1} x^n.$$

where $\binom{1/2}{n} = \frac{\frac{1}{2}(-\frac{1}{2})(-\frac{3}{2})\cdots(-\frac{2n-3}{2})}{n!}$.

Catalan Numbers

- $f = \frac{1-\sqrt{1-4x}}{2x}$, by the extended Binomial Theorem,

$$\sqrt{1-4x} = (1+(-4x))^{1/2} = \sum_{n=0}^{\infty} \binom{1/2}{n} (-4x)^n.$$

$$\frac{1-\sqrt{1-4x}}{2x} = \sum_{n=1}^{\infty} -\frac{1}{2}\binom{1/2}{n}(-4)^n x^{n-1} = \sum_{n=0}^{\infty} -\frac{1}{2}\binom{1/2}{n+1}(-4)^{n+1}x^n.$$

where $\binom{1/2}{n} = \frac{\frac{1}{2}(-\frac{1}{2})(-\frac{3}{2})\cdots(-\frac{2n-3}{2})}{n!}$.

Then we have $C_n = \frac{1}{n+1}\binom{2n}{n}$.

This is called the n-th *Catalan number*.

- **Theorem** The number of sequences $a_1, \ldots, a_{2n}$ of $2n$ terms that can be formed using exactly n $+1$'s and exactly n -1's whose partial sums are always nonnegative, i.e., $a_1 + a_2 + \cdots + a_k \geq 0$ for any $1 \leq k \leq 2n$, equals the n-th Catalan number C_n.

Catalan Numbers: Related Problems

- **Theorem** The number of sequences $a_1, \ldots, a_{2n}$ of $2n$ terms that can be formed using exactly n $+1$'s and exactly n -1's whose partial sums are always nonnegative, i.e., $a_1 + a_2 + \cdots + a_k \geq 0$ for any $1 \leq k \leq 2n$, equals the n-th Catalan number C_n.

- Many others! For example, the number of *full binary trees* with $2n+1$ vertices...

Catalan Numbers: Related Problems

- **Theorem** The number of sequences $a_1, \ldots, a_{2n}$ of $2n$ terms that can be formed using exactly n $+1$'s and exactly n -1's whose partial sums are always nonnegative, i.e., $a_1 + a_2 + \cdots + a_k \geq 0$ for any $1 \leq k \leq 2n$, equals the n-th Catalan number C_n.

- Many others! For example, the number of *full binary trees* with $2n+1$ vertices...

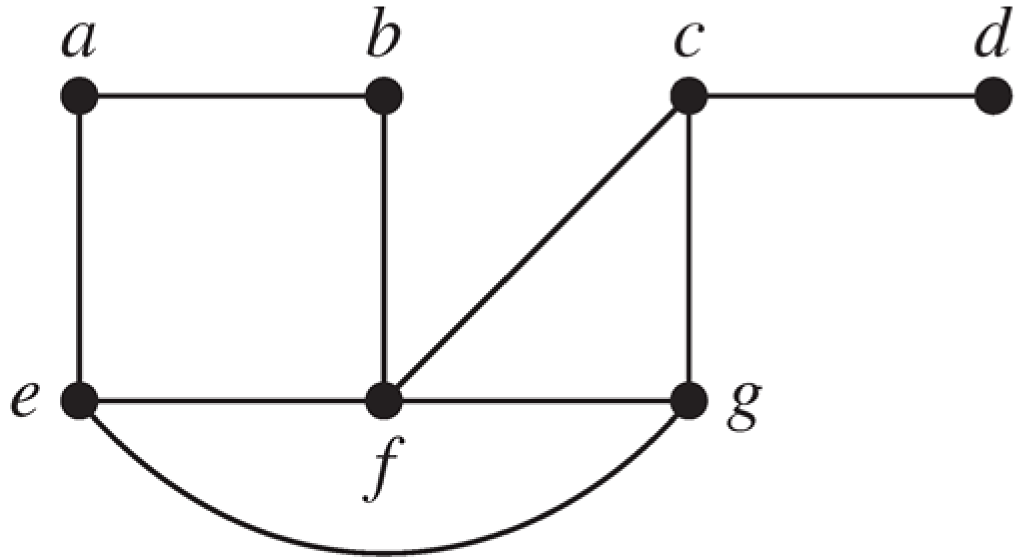
R. Stanley, *Catalan Numbers*, Cambridge University Press, 2015. Includes 214 combinatorial interpretations of C_n, and 68 additional problems!

