



Graph Theory

Chapter 8



Varying Applications (examples)

- Computer networks
- Distinguish between two chemical compounds with the same molecular formula but different structures
- Solve shortest path problems between cities
- Scheduling exams and assign channels to television stations



Topics Covered

- Definitions
- Types
- Terminology
- Representation
- Sub-graphs
- Connectivity
- Hamilton and Euler definitions
- Shortest Path
- Planar Graphs
- Graph Coloring



Definitions - Graph

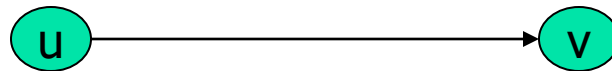
A generalization of the simple concept of a set of dots, links, edges or arcs.

Representation: Graph $G = (V, E)$ consists set of vertices denoted by V , or by $V(G)$ and set of edges E , or $E(G)$

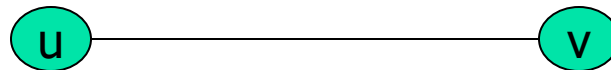


Definitions – Edge Type

Directed: Ordered pair of vertices. Represented as (u, v) directed from vertex u to v .

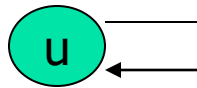


Undirected: Unordered pair of vertices. Represented as $\{u, v\}$. Disregards any sense of direction and treats both end vertices interchangeably.



Definitions – Edge Type

- **Loop:** A loop is an edge whose endpoints are equal i.e., an edge joining a vertex to it self is called a loop. Represented as $\{u, u\} = \{u\}$

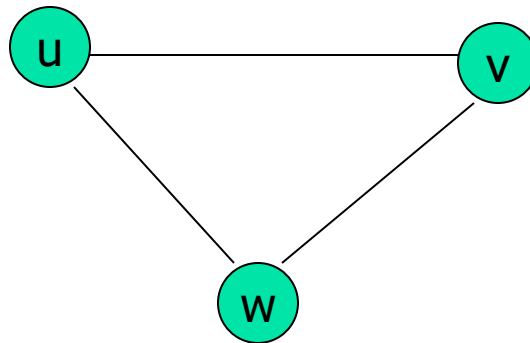


- **Multiple Edges:** Two or more edges joining the same pair of vertices.

Definitions – Graph Type

Simple (Undirected) Graph: consists of V , a nonempty set of vertices, and E , a set of unordered pairs of distinct elements of V called edges (undirected)

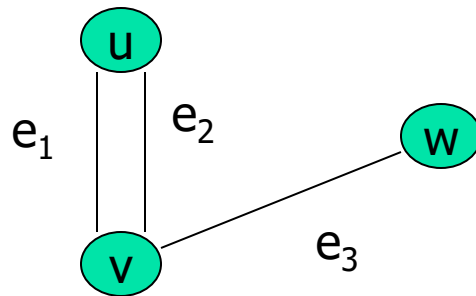
Representation Example: $G(V, E)$, $V = \{u, v, w\}$, $E = \{\{u, v\}, \{v, w\}, \{u, w\}\}$



Definitions – Graph Type

Multigraph: $G(V, E)$, consists of set of vertices V , set of Edges E and a function f from E to $\{\{u, v\} \mid u, v \in V, u \neq v\}$. The edges e_1 and e_2 are called multiple or parallel edges if $f(e_1) = f(e_2)$.

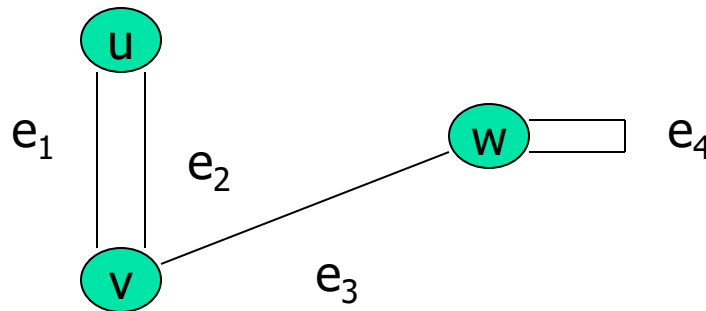
Representation Example: $V = \{u, v, w\}$, $E = \{e_1, e_2, e_3\}$



Definitions – Graph Type

Pseudograph: $G(V,E)$, consists of set of vertices V , set of Edges E and a function F from E to $\{\{u, v\} \mid u, v \in V\}$. Loops allowed in such a graph.

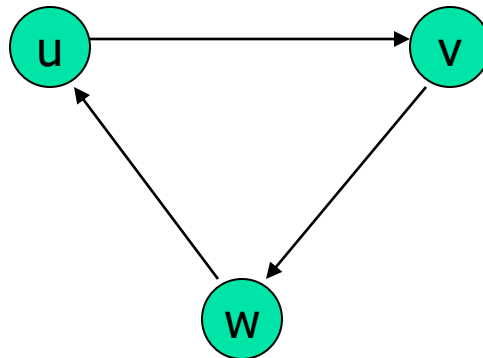
Representation Example: $V = \{u, v, w\}$, $E = \{e_1, e_2, e_3, e_4\}$



Definitions – Graph Type

Directed Graph: $G(V, E)$, set of vertices V , and set of Edges E , that are ordered pair of elements of V (directed edges)

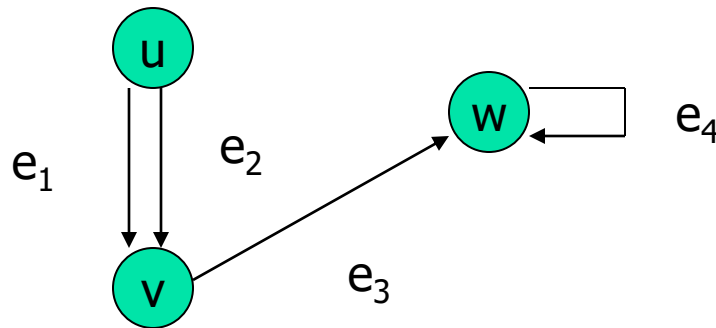
Representation Example: $G(V, E)$, $V = \{u, v, w\}$, $E = \{(u, v), (v, w), (w, u)\}$

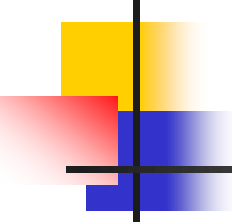


Definitions – Graph Type

Directed Multigraph: $G(V, E)$, consists of set of vertices V , set of Edges E and a function f from E to $\{\{u, v\} \mid u, v \in V\}$. The edges e_1 and e_2 are multiple edges if $f(e_1) = f(e_2)$

Representation Example: $V = \{u, v, w\}$, $E = \{e_1, e_2, e_3, e_4\}$





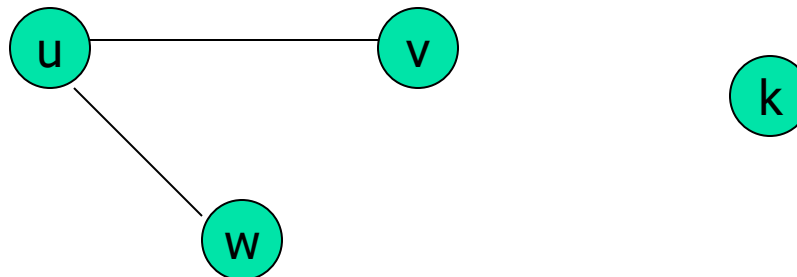
Definitions – Graph Type

Type	Edges	Multiple Edges Allowed ?	Loops Allowed ?
Simple Graph	undirected	No	No
Multigraph	undirected	Yes	No
Pseudograph	undirected	Yes	Yes
Directed Graph	directed	No	Yes
Directed Multigraph	directed	Yes	Yes

Terminology – Undirected graphs

- u and v are **adjacent** if $\{u, v\}$ is an edge, e is called **incident** with u and v . u and v are called **endpoints** of $\{u, v\}$
- **Degree of Vertex ($\deg(v)$):** the number of edges incident on a vertex. A loop contributes twice to the degree
- **Pendant Vertex:** $\deg(v) = 1$
- **Isolated Vertex:** $\deg(k) = 0$

Representation Example: For $V = \{u, v, w\}$, $E = \{\{u, w\}, \{u, v\}\}$, $\deg(u) = 2$, $\deg(v) = 1$, $\deg(w) = 1$, $\deg(k) = 0$, w and v are pendant, k is isolated

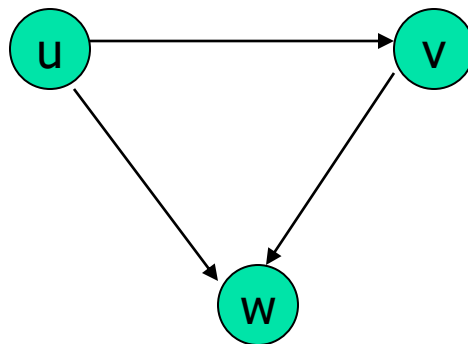


Terminology – Directed graphs

- For the edge (u, v) , u is **adjacent to** v OR v is **adjacent from** u , u – **Initial vertex**, v – **Terminal vertex**
- **In-degree ($\deg^- (u)$)**: number of edges for which u is terminal vertex
- **Out-degree ($\deg^+ (u)$)**: number of edges for which u is initial vertex

Note: A loop contributes 1 to both in-degree and out-degree (why?)

Representation Example: For $V = \{u, v, w\}$, $E = \{ (u, w), (v, w), (u, v) \}$, $\deg^+ (w) = 0$, $\deg^+ (u) = 2$, $\deg^- (v) = 1$, $\deg^+ (v) = 1$, and $\deg^- (w) = 2$, $\deg^+ (u) = 0$





Theorems: Undirected Graphs

Theorem 1

The Handshaking theorem:

$$2e = \sum_{v \in V} \deg(v)$$

Every edge connects 2 vertices



Theorems: Undirected Graphs

Theorem 2:

An undirected graph has even number of vertices with odd degree

Proof V_1 is the set of even degree vertices and V_2 refers to odd degree vertices

$$2e = \sum_{v \in V} \deg(v) = \sum_{u \in V_1} \deg(u) + \sum_{v \in V_2} \deg(v)$$

$\Rightarrow \deg(v)$ is even for $v \in V_1$,

\Rightarrow The first term in the right hand side of the last inequality is even.

\Rightarrow The sum of the last two terms on the right hand side of the last inequality is even since sum is $2e$.

Hence second term is also even

\Rightarrow second term $\sum_{v \in V_2} \deg(v) = \text{even}$



Theorems: directed Graphs

- **Theorem 3:** $\sum \deg^+(u) = \sum \deg^-(u) = |E|$



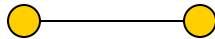
Simple graphs – special cases

- **Complete graph:** K_n , is the simple graph that contains exactly one edge between each pair of distinct vertices.

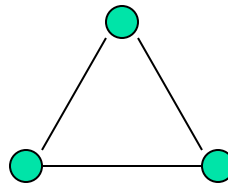
Representation Example: K_1 , K_2 , K_3 , K_4



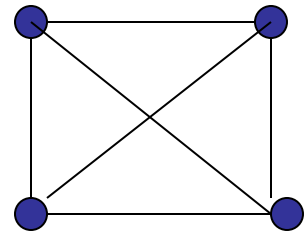
K_1



K_2



K_3

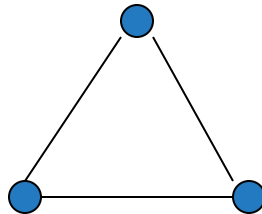


K_4

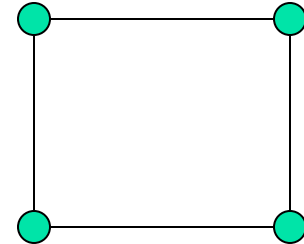
Simple graphs – special cases

- **Cycle:** C_n , $n \geq 3$ consists of n vertices $v_1, v_2, v_3 \dots v_n$ and edges $\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\} \dots \{v_{n-1}, v_n\}, \{v_n, v_1\}$

Representation Example: C_3, C_4



C_3

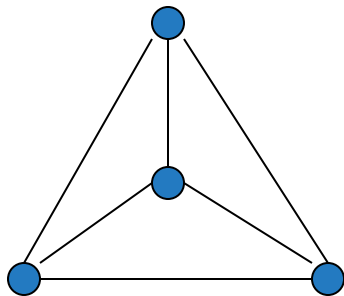


C_4

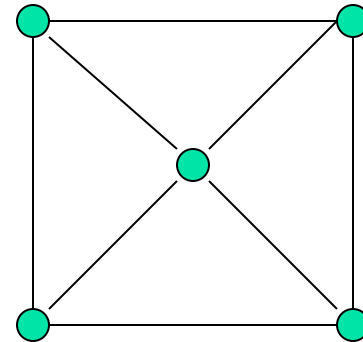
Simple graphs – special cases

- **Wheels:** W_n , obtained by adding additional vertex to C_n and connecting all vertices to this new vertex by new edges.

Representation Example: W_3 , W_4



W_3

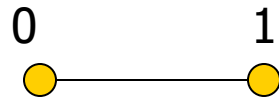


W_4

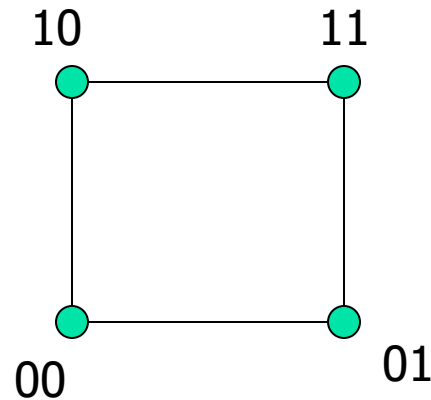
Simple graphs – special cases

- **N-cubes:** Q_n , vertices represented by $2n$ bit strings of length n . Two vertices are adjacent if and only if the bit strings that they represent differ by exactly one bit position

Representation Example: Q_1, Q_2



Q_1



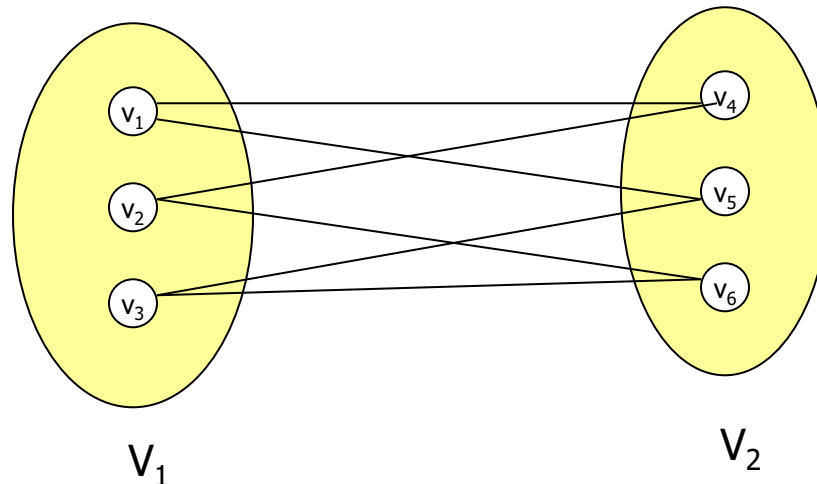
Q_2

Bipartite graphs

- In a simple graph G , if V can be partitioned into two disjoint sets V_1 and V_2 such that every edge in the graph connects a vertex in V_1 and a vertex in V_2 (so that no edge in G connects either two vertices in V_1 or two vertices in V_2)

Application example: Representing Relations

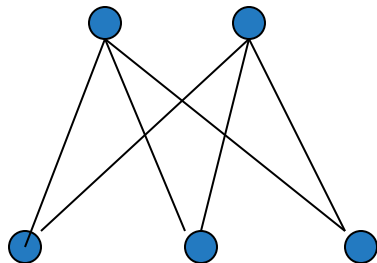
Representation example: $V_1 = \{v_1, v_2, v_3\}$ and $V_2 = \{v_4, v_5, v_6\}$,



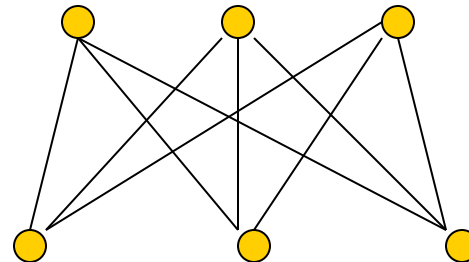
Complete Bipartite graphs

- $K_{m,n}$ is the graph that has its vertex set partitioned into two subsets of m and n vertices, respectively. There is an edge between two vertices if and only if one vertex is in the first subset and the other vertex is in the second subset.