Spanning Trees

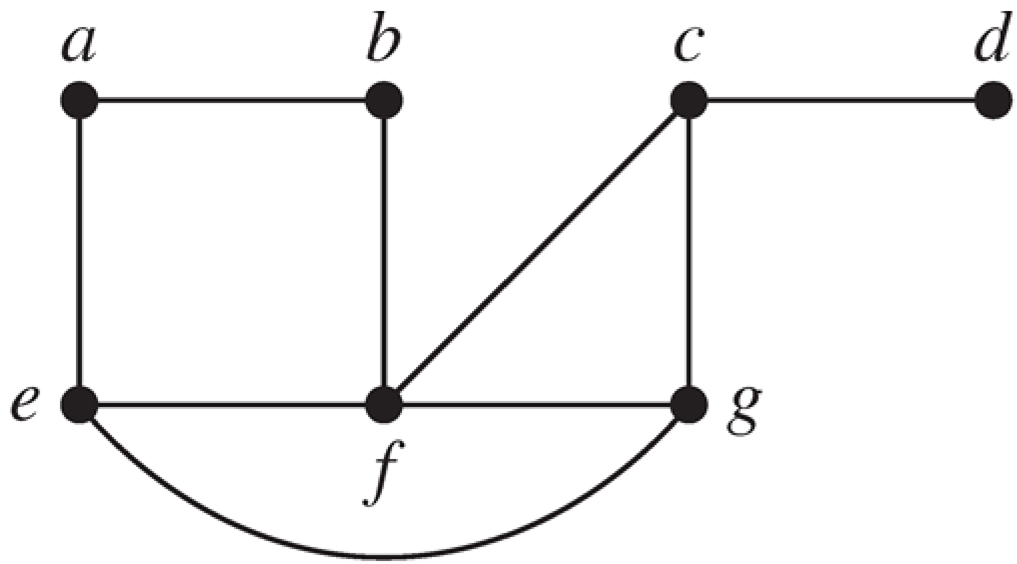
- **Definition** Let G be a simple graph. A *spanning tree* of G is a subgraph of G that is a tree containing every vertex of G.

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.



- **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.

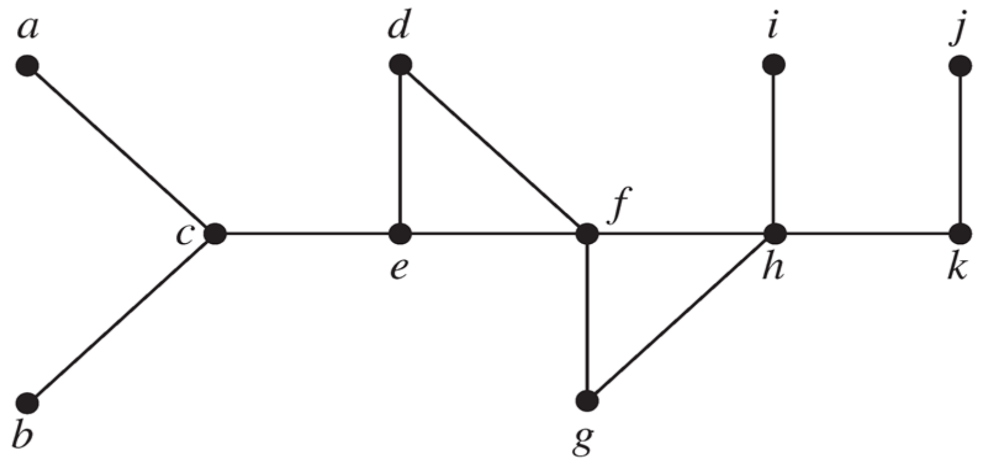
  ◇ Form a path by successively adding vertices and edges.
  Continue adding to this path as long as possible.

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

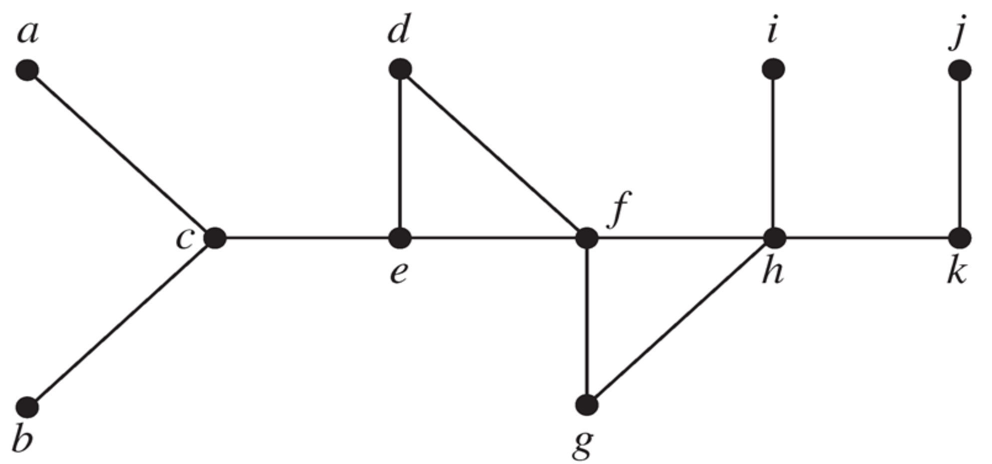
  ◇ Otherwise, move back to some vertex to repeat this procedure (*backtracking*)