Representation example: $K_{2,3}$, $K_{3,3}$



$K_{2,3}$



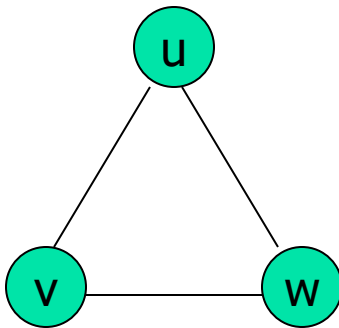
$K_{3,3}$

Subgraphs

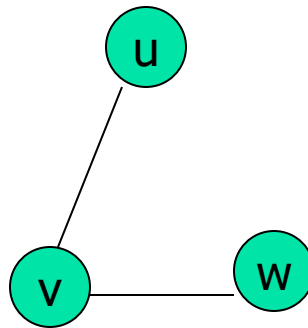
- A subgraph of a graph $G = (V, E)$ is a graph $H = (V', E')$ where V' is a subset of V and E' is a subset of E

Application example: solving sub-problems within a graph

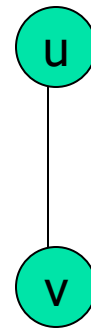
Representation example: $V = \{u, v, w\}$, $E = (\{u, v\}, \{v, w\}, \{w, u\})$,
 H_1 , H_2



G



H_1

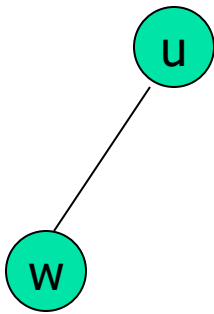


H_2

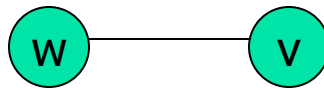
Subgraphs

- $G = G1 \cup G2$ wherein $E = E1 \cup E2$ and $V = V1 \cup V2$, G , $G1$ and $G2$ are simple graphs of G

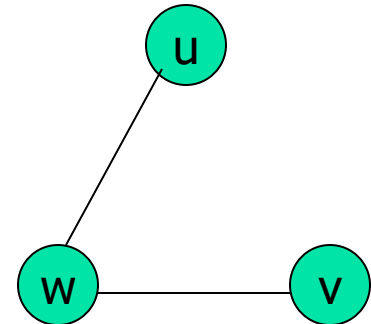
Representation example: $V1 = \{u, w\}$, $E1 = \{\{u, w\}\}$, $V2 = \{w, v\}$, $E2 = \{\{w, v\}\}$, $V = \{u, v, w\}$, $E = \{\{u, w\}, \{w, v\}\}$



G1



G2



G



Representation

- **Incidence (Matrix):** Most useful when information about edges is more desirable than information about vertices.
- **Adjacency (Matrix/List):** Most useful when information about the vertices is more desirable than information about the edges. These two representations are also most popular since information about the vertices is often more desirable than edges in most applications



Representation- Incidence Matrix

- $G = (V, E)$ be an undirected graph. Suppose that $v_1, v_2, v_3, \dots, v_n$ are the vertices and e_1, e_2, \dots, e_m are the edges of G . Then the incidence matrix with respect to this ordering of V and E is the $n \times m$ matrix $M = [m_{ij}]$, where

$$m_{ij} = \begin{cases} 1 & \text{when edge } e_j \text{ is incident with } v_i \\ 0 & \text{otherwise} \end{cases}$$

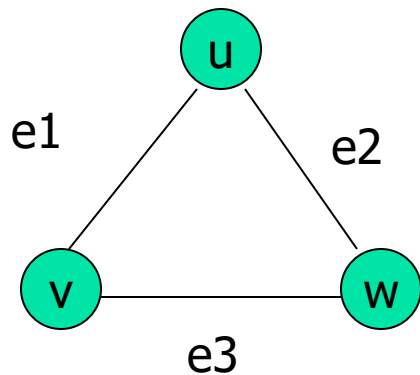
Can also be used to represent :

Multiple edges: by using columns with identical entries, since these edges are incident with the same pair of vertices

Loops: by using a column with exactly one entry equal to 1, corresponding to the vertex that is incident with the loop

Representation- Incidence Matrix

- Representation Example: $G = (V, E)$



	e_1	e_2	e_3
v	1	0	1
u	1	1	0
w	0	1	1



Representation- Adjacency Matrix

- There is an $N \times N$ matrix, where $|V| = N$, the Adjacent Matrix ($N \times N$) $A = [a_{ij}]$

For undirected graph

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G \\ 0 & \text{otherwise} \end{cases}$$

- **For directed graph**

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge of } G \\ 0 & \text{otherwise} \end{cases}$$

- This makes it easier to find subgraphs, and to reverse graphs if needed.

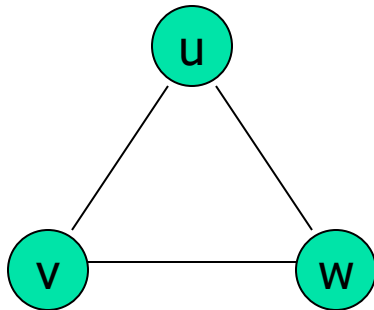


Representation- Adjacency Matrix

- Adjacency is chosen on the ordering of vertices. Hence, there are as many as $n!$ such matrices.
- The adjacency matrix of simple graphs are symmetric ($a_{ij} = a_{ji}$) (why?)
- When there are relatively few edges in the graph the adjacency matrix is a **sparse matrix**
- Directed Multigraphs can be represented by using a_{ij} = number of edges from v_i to v_j

Representation- Adjacency Matrix

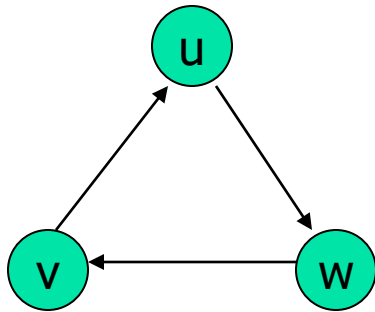
- Example: Undirected Graph $G(V, E)$



	v	u	w
v	0	1	1
u	1	0	1
w	1	1	0

Representation- Adjacency Matrix

- Example: directed Graph $G(V, E)$

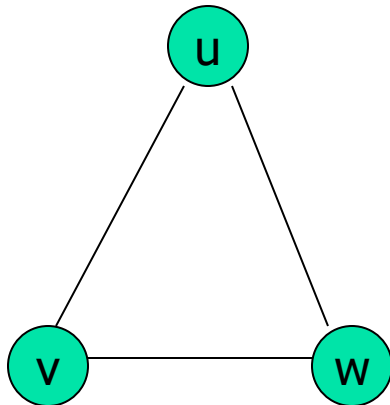


	v	u	w
v	0	1	0
u	0	0	1
w	1	0	0

Representation- Adjacency List

Each node (vertex) has a list of which nodes (vertex) it is adjacent

Example: undirected graph $G(V, E)$



node	Adjacency List
u	v , w
v	w, u
w	u , v



Graph - Isomorphism

- $G1 = (V1, E1)$ and $G2 = (V2, E2)$ are isomorphic if:
- There is a one-to-one and onto function f from $V1$ to $V2$ with the property that
 - a and b are adjacent in $G1$ if and only if $f(a)$ and $f(b)$ are adjacent in $G2$, for all a and b in $V1$.
- Function f is called isomorphism

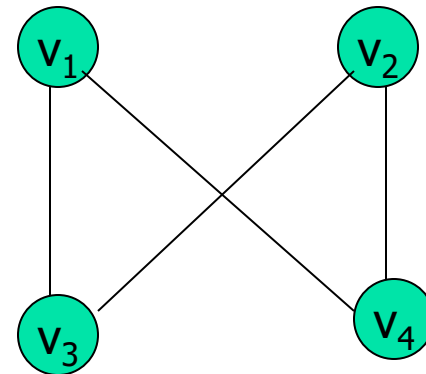
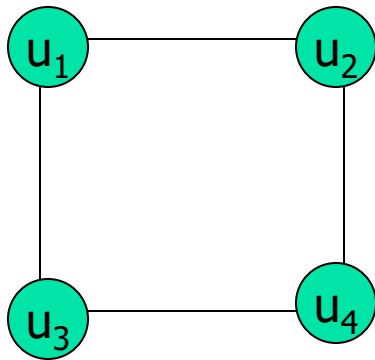
Application Example:

In chemistry, to find if two compounds have the same structure

Graph - Isomorphism

Representation example: $G1 = (V1, E1)$, $G2 = (V2, E2)$

$f(u_1) = v_1$, $f(u_2) = v_4$, $f(u_3) = v_3$, $f(u_4) = v_2$,





Connectivity

- Basic Idea: In a Graph Reachability among vertices by traversing the edges

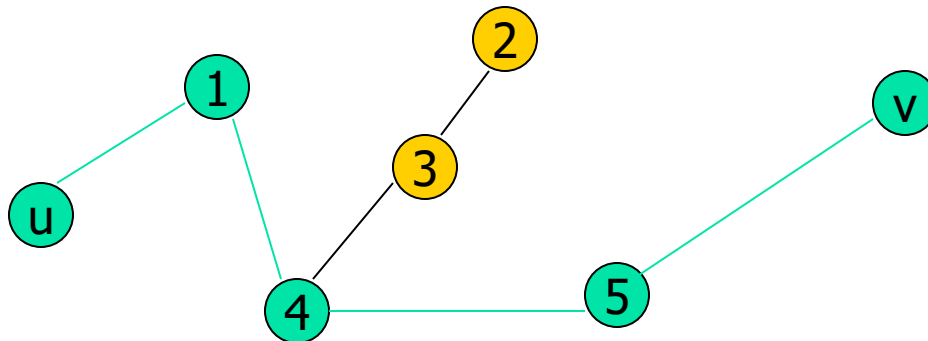
Application Example:

- In a city to city road-network, if one city can be reached from another city.
- Problems if determining whether a message can be sent between two
computer using intermediate links
- Efficiently planning routes for data delivery in the Internet

Connectivity – Path

A **Path** is a sequence of edges that begins at a vertex of a graph and travels along edges of the graph, always connecting pairs of adjacent vertices.

Representation example: $G = (V, E)$, Path P represented, from u to v is $\{\{u, 1\}, \{1, 4\}, \{4, 5\}, \{5, v\}\}$





Connectivity – Path

Definition for Directed Graphs

A **Path** of length n (> 0) from u to v in G is a sequence of n edges $e_1, e_2, e_3, \dots, e_n$ of G such that $f(e_1) = (x_0, x_1)$, $f(e_2) = (x_1, x_2)$, \dots , $f(e_n) = (x_{n-1}, x_n)$, where $x_0 = u$ and $x_n = v$. A path is said to pass through x_0, x_1, \dots, x_n or traverse $e_1, e_2, e_3, \dots, e_n$

For Simple Graphs, sequence is x_0, x_1, \dots, x_n

In directed multigraphs when it is not necessary to distinguish between their edges, we can use sequence of vertices to represent the path

Circuit/Cycle: $u = v$, length of path > 0

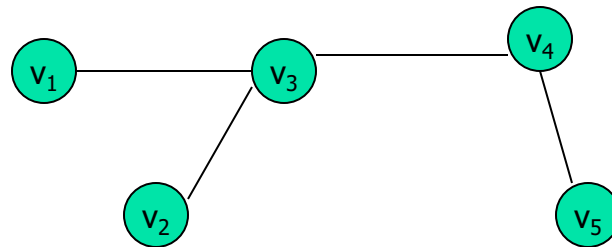
Simple Path: does not contain an edge more than once

Connectivity – Connectedness

Undirected Graph

An undirected graph is connected if there exists a simple path between every pair of vertices

Representation Example: $G(V, E)$ is connected since for $V = \{v_1, v_2, v_3, v_4, v_5\}$, there exists a path between $\{v_i, v_j\}$, $1 \leq i, j \leq 5$

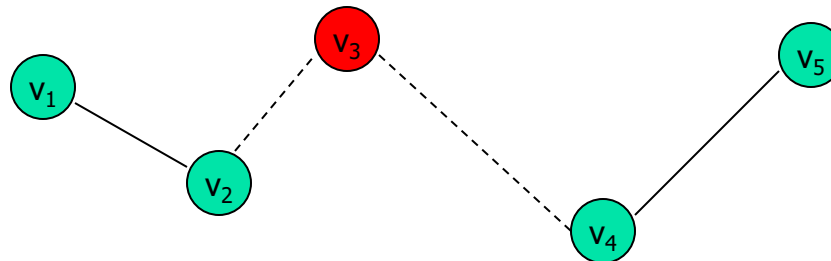


Connectivity – Connectedness

Undirected Graph

- **Articulation Point (Cut vertex):** removal of a vertex produces a subgraph with more connected components than in the original graph. The removal of a cut vertex from a connected graph produces a graph that is not connected
- **Cut Edge:** An edge whose removal produces a subgraph with more connected components than in the original graph.

Representation example: $G(V, E)$, v_3 is the articulation point or edge $\{v_2, v_3\}$, the number of connected components is 2 (> 1)





Connectivity – Connectedness

Directed Graph

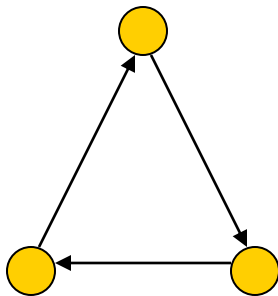
- A directed graph is **strongly connected** if there is a path from a to b and from b to a whenever a and b are vertices in the graph
- A directed graph is **weakly connected** if there is a (undirected) path between every two vertices in the underlying undirected path

A strongly connected Graph can be weakly connected but the vice-versa is not true (why?)

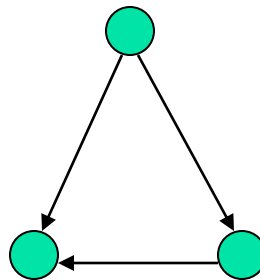
Connectivity – Connectedness

Directed Graph

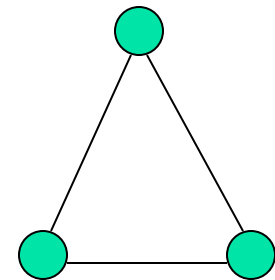
Representation example: G1 (Strong component), G2 (Weak Component), G3 is undirected graph representation of G2 or G1



G1



G2



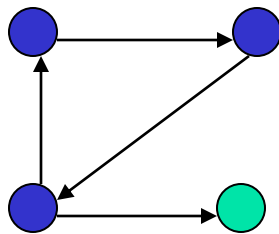
G3

Connectivity – Connectedness

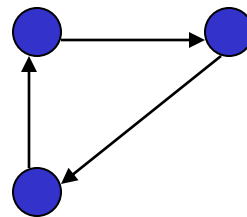
- **Directed Graph**

Strongly connected Components: subgraphs of a Graph G that are strongly connected

Representation example: G_1 is the strongly connected component in G



G



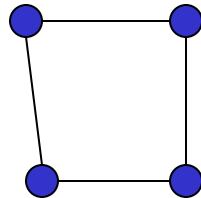
G_1



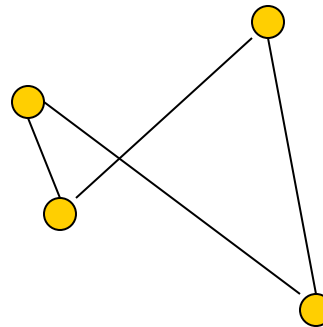
Isomorphism - revisited

A isomorphic invariant for simple graphs is the existence of a simple circuit of length k , k is an integer > 2 (why ?)

Representation example: $G1$ and $G2$ are isomorphic since we have the invariants, similarity in degree of nodes, number of edges, length of circuits



$G1$



$G2$



Counting Paths

- **Theorem:** Let G be a graph with adjacency matrix A with respect to the ordering v_1, v_2, \dots, v_n (with directed on undirected edges, with multiple edges and loops allowed). The number of different paths of length r from v_i to v_j , where r is a positive integer, equals the $(i, j)^{\text{th}}$ entry of (adjacency matrix) A^r .

Proof: By Mathematical Induction.

Base Case: For the case $N = 1$, $a_{ij} = 1$ implies that there is a path of length 1. This is true since this corresponds to an edge between two vertices.

We assume that theorem is true for $N = r$ and prove the same for $N = r + 1$. Assume that the $(i, j)^{\text{th}}$ entry of A^r is the number of different paths of length r from v_i to v_j . By induction hypothesis, b_{ik} is the number of paths of length r from v_i to v_k .



Counting Paths

Case $r + 1$: In $A^{r+1} = A^r \cdot A$,

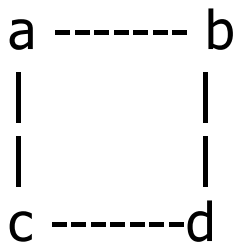
The $(i, j)^{\text{th}}$ entry in A^{r+1} , $b_{i1}a_{1j} + b_{i2}a_{2j} + \dots + b_{in}a_{nj}$
where b_{ik} is the $(i, k)^{\text{th}}$ entry of A^r .

By induction hypothesis, b_{ik} is the number of paths of length r from v_i to v_k .

The $(i, j)^{\text{th}}$ entry in A^{r+1} corresponds to the length between i and j and the length is $r+1$. This path is made up of length r from v_i to v_k and of length from v_k to v_j . By product rule for counting, the number of such paths is $b_{ik} \cdot a_{kj}$. The result is $b_{i1}a_{1j} + b_{i2}a_{2j} + \dots + b_{in}a_{nj}$, the desired result.



Counting Paths

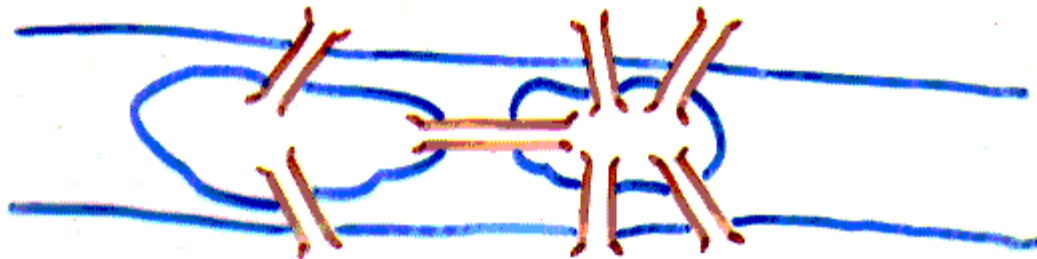


$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \quad A^4 = \begin{bmatrix} 8 & 0 & 0 & 8 \\ 0 & 8 & 8 & 0 \\ 0 & 8 & 8 & 0 \\ 8 & 0 & 0 & 8 \end{bmatrix}$$

Number of paths of length 4 from a to d is (1,4) th entry of $A^4 = 8$.

The Seven Bridges of Königsberg, Germany

- The residents of Königsberg, Germany, wondered if it was possible to take a walking tour of the town that crossed each of the seven bridges over the Presel river exactly once. Is it possible to start at some node and take a walk that uses each edge exactly once, and ends at the starting node?

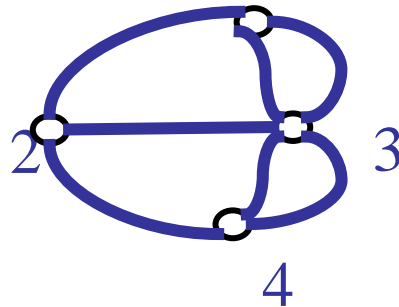




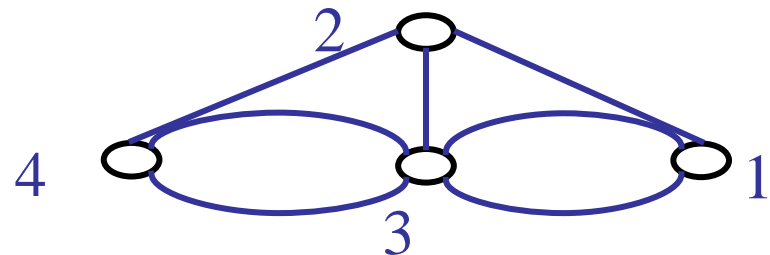
The Seven Bridges of Königsberg, Germany

You can redraw the original picture as long as for every edge between nodes i and j in the original you put an edge between nodes i and j in the redrawn version (and you put no other edges in the redrawn version).

Original:



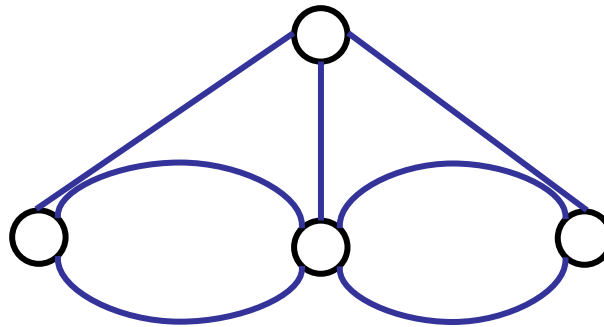
Redrawn:





The Seven Bridges of Königsberg, Germany

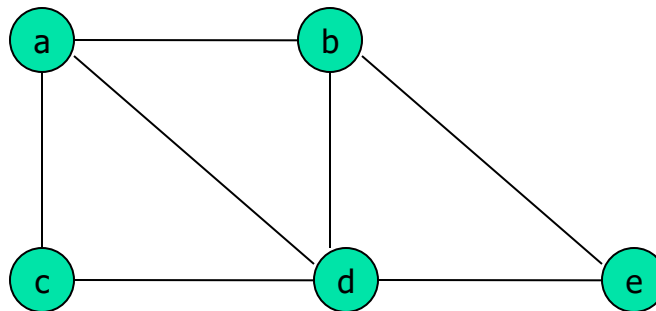
Euler:



- Has no tour that uses each edge exactly once.
- (Even if we allow the walk to start and finish in different places.)
- Can you see why?

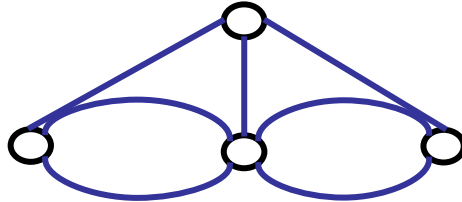
Euler - definitions

- An **Eulerian path** (**Eulerian trail**, **Euler walk**) in a graph is a path that uses each edge precisely once. If such a path exists, the graph is called **traversable**.
- An **Eulerian cycle** (**Eulerian circuit**, **Euler tour**) in a graph is a cycle that uses each edge precisely once. If such a cycle exists, the graph is called **Eulerian** (also **unicursal**).
- Representation example: G1 has Euler path a, c, d, e, b, d, a, b





The problem in our language:

Show that  is not Eulerian.

In fact, it contains no Euler trail.



Euler - theorems

1. A connected graph G is Eulerian if and only if G is connected and has no vertices of odd degree
2. A connected graph G has an Euler trail from node a to some other node b if and only if G is connected and $a \neq b$ are the only two nodes of odd degree



Euler – theorems (\Rightarrow)

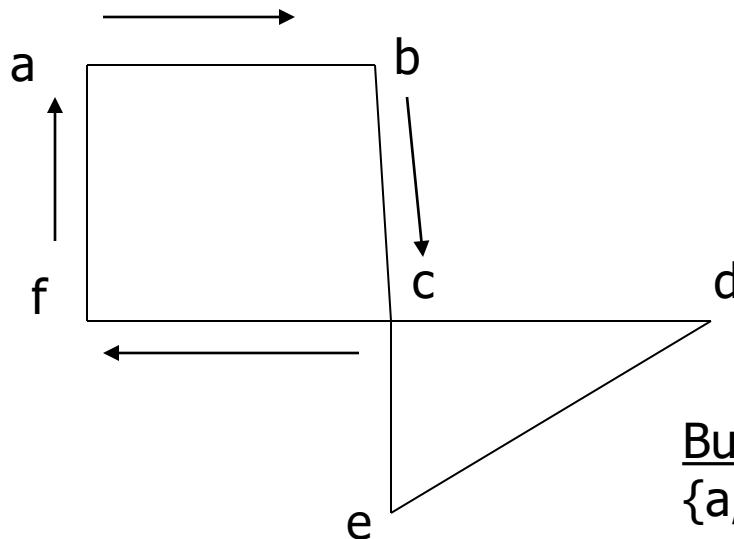
Assume G has an Euler trail T from node a to node b (a and b not necessarily distinct).

For every node besides a and b , T uses an edge to exit for each edge it uses to enter. Thus, the degree of the node is even.

1. If $a = b$, then a also has even degree. \rightarrow Euler circuit
2. If $a \neq b$, then a and b both have odd degree. \rightarrow Euler path

Euler - theorems

1. A connected graph G is Eulerian if and only if G is connected and has no vertices of odd degree



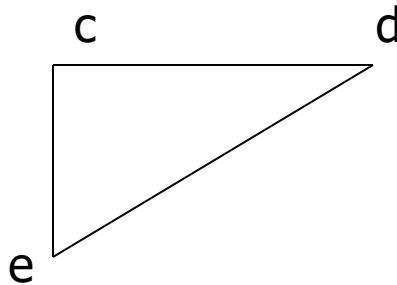
Building a simple path:
 $\{a,b\}, \{b,c\}, \{c,f\}, \{f,a\}$

Euler circuit constructed if all edges are used. True here?



Euler - theorems

1. A connected graph G is Eulerian if and only if G is connected and has no vertices of odd degree



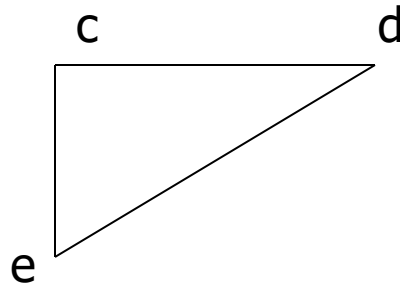
Delete the simple path:
 $\{a,b\}, \{b,c\}, \{c,f\}, \{f,a\}$

C is the common vertex for this sub-graph with its "parent".



Euler - theorems

1. A connected graph G is Eulerian if and only if G is connected and has no vertices of odd degree



Constructed subgraph may not be connected.

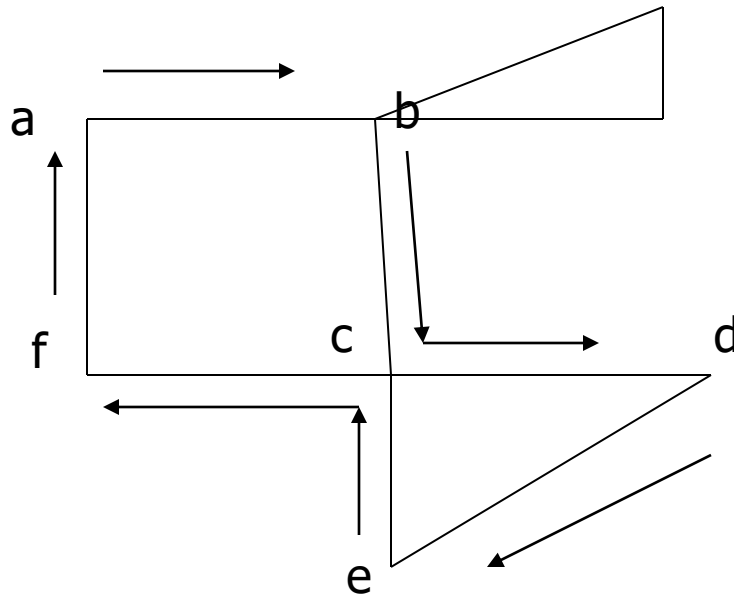
C is the common vertex for this sub-graph with its "parent".

C has even degree.

Start at c and take a walk:
 $\{c,d\}, \{d,e\}, \{e,c\}$

Euler - theorems

1. A connected graph G is Eulerian if and only if G is connected and has no vertices of odd degree



"Splice" the circuits in the 2 graphs:

$\{a,b\}, \{b,c\}, \{c,f\}, \{f,a\}$

"+"

$\{c,d\}, \{d,e\}, \{e,c\}$

"="

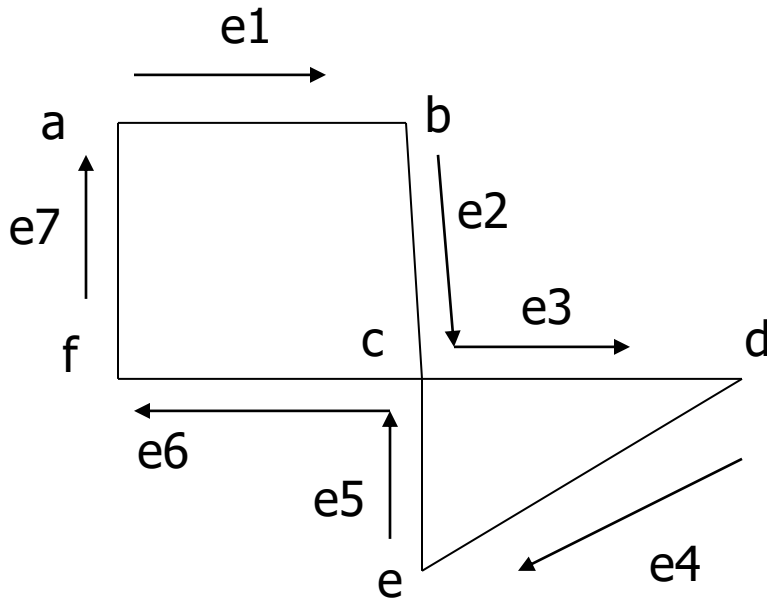
$\{a,b\}, \{b,c\}, \{c,d\}, \{d,e\}, \{e,c\}, \{c,f\}, \{f,a\}$



Euler Circuit

1. Circuit $C :=$ a circuit in G beginning at an arbitrary vertex v .
 1. Add edges successively to form a path that returns to this vertex.
2. $H := G - \text{above circuit } C$
3. While H has edges
 1. Sub-circuit $sc :=$ a circuit that begins at a vertex in H that is also in C (e.g., vertex " c ")
 2. $H := H - sc$ (- all isolated vertices)
 3. Circuit $:=$ circuit C "spliced" with sub-circuit sc
4. Circuit C has the Euler circuit.