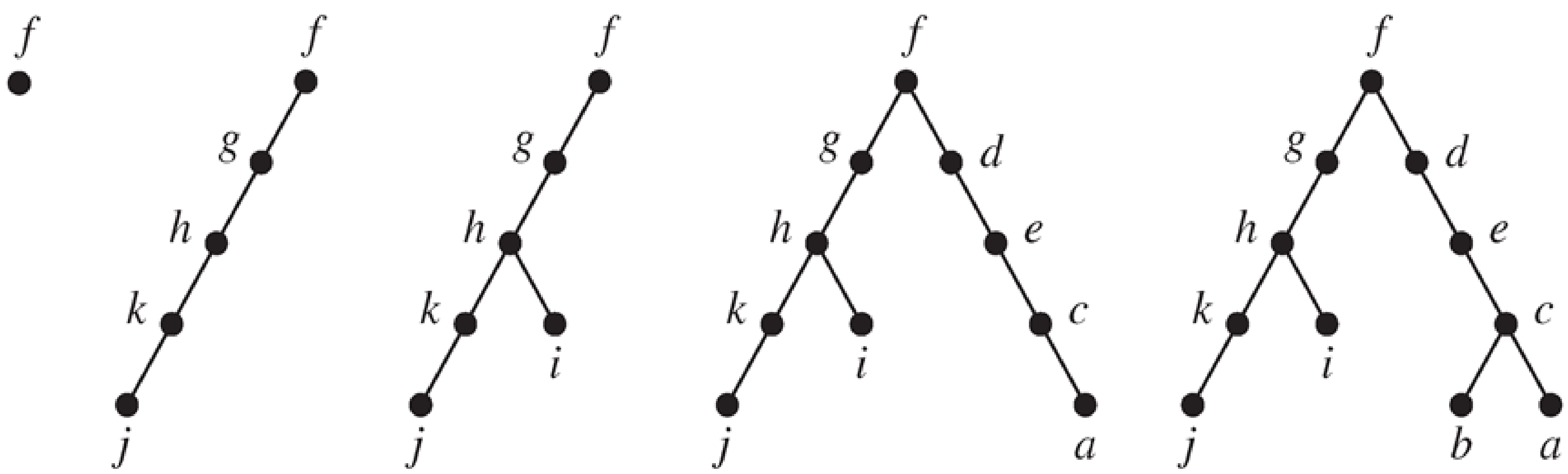
Depth-First Search

- **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.

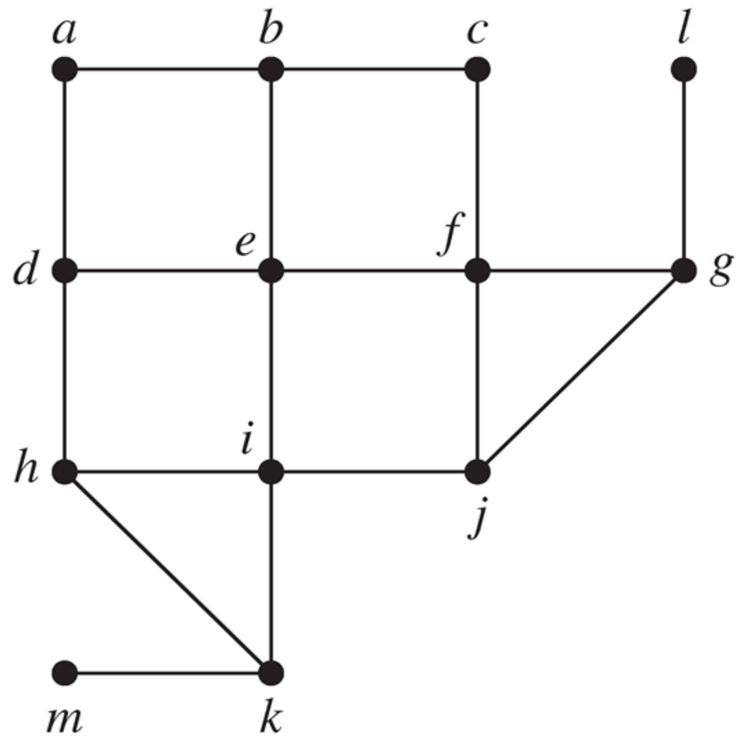
  ◇ Form a path by adding all edges incident to this vertex and the other endpoint of each of these edges

  ◇ 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.