Representation- Incidence Matrix



	e_1	e_2	e_3	e_4	e_5	e_6	e_7
a	1	0	0	0	0	0	1
b	1	1	0	0	0	0	0
c	0	1	1	0	1	1	0
d	0	0	1	1	0	0	0
e	0	0	0	1	1	0	0
f	0	0	0	0	0	1	1



Hamiltonian Graph

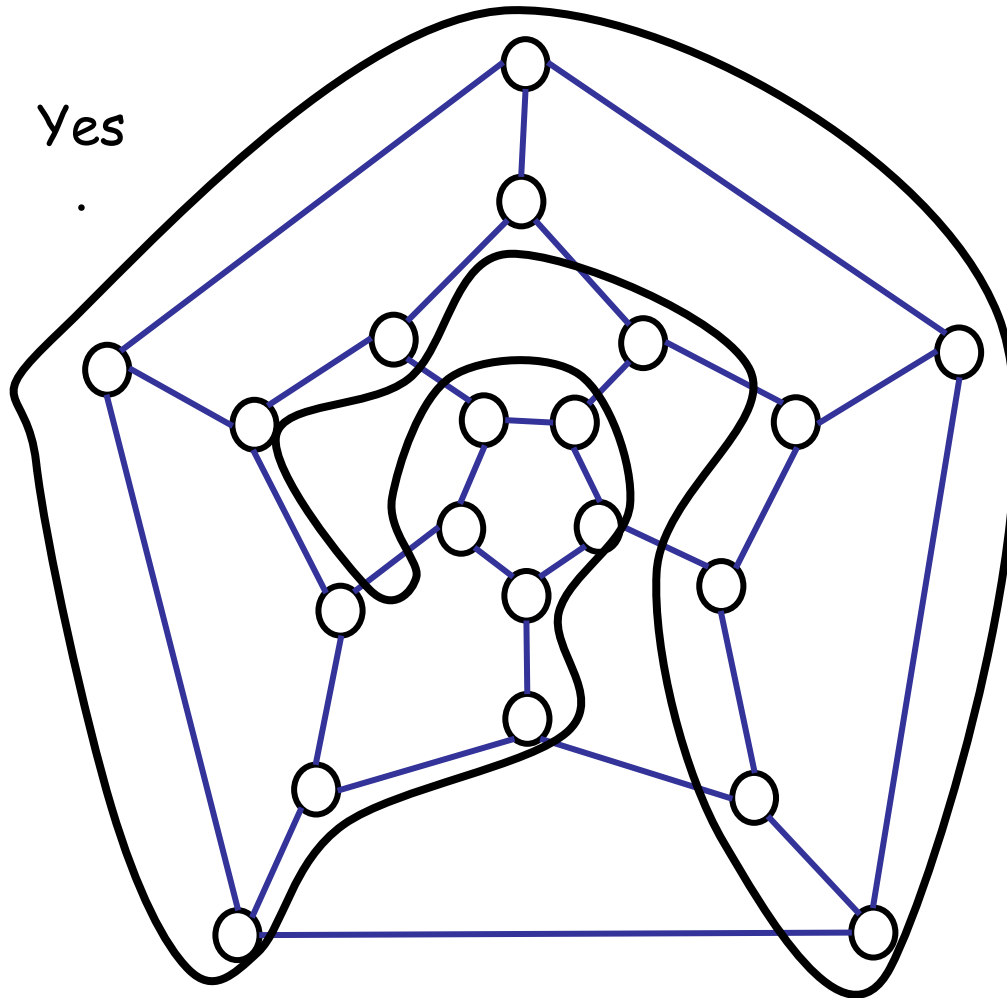
- **Hamiltonian path** (also called *traceable path*) is a path that visits each vertex exactly once.
- A **Hamiltonian cycle** (also called *Hamiltonian circuit*, *vertex tour* or *graph cycle*) is a cycle that visits each vertex exactly once (except for the starting vertex, which is visited once at the start and once again at the end).
- A graph that contains a Hamiltonian path is called a **traceable graph**. A graph that contains a Hamiltonian cycle is called a **Hamiltonian graph**. Any Hamiltonian cycle can be converted to a Hamiltonian path by removing one of its edges, but a Hamiltonian path can be extended to Hamiltonian cycle only if its endpoints are adjacent.

A graph of the vertices of a dodecahedron.

Is it Hamiltonian?

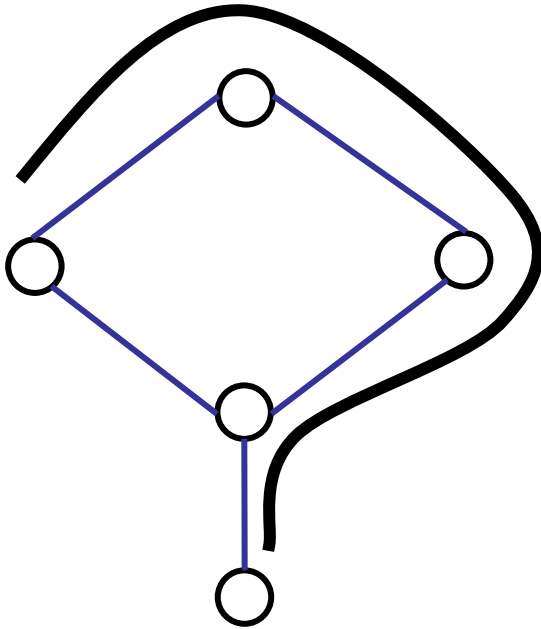
Yes

.





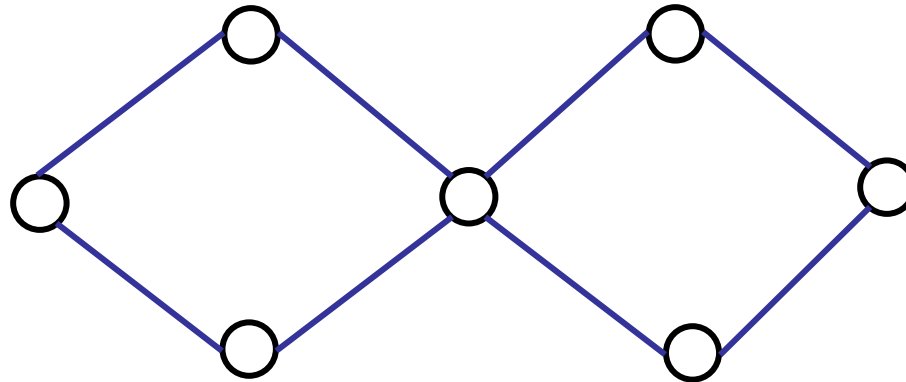
Hamiltonian Graph



This one has a Hamiltonian path, but not a Hamiltonian tour.



Hamiltonian Graph



This one has an Euler tour, but no Hamiltonian path.



Hamiltonian Graph

- Similar notions may be defined for directed graphs, where edges (arcs) of a path or a cycle are required to point in the same direction, i.e., connected tail-to-head.
- The *Hamiltonian cycle problem* or *Hamiltonian circuit problem* in graph theory is to find a Hamiltonian cycle in a given graph. The *Hamiltonian path problem* is to find a Hamiltonian path in a given graph.
- There is a simple relation between the two problems. The Hamiltonian path problem for graph **G** is equivalent to the Hamiltonian cycle problem in a graph **H** obtained from **G** by adding a new vertex and connecting it to all vertices of **G**.
- Both problems are NP-complete. However, certain classes of graphs always contain Hamiltonian paths. For example, it is known that every tournament has an odd number of Hamiltonian paths.



Hamiltonian Graph

- **DIRAC'S Theorem:** if G is a simple graph with n vertices with $n \geq 3$ such that the degree of every vertex in G is at least $n/2$ then G has a Hamilton circuit.
- **ORE'S Theorem:** if G is a simple graph with n vertices with $n \geq 3$ such that $\deg(u) + \deg(v) \geq n$ for every pair of nonadjacent vertices u and v in G , then G has a Hamilton circuit.



Shortest Path

- Generalize distance to weighted setting
- Digraph $G = (V, E)$ with weight function $W: E \rightarrow R$ (assigning real values to edges)
- Weight of path $p = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ is

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of the minimum weight
- Applications
 - static/dynamic network routing
 - robot motion planning
 - map/route generation in traffic

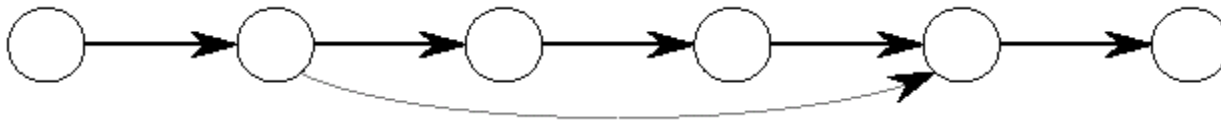


Shortest-Path Problems

- Shortest-Path problems
 - **Single-source (single-destination).** Find a shortest path from a given source (vertex s) to each of the vertices. The topic of this lecture.
 - **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
 - **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.
 - Unweighted shortest-paths – BFS.

Optimal Substructure

- *Theorem:* subpaths of shortest paths are shortest paths
- Proof ("cut and paste")
 - if some subpath were not the shortest path, one could substitute the shorter subpath and create a shorter total path





Negative Weights and Cycles?

- Negative edges are OK, as long as there are no *negative weight cycles* (otherwise paths with arbitrary small “lengths” would be possible)
- Shortest-paths can have no cycles (otherwise we could improve them by removing cycles)
 - Any shortest-path in graph G can be no longer than $n - 1$ edges, where n is the number of vertices

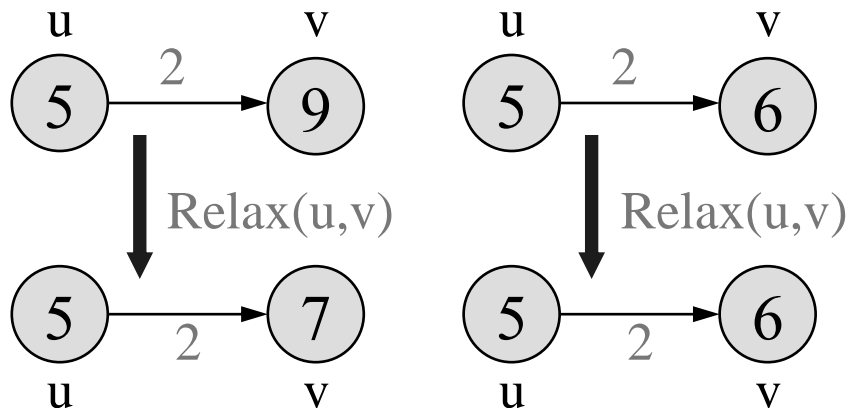


Shortest Path Tree

- The result of the algorithms – a *shortest path tree*. For each vertex v , it
 - records a shortest path from the start vertex s to v .
 $v.\text{parent}()$ gives a predecessor of v in this shortest path
 - gives a shortest path length from s to v , which is recorded in $v.\text{d}()$.
- The same pseudo-code assumptions are used.
- **Vertex** ADT with operations:
 - **adjacent()**: *VertexSet*
 - **d()**: *int* and **setd**(k : *int*)
 - **parent()**: *Vertex* and **setparent**(p : *Vertex*)

Relaxation

- For each vertex v in the graph, we maintain $v.d()$, the estimate of the shortest path from s , initialized to ∞ at the start
- Relaxing an edge (u, v) means testing whether we can improve the shortest path to v found so far by going through u



```
Relax ( $u, v, G$ )  
if  $v.d() > u.d() + G.w(u, v)$  then  
     $v.setd(u.d() + G.w(u, v))$   
     $v.setparent(u)$ 
```




Dijkstra's Algorithm

- Non-negative edge weights
- Greedy, similar to Prim's algorithm for MST
- Like breadth-first search (if all weights = 1, one can simply use BFS)
- Use Q , a priority queue ADT keyed by $v.d()$ (BFS used FIFO queue, here we use a PQ, which is re-organized whenever some d decreases)
- Basic idea
 - maintain a set S of solved vertices
 - at each step select "closest" vertex u , add it to S , and relax all edges from u

Dijkstra's Algorithm

Solution to **Single-source (single-destination)**.

- Input: Graph G , start vertex s

Dijkstra (G, s)

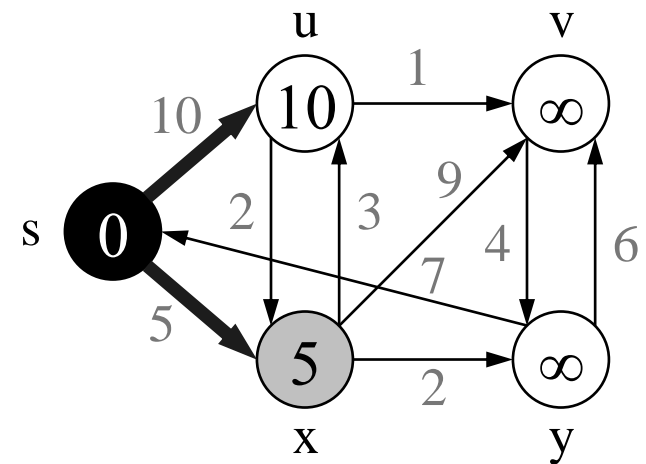
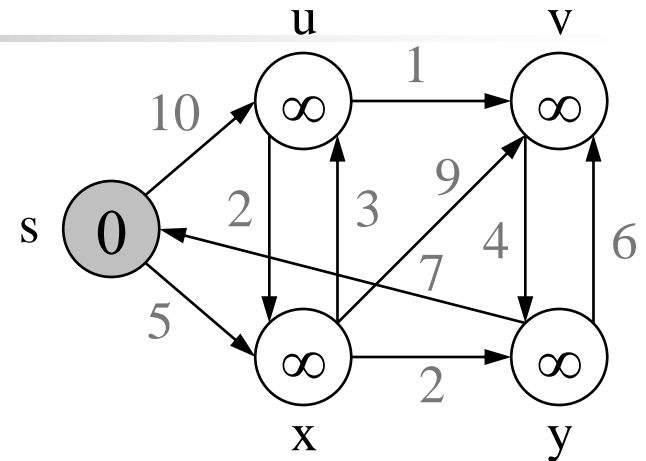
```
01 for each vertex  $u \in G.V()$ 
02      $u.setd(\infty)$ 
03      $u.setparent(NIL)$ 
04  $s.setd(0)$ 
05  $S \leftarrow \emptyset$  // Set  $S$  is used to explain the
    algorithm
06  $Q.init(G.V())$  //  $Q$  is a priority queue ADT
07 while not  $Q.isEmpty()$ 
08      $u \leftarrow Q.extractMin()$ 
09      $S \leftarrow S \cup \{u\}$ 
10     for each  $v \in u.adjacent()$  do
11         Relax ( $u, v, G$ )
12          $Q.modifyKey(v)$ 
```

relaxing
edges

Dijkstra's Example

Dijkstra(G, s)

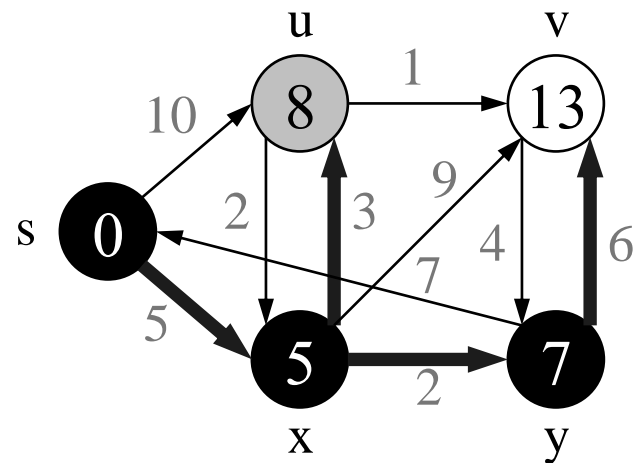
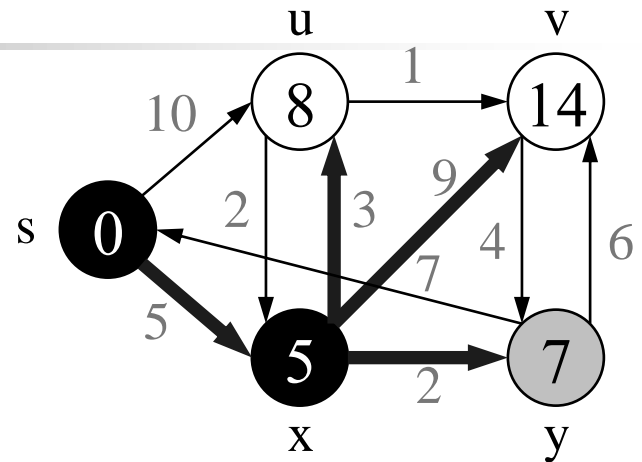
```
01 for each vertex  $u \in G.V()$ 
02    $u.setd(\infty)$ 
03    $u.setparent(NIL)$ 
04  $s.setd(0)$ 
05  $S \leftarrow \emptyset$ 
06  $Q.init(G.V())$ 
07 while not  $Q.isEmpty()$ 
08    $u \leftarrow Q.extractMin()$ 
09    $S \leftarrow S \cup \{u\}$ 
10   for each  $v \in u.adjacent()$  do
11     Relax( $u, v, G$ )
12      $Q.modifyKey(v)$ 
```



Dijkstra's Example

Dijkstra(G, s)

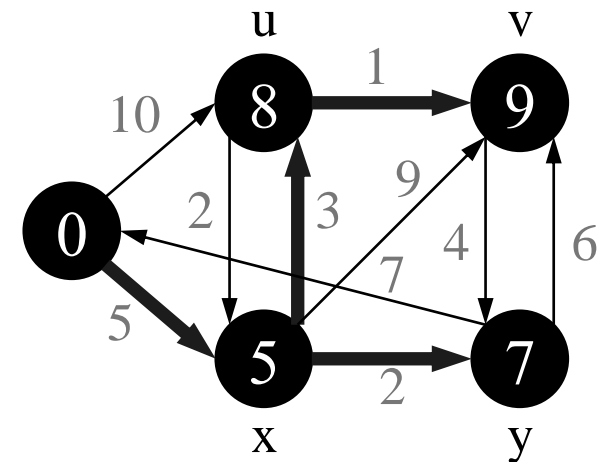
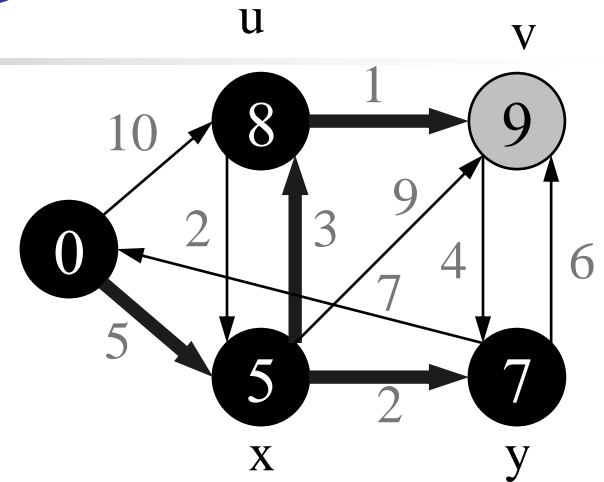
```
01 for each vertex  $u \in G.V()$ 
02    $u.setd(\infty)$ 
03    $u.setparent(NIL)$ 
04  $s.setd(0)$ 
05  $S \leftarrow \emptyset$ 
06  $Q.init(G.V())$ 
07 while not  $Q.isEmpty()$ 
08    $u \leftarrow Q.extractMin()$ 
09    $S \leftarrow S \cup \{u\}$ 
10   for each  $v \in u.adjacent()$  do
11     Relax( $u, v, G$ )
12      $Q.modifyKey(v)$ 
```



Dijkstra's Example

Dijkstra(G, s)

```
01 for each vertex  $u \in G.V()$ 
02      $u.setd(\infty)$ 
03      $u.setparent(NIL)$ 
04  $s.setd(0)$ 
05  $S \leftarrow \emptyset$ 
06  $Q.init(G.V())$ 
07 while not  $Q.isEmpty()$ 
08      $u \leftarrow Q.extractMin()$ 
09      $S \leftarrow S \cup \{u\}$ 
10     for each  $v \in u.adjacent()$  do
11         Relax( $u, v, G$ )
12          $Q.modifyKey(v)$ 
```





Dijkstra's Algorithm

- $O(n^2)$ operations
 - $(n-1)$ iterations: 1 for each vertex added to the distinguished set S .
 - $(n-1)$ iterations: for each adjacent vertex of the one added to the distinguished set.
- Note: it is single source – single destination algorithm



Traveling Salesman Problem

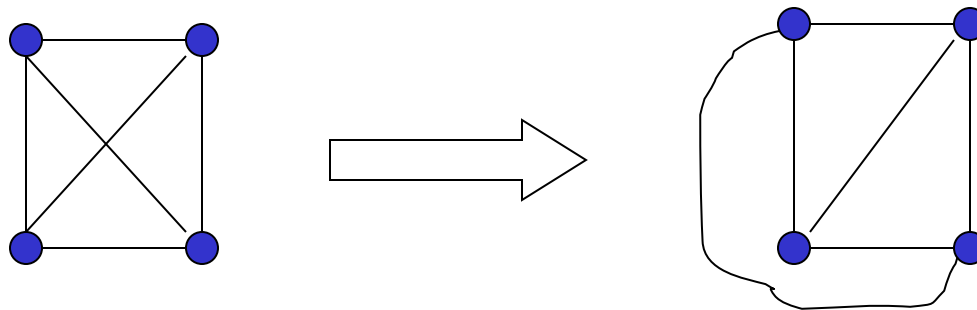
- Given a number of cities and the costs of traveling from one to the other, what is the cheapest roundtrip route that visits each city once and then returns to the starting city?
- An equivalent formulation in terms of graph theory is: Find the Hamiltonian cycle with the least weight in a weighted graph.
- It can be shown that the requirement of returning to the starting city does not change the computational complexity of the problem.
- A related problem is the (bottleneck TSP): Find the Hamiltonian cycle in a weighted graph with the minimal length of the longest edge.

Planar Graphs

A graph (or multigraph) G is called *planar* if G can be drawn in the plane with its edges intersecting only at vertices of G , such a drawing of G is called an *embedding* of G in the plane.

Application Example: VLSI design (overlapping edges requires extra layers),
Circuit design (cannot overlap wires on board)

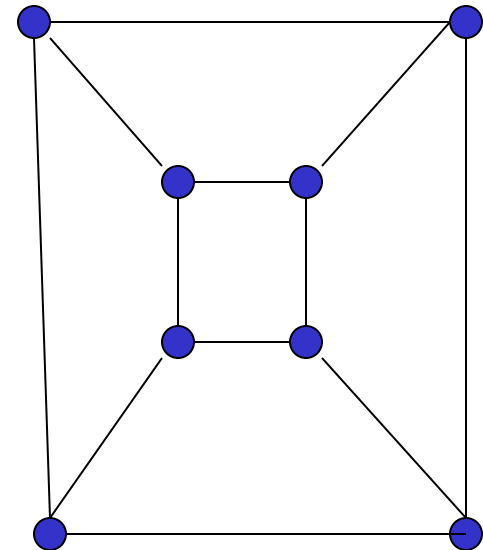
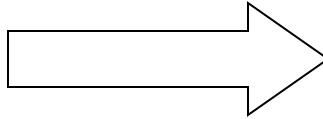
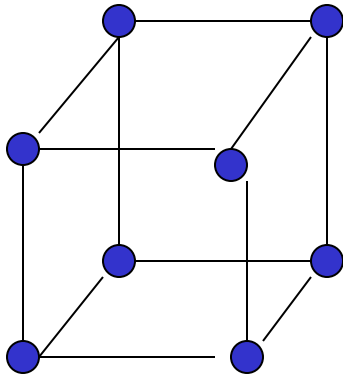
Representation examples: K_1, K_2, K_3, K_4 are planar, K_n for $n > 4$ are non-planar



K_4

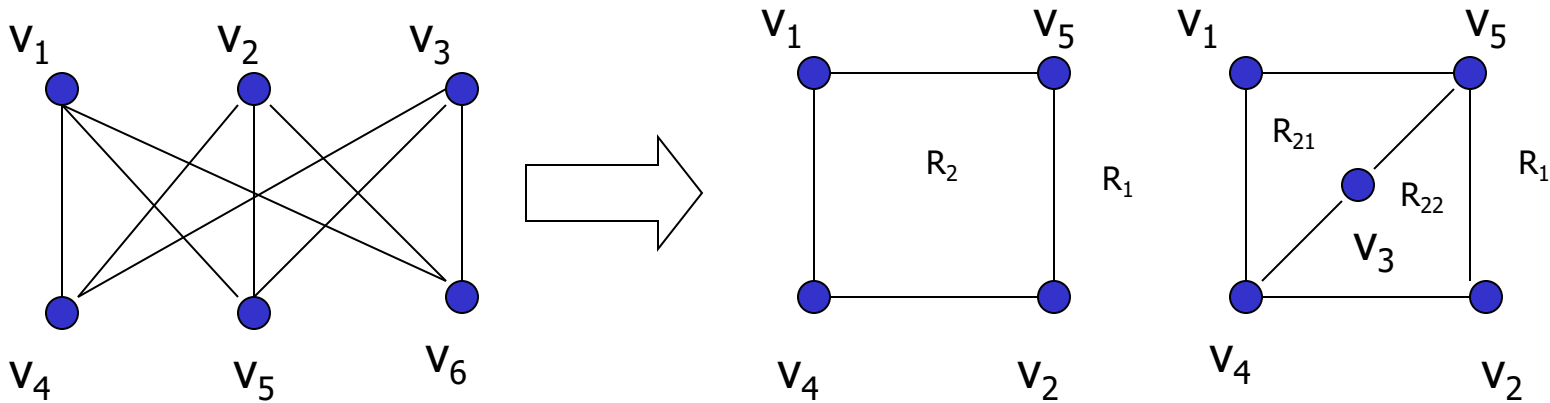
Planar Graphs

- Representation examples: Q_3



Planar Graphs

- Representation examples: $K_{3,3}$ is Nonplanar

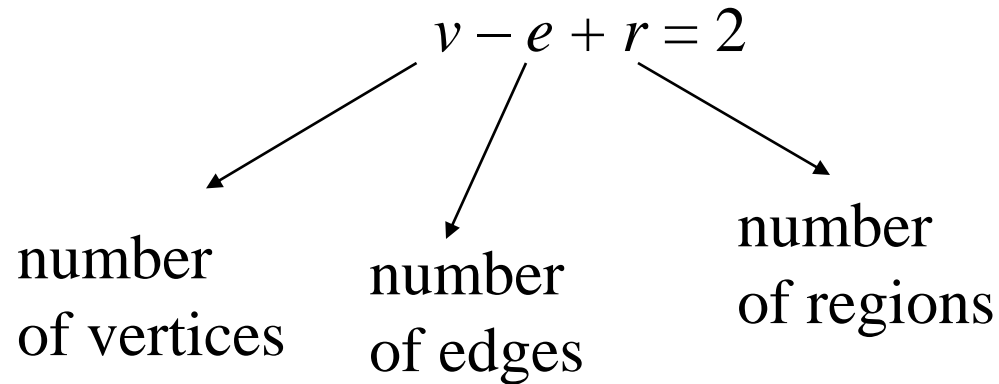




Planar Graphs

Theorem : *Euler's planar graph theorem*

For a **connected** planar graph or multigraph:

$$v - e + r = 2$$


number
of vertices

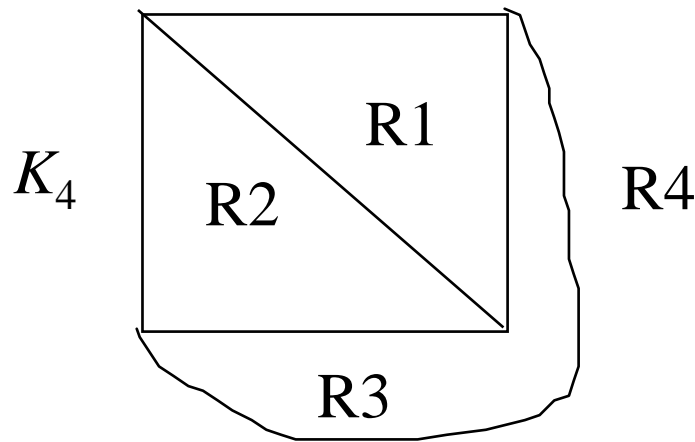
number
of edges

number
of regions



Planar Graphs

Example of Euler's theorem



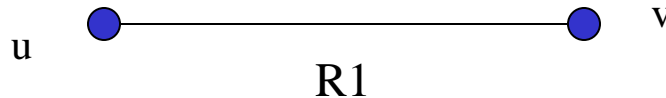
A planar graph divides the plane into several regions (faces), one of them is the infinite region.

$$v=4, e=6, r=4, v-e+r=2$$

Planar Graphs

- Proof of Euler's formula: By Induction

Base Case: for G_1 , $e_1 = 1$, $v_1 = 2$ and $r_1 = 1$

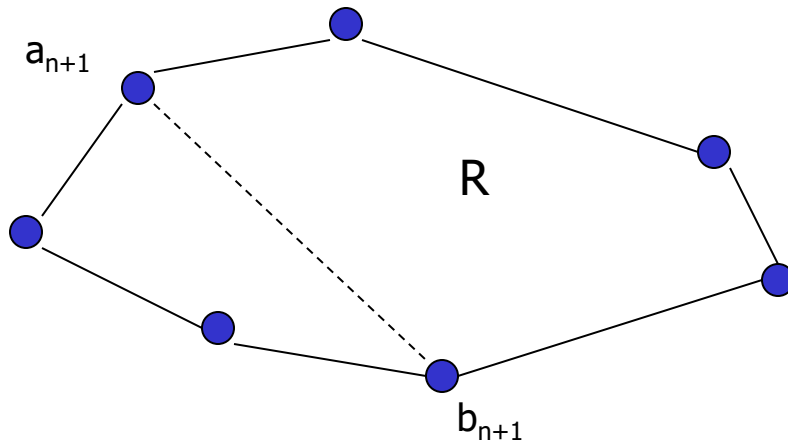


$n+1$ Case: Assume, $r_n = e_n - v_n + 2$ is true. Let $\{a_{n+1}, b_{n+1}\}$ be the edge that is added to G_n to obtain G_{n+1} and we prove that $r_n = e_n - v_n + 2$ is true. Can be proved using two cases.

Planar Graphs

- Case 1:

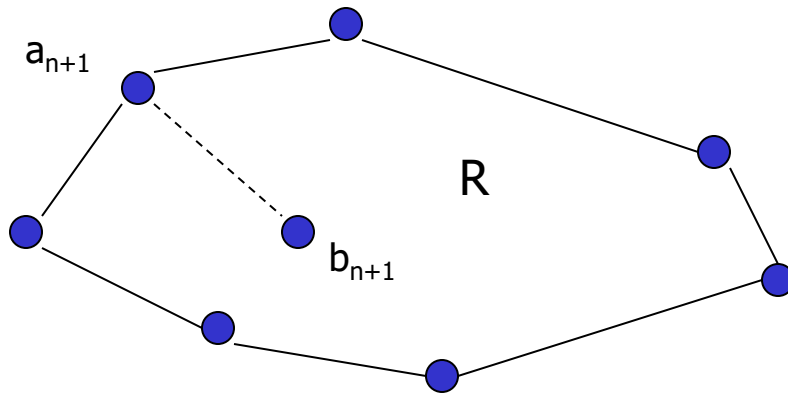
$$r_{n+1} = r_n + 1, e_{n+1} = e_n + 1, v_{n+1} = v_n \Rightarrow r_{n+1} = e_{n+1} - v_{n+1} + 2$$



Planar Graphs

- Case 2:

$$r_{n+1} = r_n, e_{n+1} = e_n + 1, v_{n+1} = v_n + 1 \Rightarrow r_{n+1} = e_{n+1} - v_{n+1} + 2$$



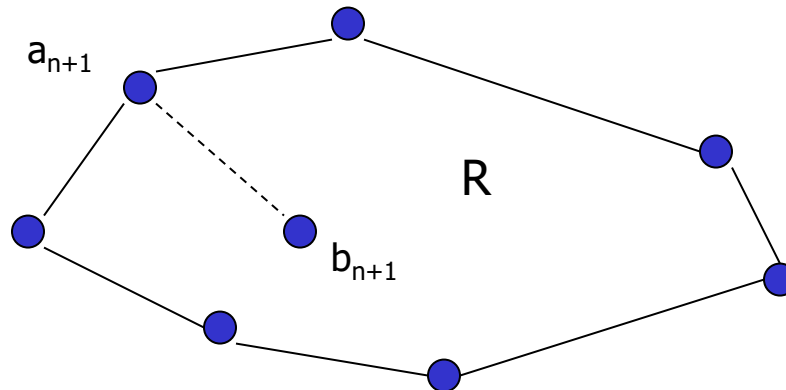
Planar Graphs

Corollary 1: Let $G = (V, E)$ be a connected simple planar graph with $|V| = v$, $|E| = e > 2$, and r regions. Then $3r \leq 2e$ and $e \leq 3v - 6$

Proof: Since G is loop-free and is not a multigraph, the boundary of each region (including the infinite region) contains at least three edges. Hence, each region has degree ≥ 3 .