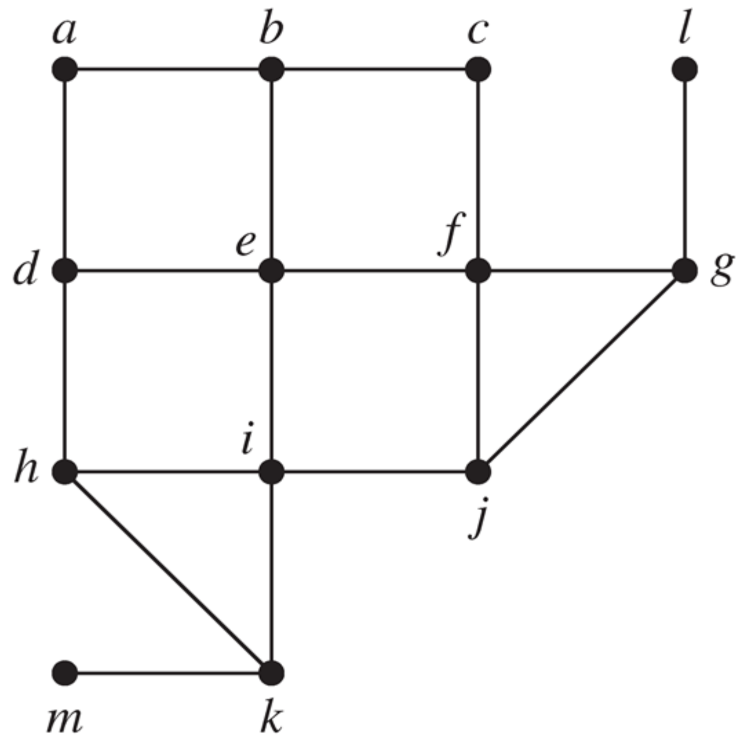
Breadth-First Search

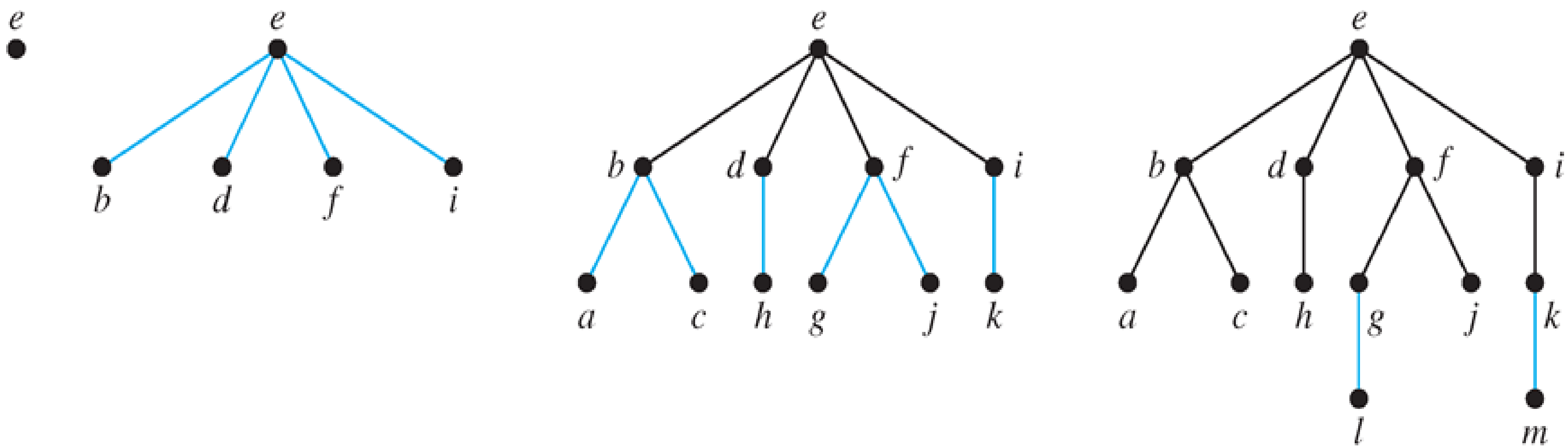
- **Example**



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, ...

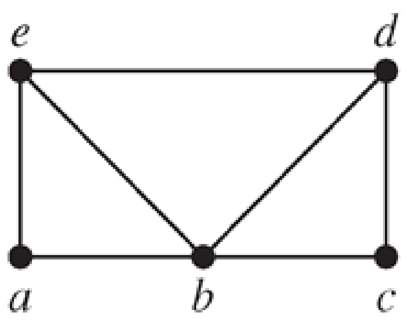
Applications of DFS, BFS

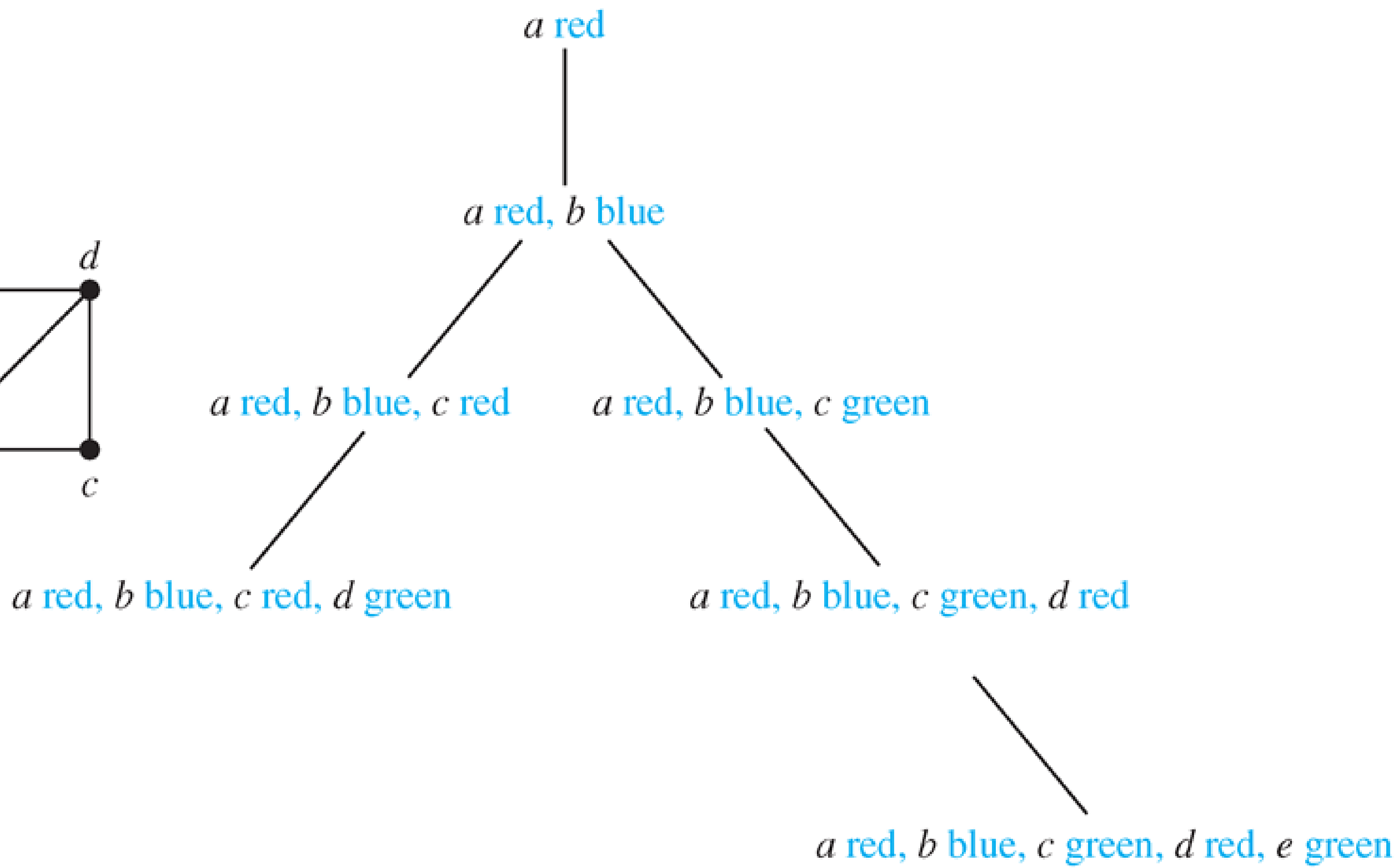
- find paths, circuits, connected components, cut vertices, ...