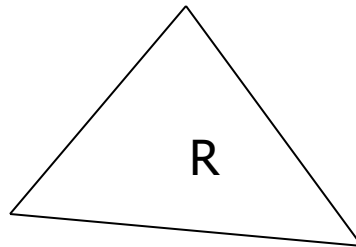
Degree of region: No. of edges on its boundary; 1 edge may occur twice on boundary \rightarrow contributes 2 to the region degree.

Each edge occurs exactly twice: either in the same region or in 2 different regions

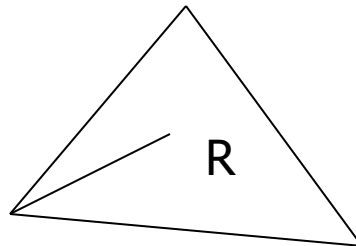




Region Degree



Degree of R = 3



Degree of R = ?



Planar Graphs

Each edge occurs exactly twice: either in the same region or in 2 different regions

$\Rightarrow 2e = \text{sum of degree of } r \text{ regions determined by } 2e$

$\Rightarrow 2e \geq 3r$. (since each region has a degree of at least 3)

$\Rightarrow r \leq (2/3) e$

$\Rightarrow \text{From Euler's theorem, } 2 = v - e + r$

$\Rightarrow 2 \leq v - e + 2e/3$

$\Rightarrow 2 \leq v - e/3$

$\Rightarrow \text{So } 6 \leq 3v - e$

$\Rightarrow \text{or } e \leq 3v - 6$



Planar Graphs

Corollary 2: Let $G = (V, E)$ be a connected simple planar graph then G has a vertex degree that does not exceed 5

Proof: If G has one or two vertices the result is true

If G has 3 or more vertices then by Corollary 1, $e \leq 3v - 6$

$$\Rightarrow 2e \leq 6v - 12$$

If the degree of every vertex were at least 6:

by Handshaking theorem: $2e = \text{Sum}(\text{deg}(v))$

$$\Rightarrow 2e \geq 6v. \text{ But this contradicts the inequality } 2e \leq 6v - 12$$

\Rightarrow There must be at least one vertex with degree no greater than 5



Planar Graphs

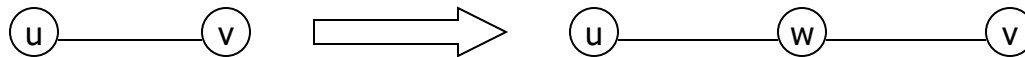
Corollary 3: Let $G = (V, E)$ be a connected simple planar graph with v vertices ($v \geq 3$), e edges, and no circuits of length 3 then $e \leq 2v - 4$

Proof: Similar to Corollary 1 except the fact that no circuits of length 3 imply that degree of region must be at least 4.



Planar Graphs

- **Elementary sub-division:** Operation in which a graph are obtained by removing an edge $\{u, v\}$ and adding the vertex w and edges $\{u, w\}, \{w, v\}$

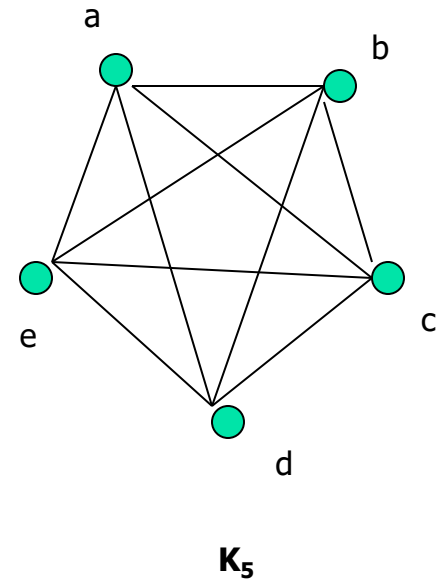
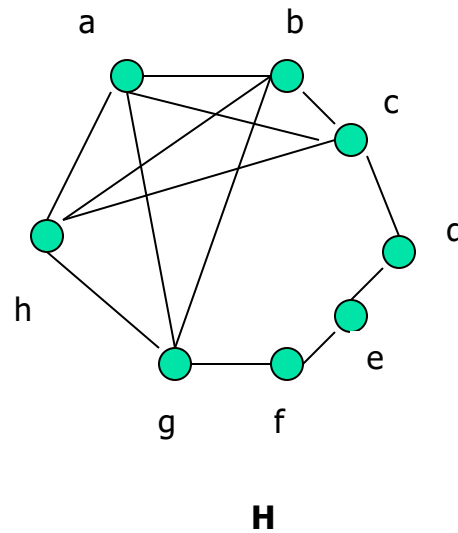
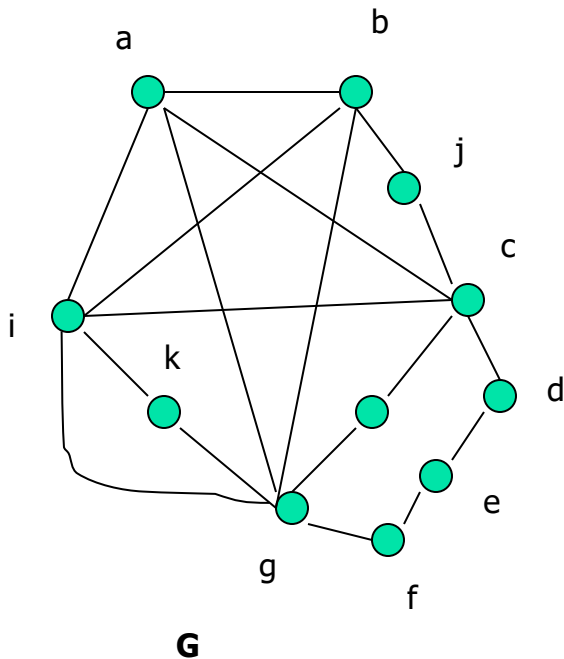


- **Homeomorphic Graphs:** Graphs G_1 and G_2 are termed as homeomorphic if they are obtained by sequence of elementary sub-divisions.

Planar Graphs

- **Kuwratoski's Theorem:** A graph is non-planar if and only if it contains a subgraph homeomorphic to $K_{3,3}$ or K_5

Representation Example: G is Nonplanar



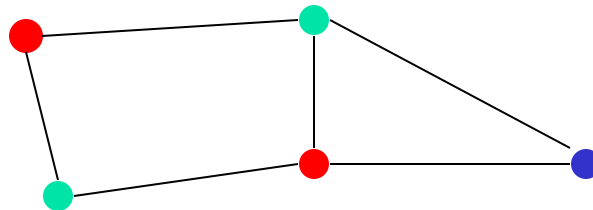


Graph Coloring Problem

- **Graph coloring** is an assignment of "*colors*", almost always taken to be consecutive integers starting from 1 without loss of generality, to certain objects in a graph. Such objects can be vertices, edges, faces, or a mixture of the above.
- Application examples: scheduling, register allocation in a microprocessor, frequency assignment in mobile radios, and pattern matching

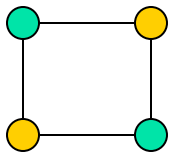
Vertex Coloring Problem

- Assignment of colors to the vertices of the graph such that proper coloring takes place (no two adjacent vertices are assigned the same color)
- **Chromatic number:** least number of colors needed to color the graph
- A graph that can be assigned a (proper) k -coloring is **k -colorable**, and it is **k -chromatic** if its chromatic number is exactly k .

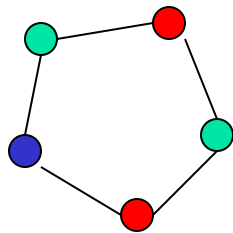


Vertex Coloring Problem

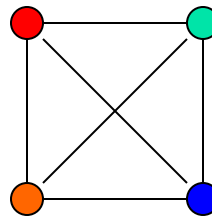
- The problem of finding a minimum coloring of a graph is NP-Hard
- The corresponding decision problem (Is there a coloring which uses at most k colors?) is NP-complete
- The chromatic number for $C_n = 3$ (n is odd) or 2 (n is even), $K_n = n$, $K_{m,n} = 2$
- C_n : cycle with n vertices; K_n : fully connected graph with n vertices; $K_{m,n}$: complete bipartite graph



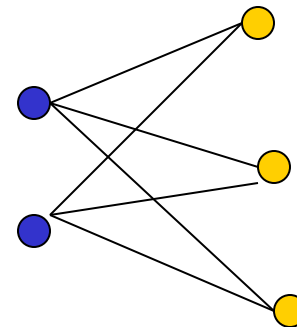
C_4



C_5



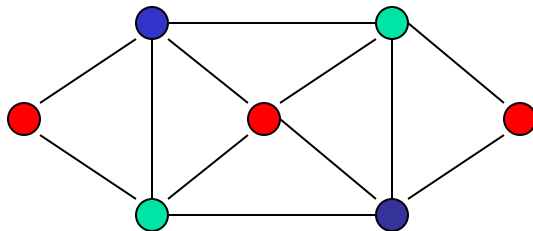
K_4



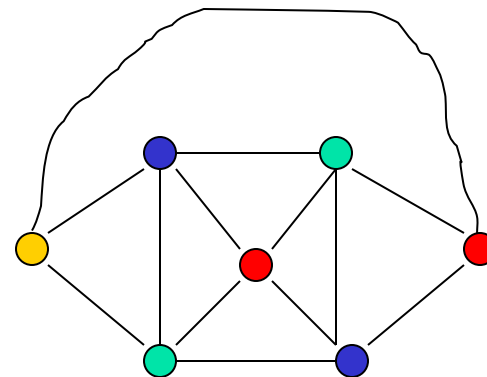
$K_{2,3}$

Vertex Covering Problem

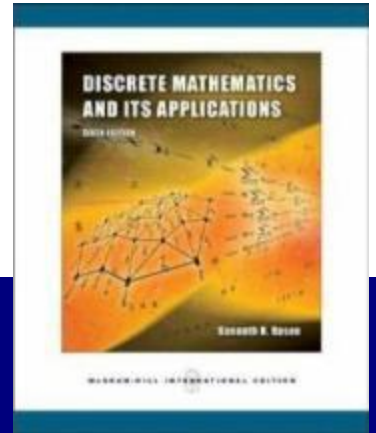
- **The Four color theorem:** the chromatic number of a planar graph is no greater than 4
- Example: G1 chromatic number = 3, G2 chromatic number = 4
- (Most proofs rely on case by case analysis).



G1



G2



Discrete Mathematics

Chapter 10

Trees



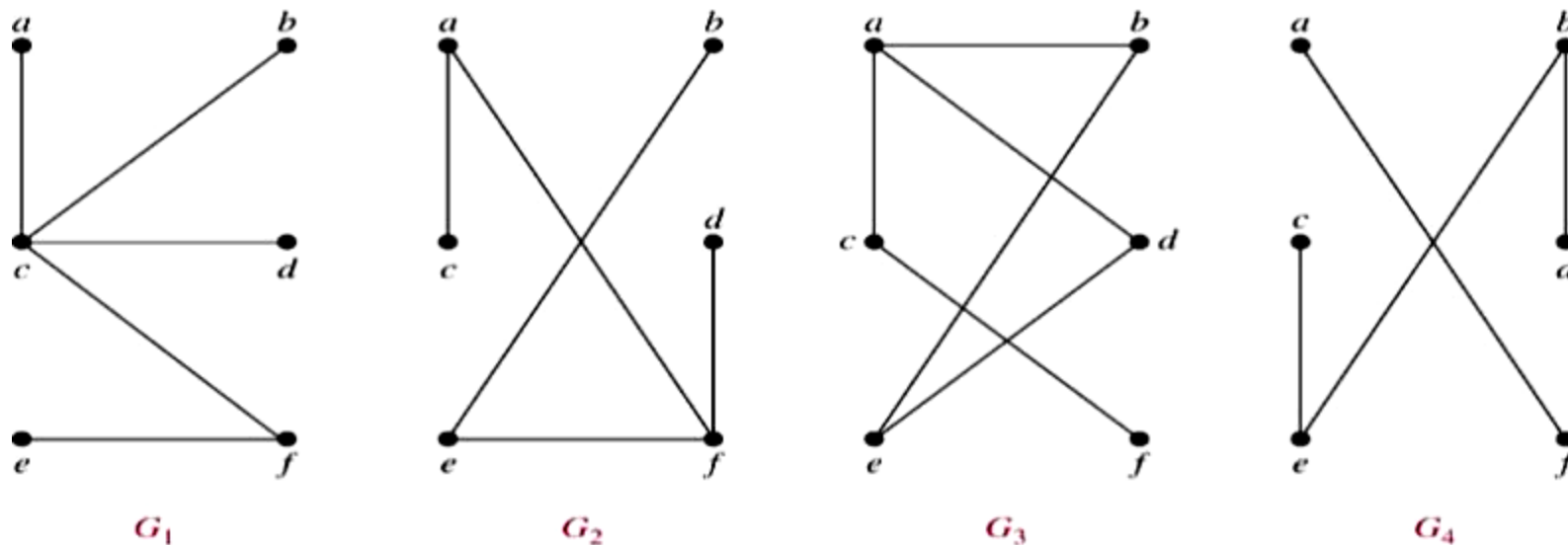
Outline

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

10.1 Introduction to Trees

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

Example 1. Which of the graphs are trees?

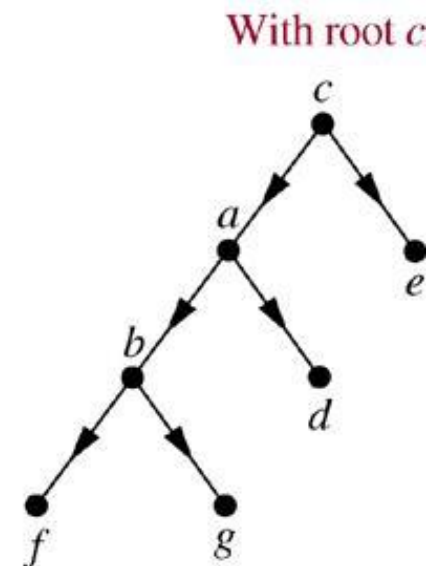
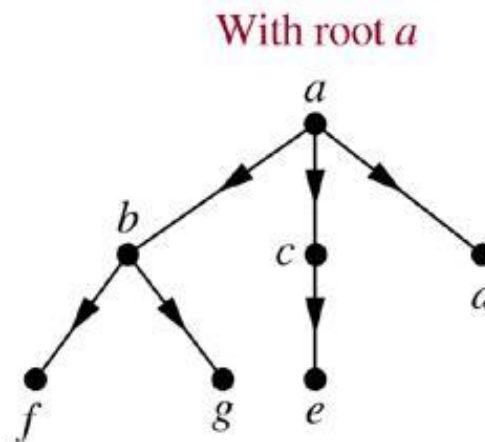
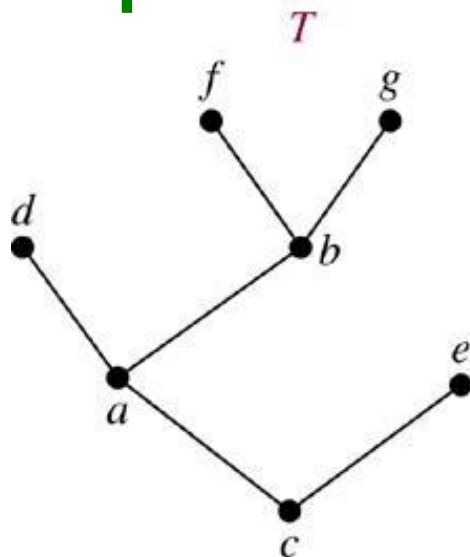


Sol: G_1, G_2

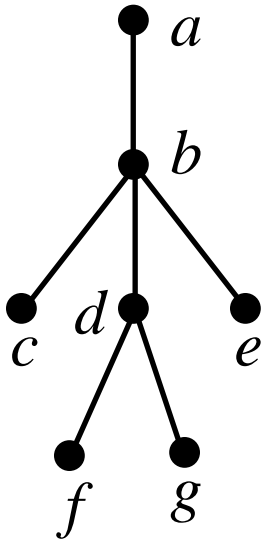
Thm 1. Any undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

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

Example



Def:



a is the **parent** of b , b is the **child** of a ,
 c, d, e are **siblings**,

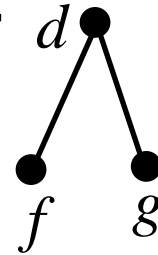
a, b, d are **ancestors** of f

c, d, e, f, g are **descendants** of b

c, e, f, g are **leaves** of the tree (deg=1)

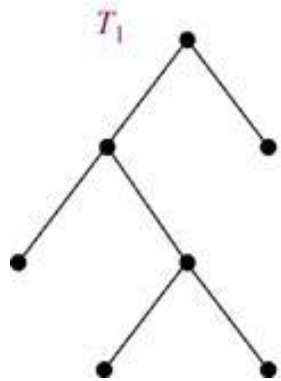
a, b, d are **internal vertices** of the tree
(at least one child)

subtree with d as its root:

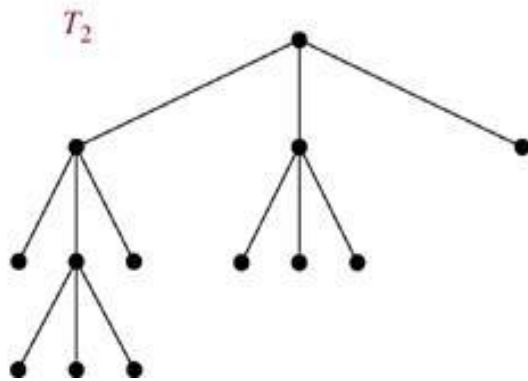


Def 3 A rooted tree is called an m -ary tree if every internal vertex has no more than m children. The tree is called a **full m -ary tree** if every internal vertex has exactly m children. An m -ary tree with $m=2$ is called a **binary tree**.