  find shortest paths, determine whether bipartite, ...

  graph coloring, sums of subsets, ...

Applications of DFS, BFS



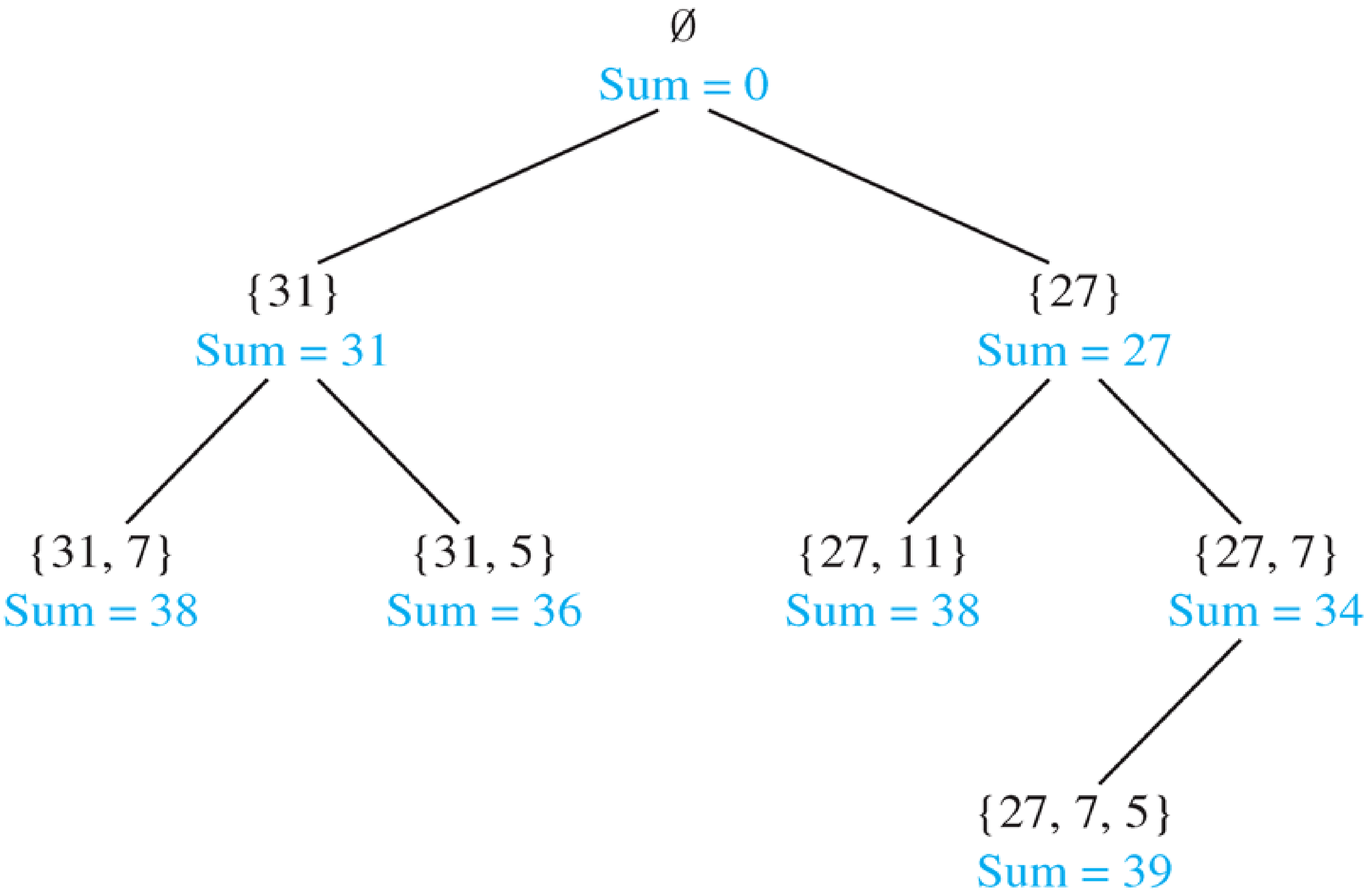


Applications of DFS, BFS

- find 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.

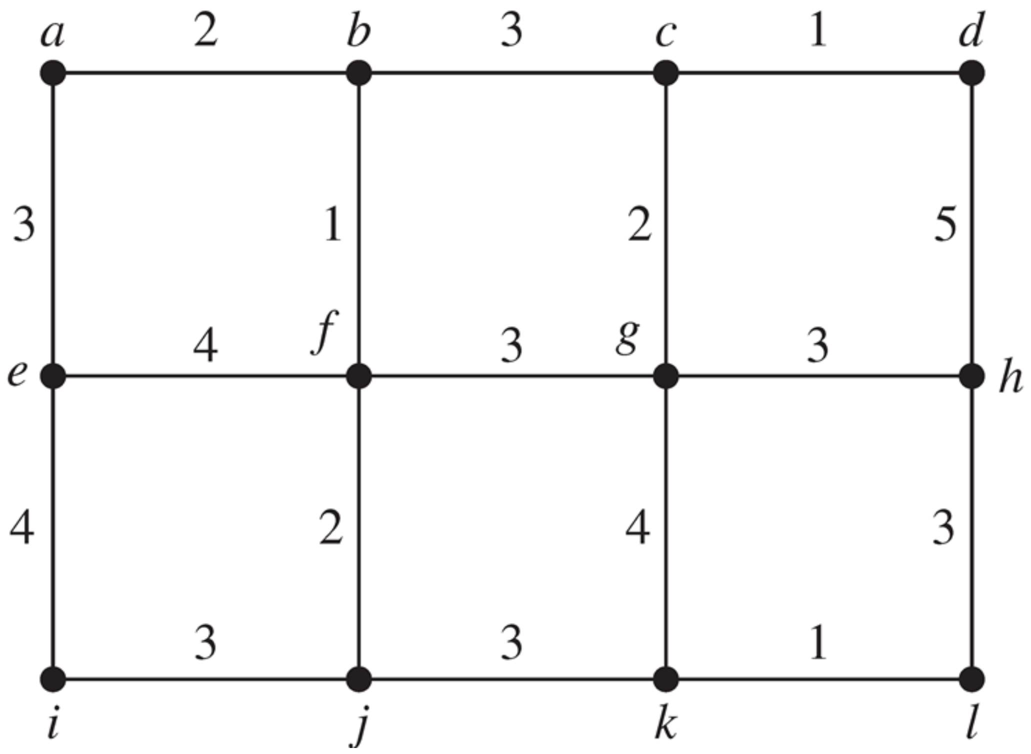
procedure *Prim*(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for $i := 1$ **to** $n - 2$
  e := an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T
  $T := T$ with e added
return T {T is a minimum spanning tree of G}

We can maintain a *heap* of all the edges with at least one endpoint in T, and in each iteration, we do Extract-Mins until we see an edge that has one endpoint in T and one endpoint not in T.