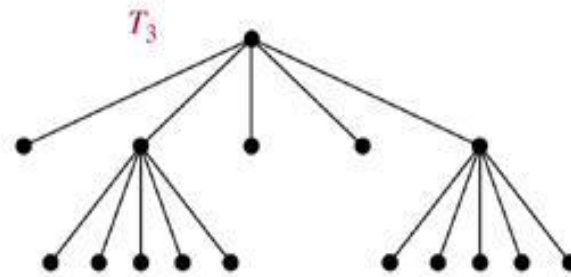
Example 3



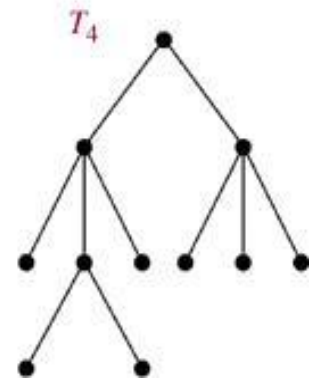
full binary tree



full 3-ary tree

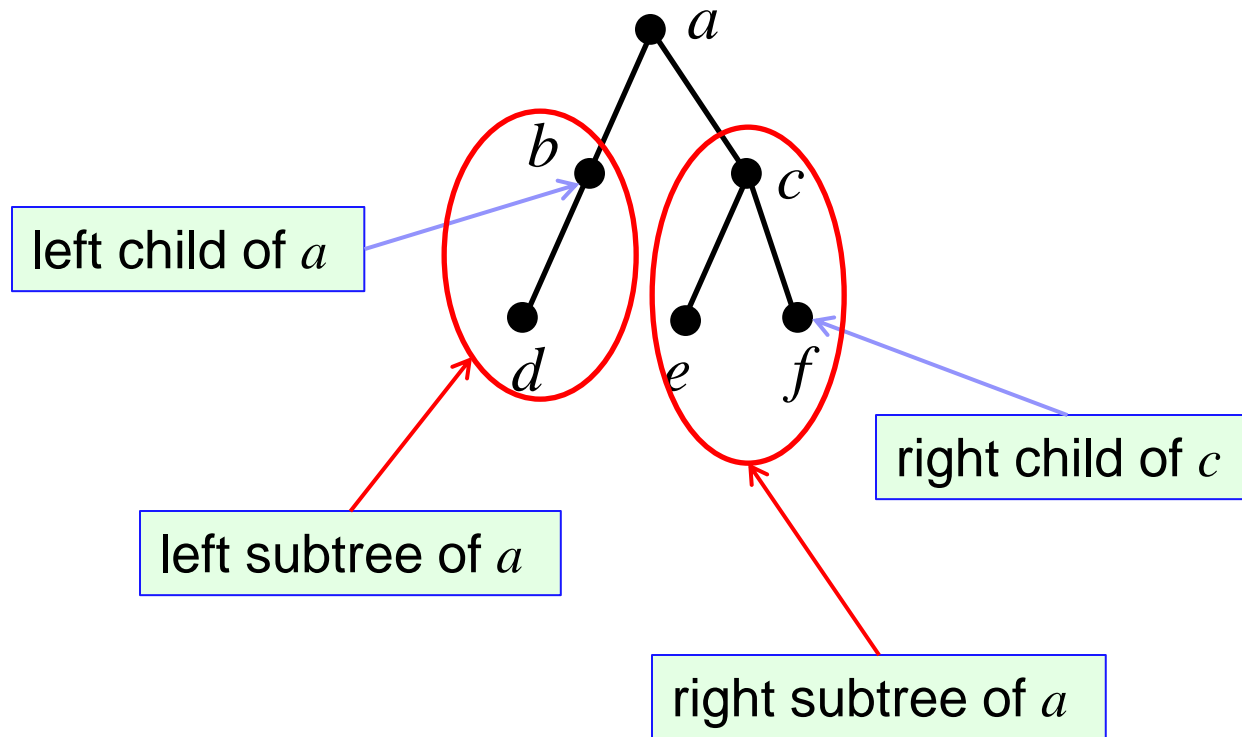


full 5-ary tree



not full 3-ary tree

Def:



Properties of Trees

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

Pf. (by induction on n)

$n = 1$: K_1 is the only tree of order 1, $|E(K_1)| = 0$. **ok!**

Assume the result is true for every trees of order $n = k$.

Let T be a tree of order $n = k+1$, v be a leaf of T ,
and w be the parent of v .

Let T' be the tree $T - \{v\}$.

$\therefore |V(T')| = k$, and $|E(T')| = k-1$ by the induction hypothesis.

$\Rightarrow |E(T)| = k$

By induction, the result is true for all trees. #

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

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

Each internal vertex has m children.

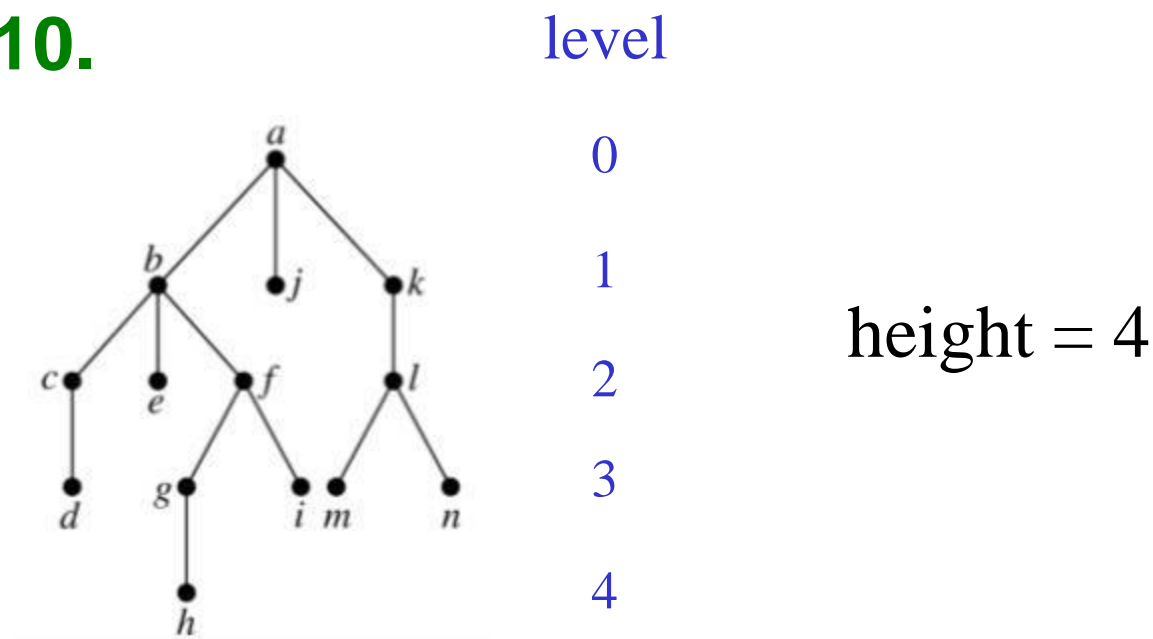
\Rightarrow there are $mi + 1$ vertices in the tree

Exercise: 19

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

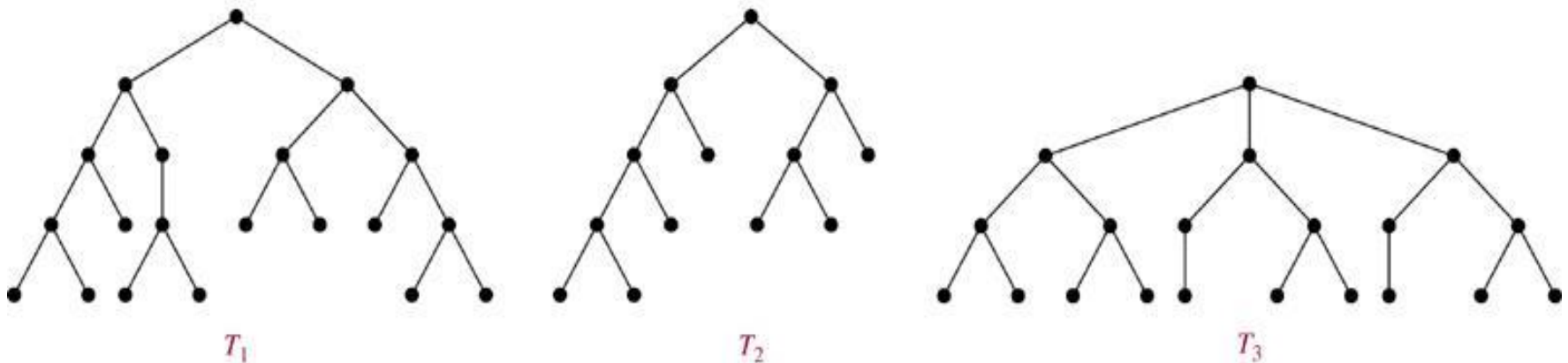
Def: The **level** of a vertex v in a rooted tree is the length of the unique path from the root to this vertex. The level of the root is defined to be zero. The **height** of a rooted tree is the maximum of the levels of vertices.

Example 10.



Def: A rooted m -ary tree of height h is **balanced** if all leaves are at levels h or $h-1$.

Example 11 Which of the rooted trees shown below are balanced?



Sol. T_1, T_3

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

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

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

Sol.

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

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

10.2 Applications of Trees

Binary Search Trees

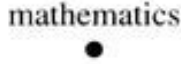
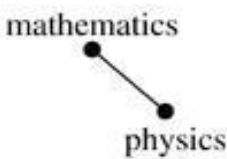
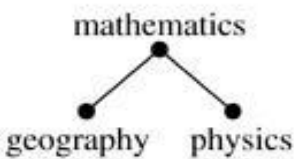
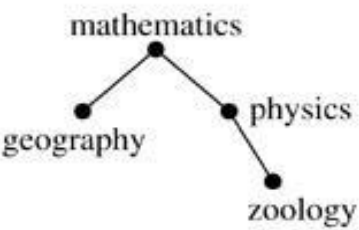
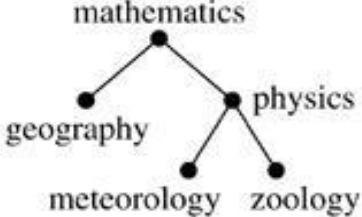
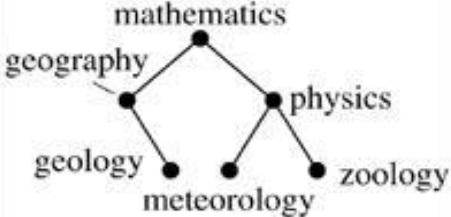
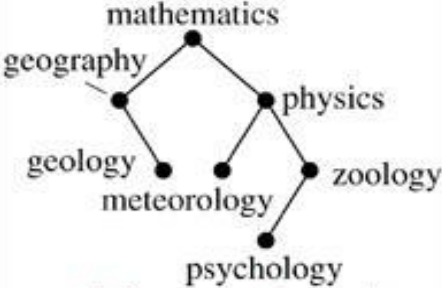
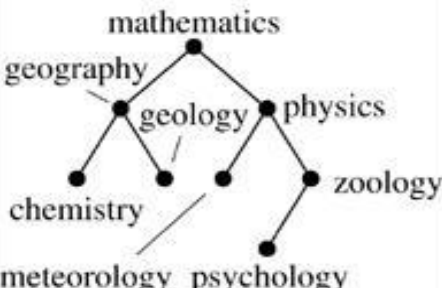
Goal: Implement a searching algorithm that finds items efficiently when the items are totally ordered.

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

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

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

Sol.

	 <p>physics > mathematics</p>	 <p>geography < mathematics</p>	 <p>zoology > mathematics zoology > physics</p>
 <p>meteorology > mathematics meteorology < physics</p>	 <p>geology < mathematics geology > geography</p>	 <p>psychology > mathematics psychology > physics psychology < zoology</p>	 <p>chemistry < mathematics chemistry < geography</p>

Algorithm 1 (Locating and Adding Items to a Binary Search Tree.)

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

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

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

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

begin

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

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

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

else

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

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

end

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

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

 let v be this new vertex

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

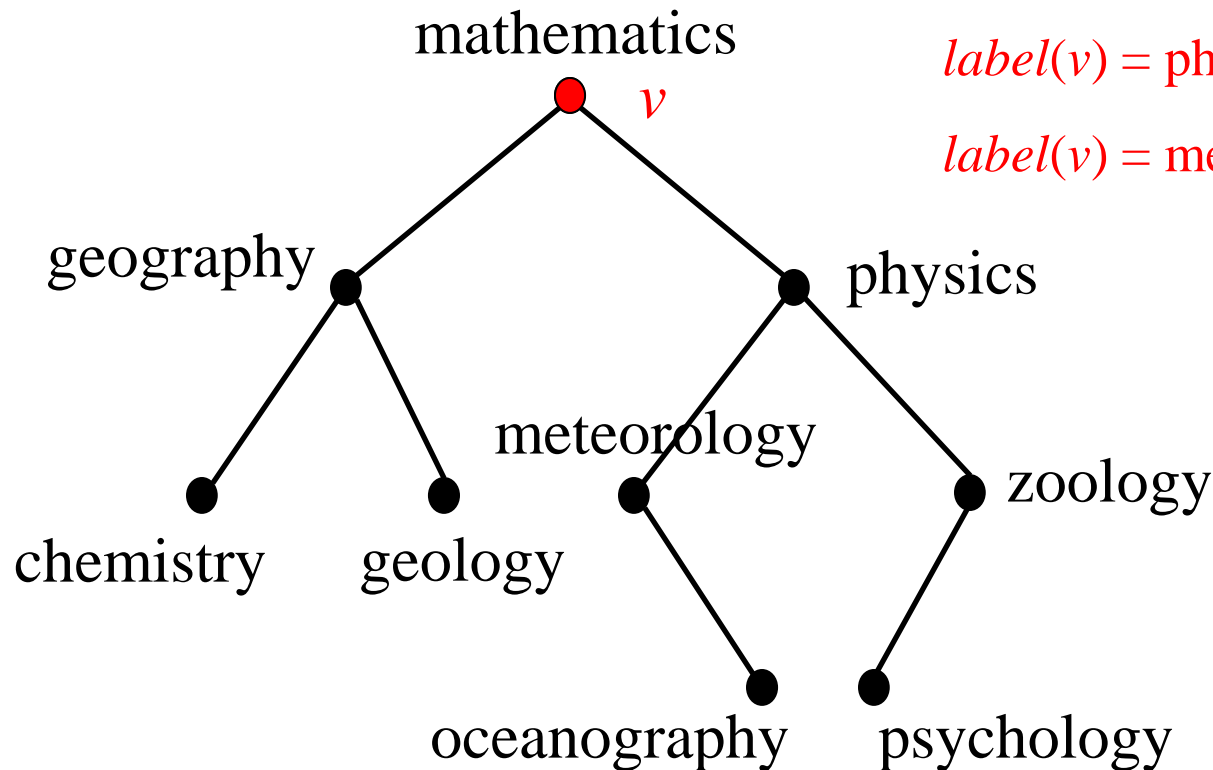
Example 2 Use Algorithm 1 to insert the word *oceanography* into the binary search tree in Example 1.