time complexity: $e \log v$

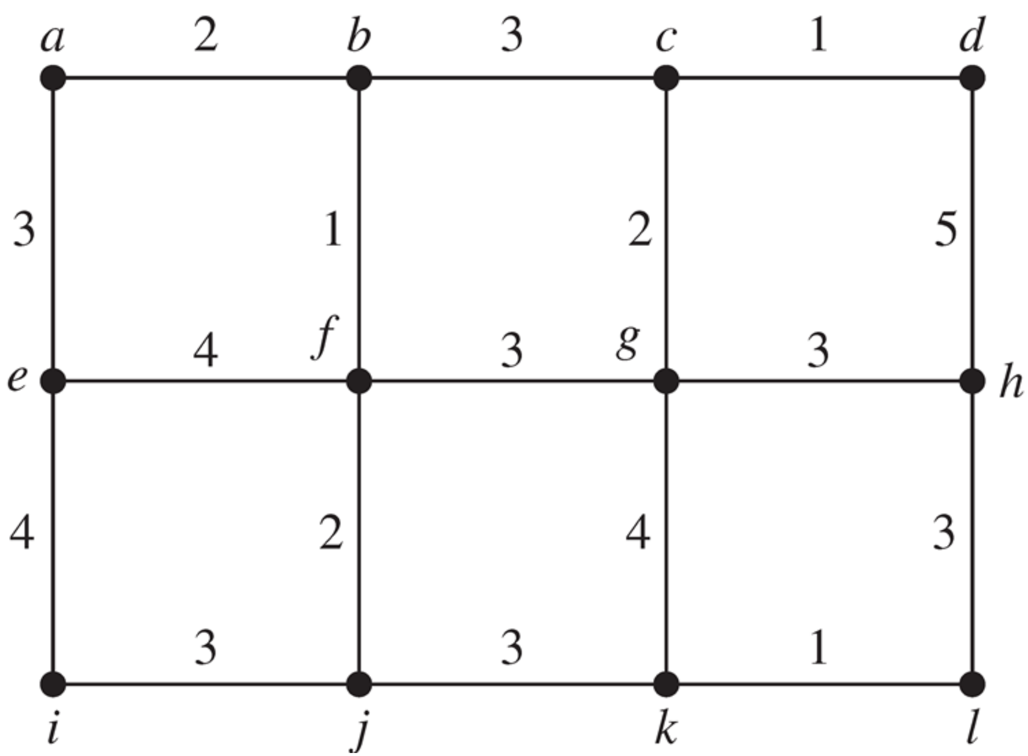
Prim's Algorithm

- **Example**



Prim's Algorithm

- **Example**



Choice	Edge	Weight
1	$\{b, f\}$	1
2	$\{a, b\}$	2
3	$\{f, j\}$	2
4	$\{a, e\}$	3
5	$\{i, j\}$	3
6	$\{f, g\}$	3
7	$\{c, g\}$	2
8	$\{c, d\}$	1
9	$\{g, h\}$	3
10	$\{h, l\}$	3
11	$\{k, l\}$	1
	Total:	24

Prim's Algorithm: Correctness

- **Proof** by *induction*.

Prim's Algorithm: Correctness

- **Proof** by *induction*.

  i.h.: After each iteration, the tree T is a subgraph of some MST M. This is trivially true for the basic step, since intially T has only one vertex and no edges.

Prim's Algorithm: Correctness

- **Proof** by *induction*.

  i.h.: After each iteration, the tree T is a subgraph of some MST M. This is trivially true for the basic step, since intially T has only one vertex and no edges.

  Prim's algorithm tells us to add the edge e. We need to prove that $T \cup \{e\}$ is also a subtree of some MST.

Prim's Algorithm: Correctness

- **Proof** by *induction*.

  i.h.: After each iteration, the tree T is a subgraph of some MST M.
  This is trivially true for the basic step, since intially T has only one vertex and no edges.
  Prim's algorithm tells us to add the edge e. We need to prove that $T \cup \{e\}$ is also a subtree of some MST.

  If $e \in M$, this is clearly true. Since by i.h., $T \subset M$ and $e \in M$, then $T \cup \{e\} \subset M$. Now suppose that $e \notin M$.

Prim's Algorithm: Correctness

- **Proof** by *induction*.

  i.h.: After each iteration, the tree T is a subgraph of some MST M. This is trivially true for the basic step, since intially T has only one vertex and no edges.
  Prim's algorithm tells us to add the edge e. We need to prove that $T \cup \{e\}$ is also a subtree of some MST.

  If $e \in M$, this is clearly true. Since by i.h., $T \subset M$ and $e \in M$, then $T \cup \{e\} \subset M$. Now suppose that $e \notin M$.

  If we add e to M, then a circuit will be created in M. Since e has one endpoint in T and the other endpoint not in T, there has to be some other edge e' in this circuit that has exactly one endpoint in T.

Prim's Algorithm: Correctness

- **Proof** by *induction*.

  i.h.: After each iteration, the tree T is a subgraph of some MST M. This is trivially true for the basic step, since intially T has only one vertex and no edges.

  Prim's algorithm tells us to add the edge e. We need to prove that $T \cup \{e\}$ is also a subtree of some MST.

  If $e \in M$, this is clearly true. Since by i.h., $T \subset M$ and $e \in M$, then $T \cup \{e\} \subset M$. Now suppose that $e \notin M$.

  If we add e to M, then a circuit will be created in M. Since e has one endpoint in T and the other endpoint not in T, there has to be some other edge e' in this circuit that has exactly one endpoint in T.

  Since Prim's algorithm has chosen to add e, we have $w(e) \leq w(e')$. So if we add e to M and remove e' from M, we will have a new tree M' whose total weight $\leq$ that of M, and $T \cup \{e\} \subset M'$.

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}

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: $e \log e$ *Union-Find*

Kruscal's Algorithm

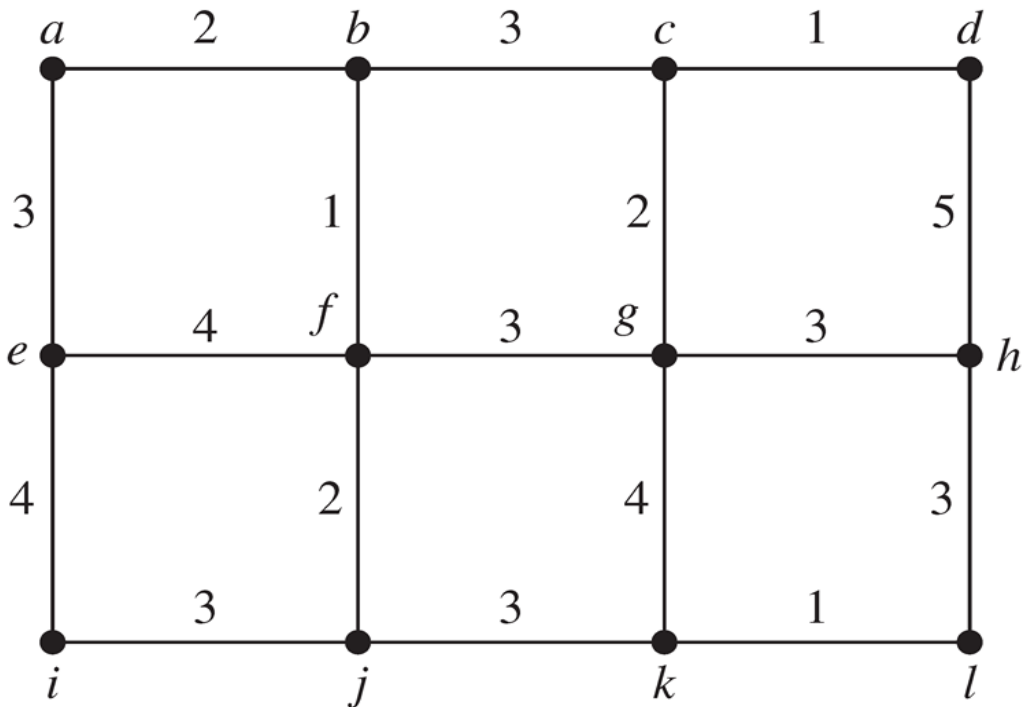
ALGORITHM 2 Kruskal's Algorithm.

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}

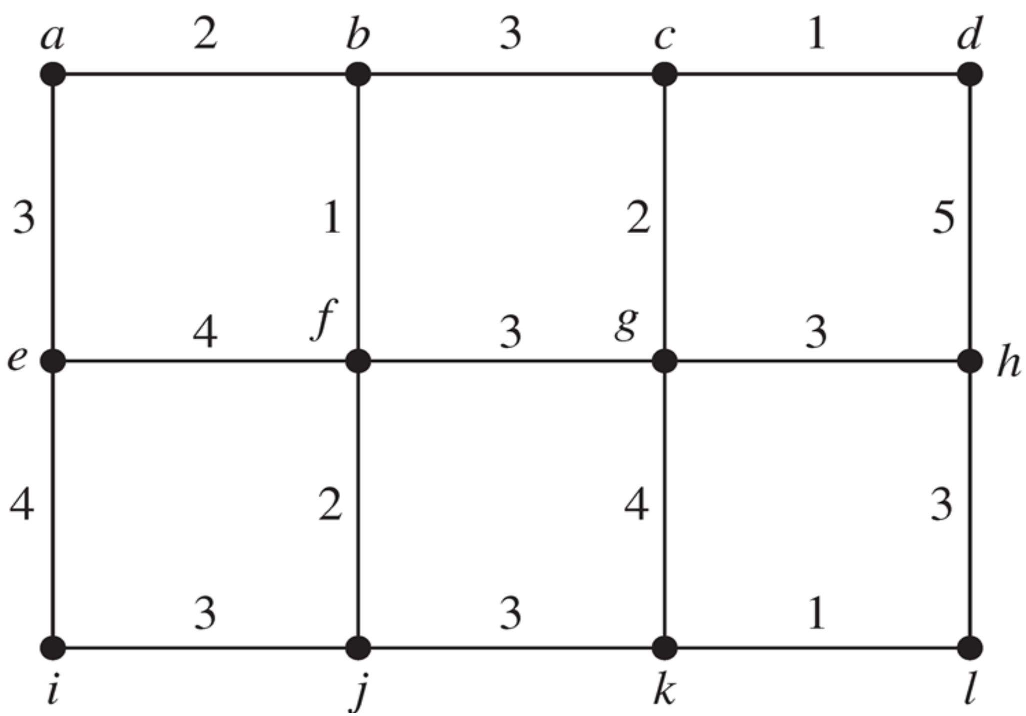
time complexity: $e \log e$ *Union-Find*

see *CLRS* / *Algorithm Design*, J. Kleinberg, E. Tardos

- **Example**



Example



Choice	Edge	Weight
1	$\{c, d\}$	1
2	$\{k, l\}$	1
3	$\{b, f\}$	1
4	$\{c, g\}$	2
5	$\{a, b\}$	2
6	$\{f, j\}$	2
7	$\{b, c\}$	3
8	$\{j, k\}$	3
9	$\{g, h\}$	3
10	$\{i, j\}$	3
11	$\{a, e\}$	3
	Total:	24

Kruscal's Algorithm: Correctness

- **Proof** by *induction*. Similar to Prim's algorithm.

Kruscal's Algorithm: Correctness

- **Proof** by *induction*. Similar to Prim's algorithm.
- A unifying structure: Given a graph $G = (V, E)$, every subset $S \subseteq V$ defines a *cut* $(S, \bar{S})$. We consider the edge set between S and $\bar{S}$.

Kruscal's Algorithm: Correctness

- **Proof** by *induction*. Similar to Prim's algorithm.
- A unifying structure: Given a graph $G = (V, E)$, every subset $S \subseteq V$ defines a *cut* $(S, \bar{S})$. We consider the edge set between S and $\bar{S}$.

  Theorem Let $(S, \bar{S})$ be an arbitrary cut, and let e be an edge across the cut (one endpoint in S, the other in $\bar{S}$) that has the smallest weight of all edges cross the cut. Then there must be an MST T containing e.

  Theorem Let $(S, \bar{S})$ be an arbitrary cut, and let E' be the set of edges across the cut of minimum weight ($w(e) = w(e')$ for any two edges $e, e' \in E'$ and $w(e) < w(e')$ for any $e \in E'$ and $e' \notin E'$). Let T be an arbitrary MST. Then T must contain some edge in E'.

Next Lecture

- reduction, review ...