Sol.

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

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

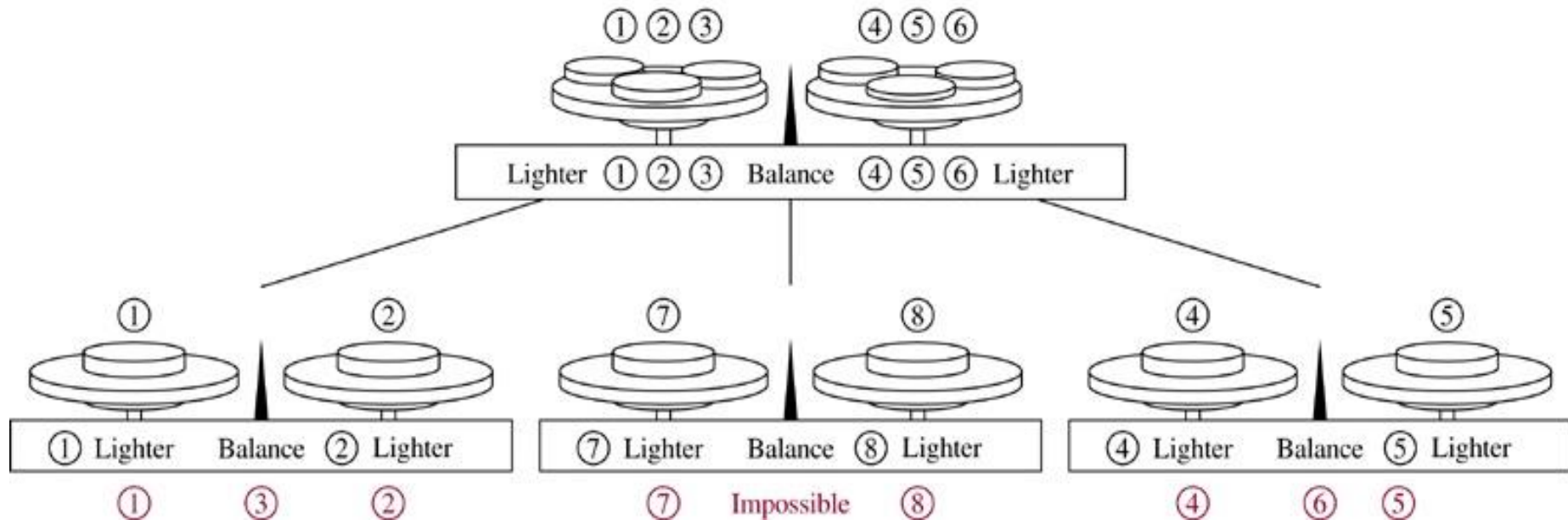
$label(v) = \text{meteorology} < \text{oceanography}$



Exercise: 1,3

Sol. \Rightarrow 3-ary tree

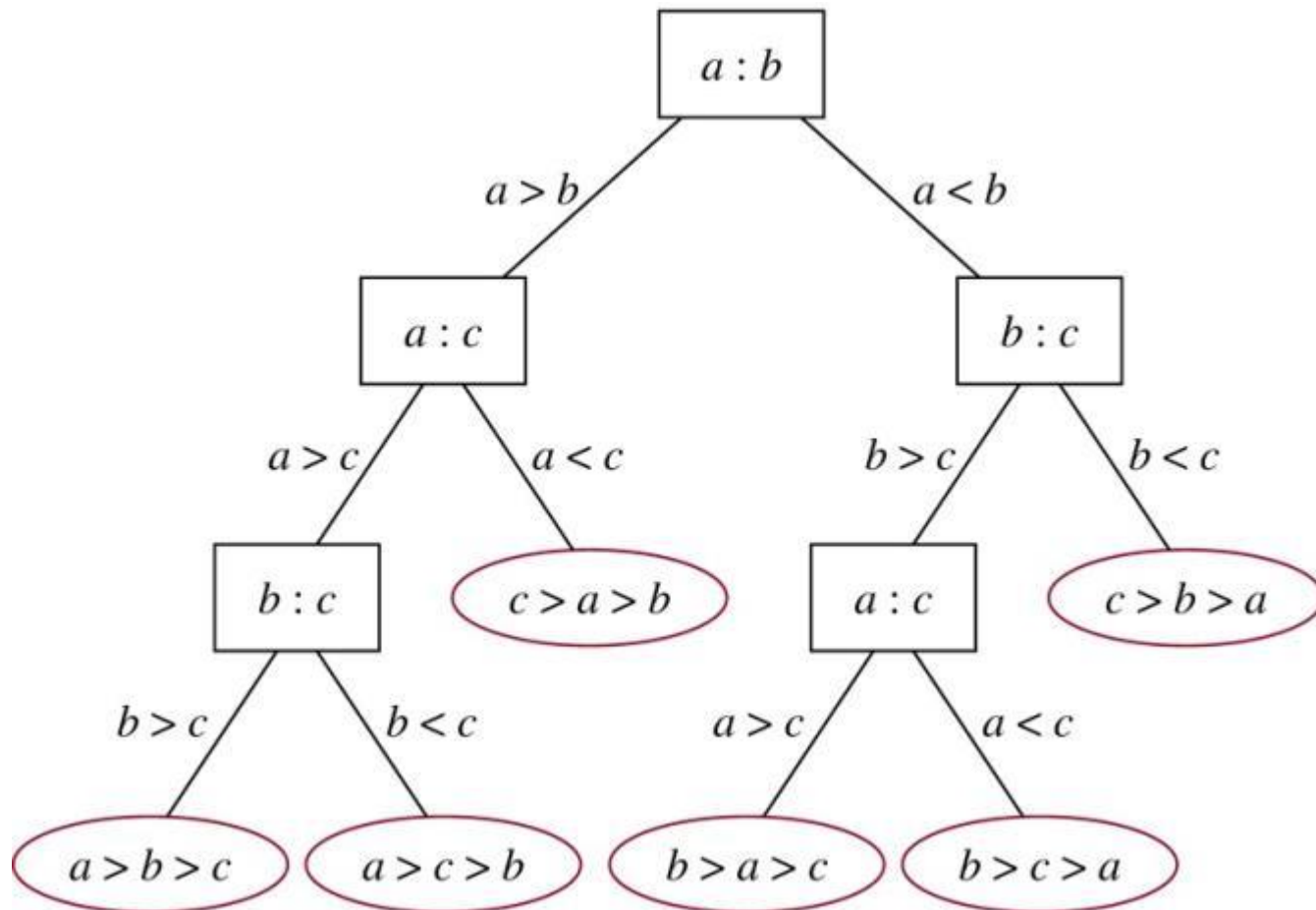
Need 8 leaves \Rightarrow



Exercise: 7

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

Sol.



Prefix Codes

Problem: Using bit strings to encode the letter of the English alphabet

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

Example: (not prefix code)

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

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

Example: (prefix code)

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

The string 10110 is the encoding of *ate*.

The bit string used to encode a character is the sequence of labels of the edges in the unique path from the root to the leaf that has this character as its label.

Ch10-21



Huffman Coding (data compression)

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

Algorithm 2 (Huffman Coding)

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

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

while F is not a tree

begin

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

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

end

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

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

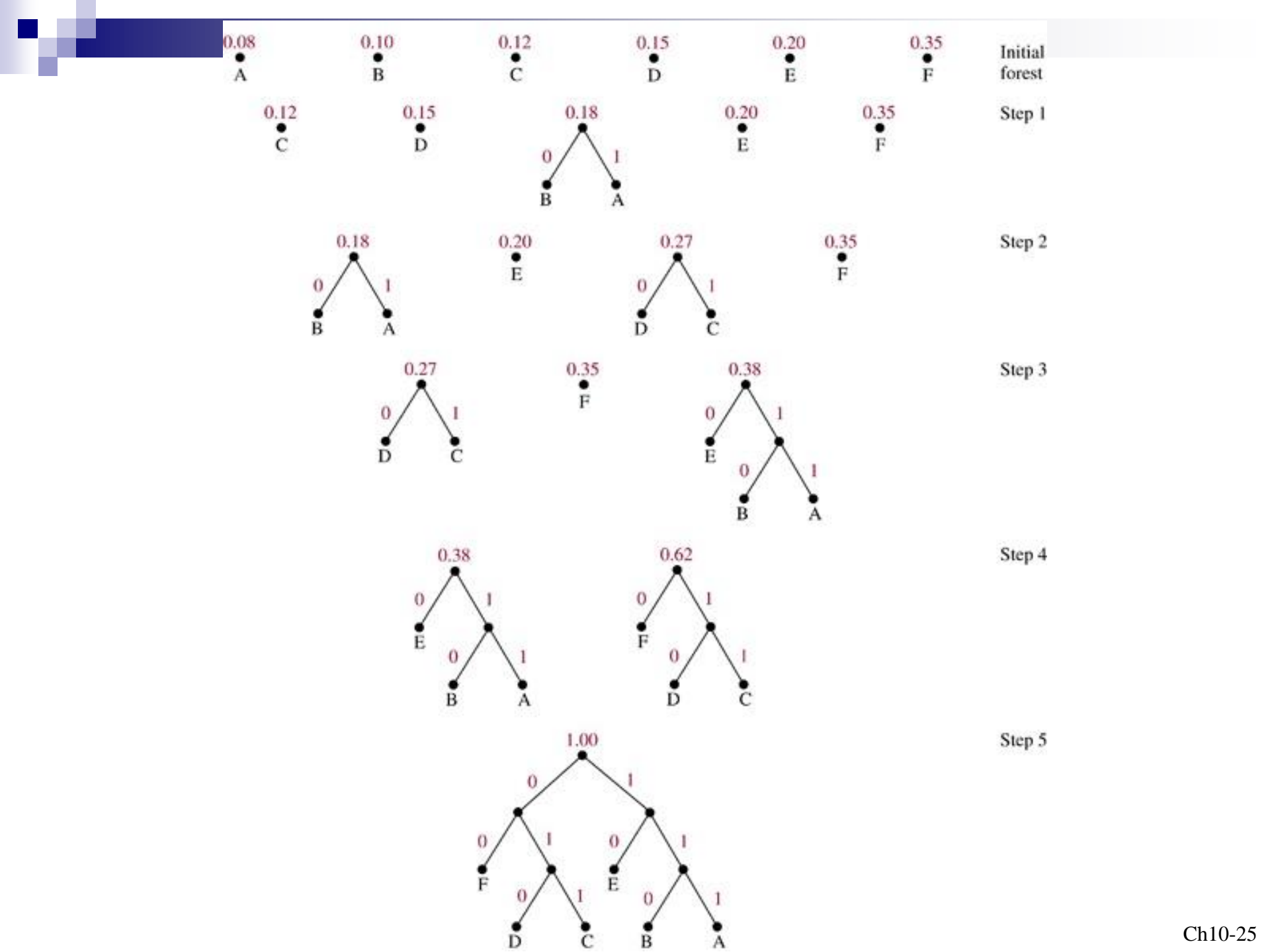
What is the average number of bits used to encode a character?

Sol:

1.

2. The average number of bits is:

$$\begin{aligned} &= 3 \times 0.08 + 3 \times 0.10 + 3 \times 0.12 + 3 \times 0.15 + 2 \times 0.20 + 2 \times 0.35 \\ &= 2.45 \end{aligned}$$



10.3 Tree Traversal

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

Universal Address Systems

Label vertices:

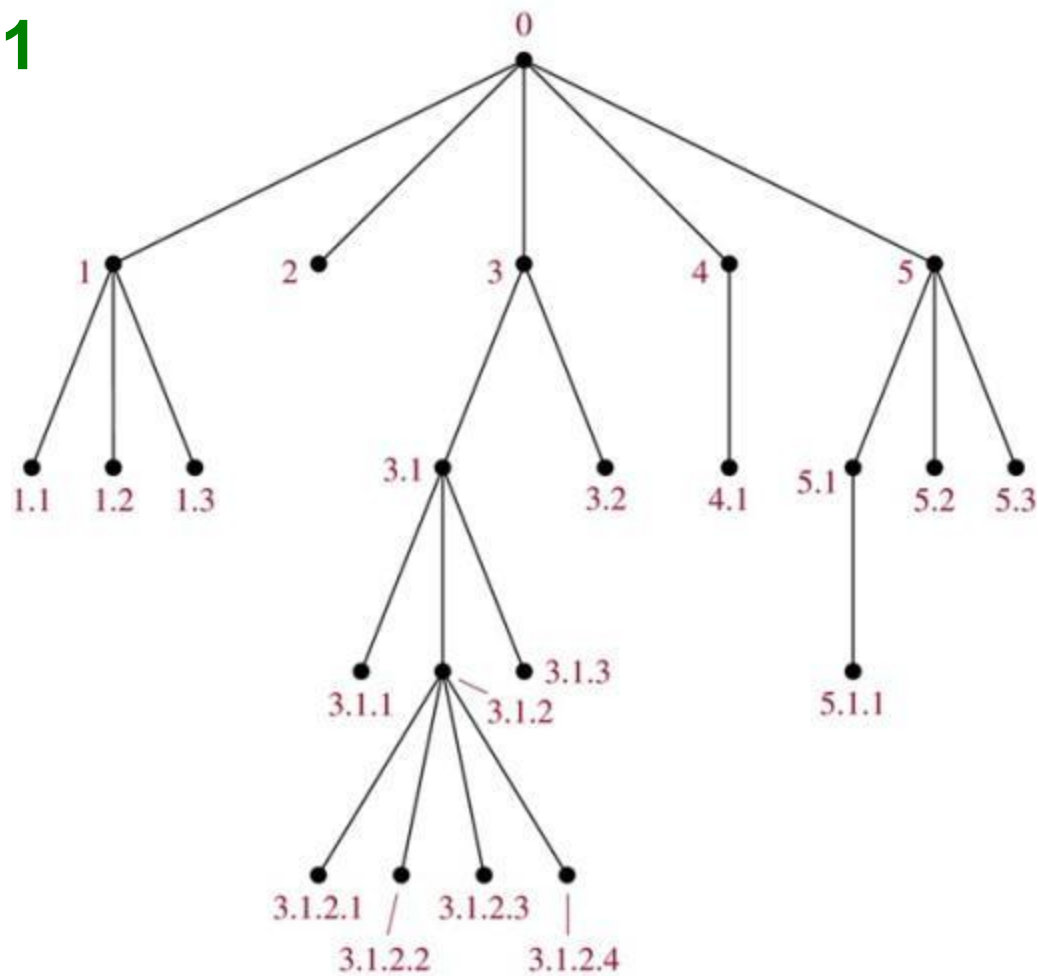
1. root $\rightarrow 0$, its k children $\rightarrow 1, 2, \dots, k$ (from left to right)
2. For each vertex v at level n with label A , its r children $\rightarrow A.1, A.2, \dots, A.r$ (from left to right).

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

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

if there is an i , $0 \leq i \leq n$, with $x_1=y_1, x_2=y_2, \dots, x_{i-1}=y_{i-1}$, and $x_i < y_i$;
or if $n < m$ and $x_i=y_i$ for $i=1, 2, \dots, n$.

Example 1

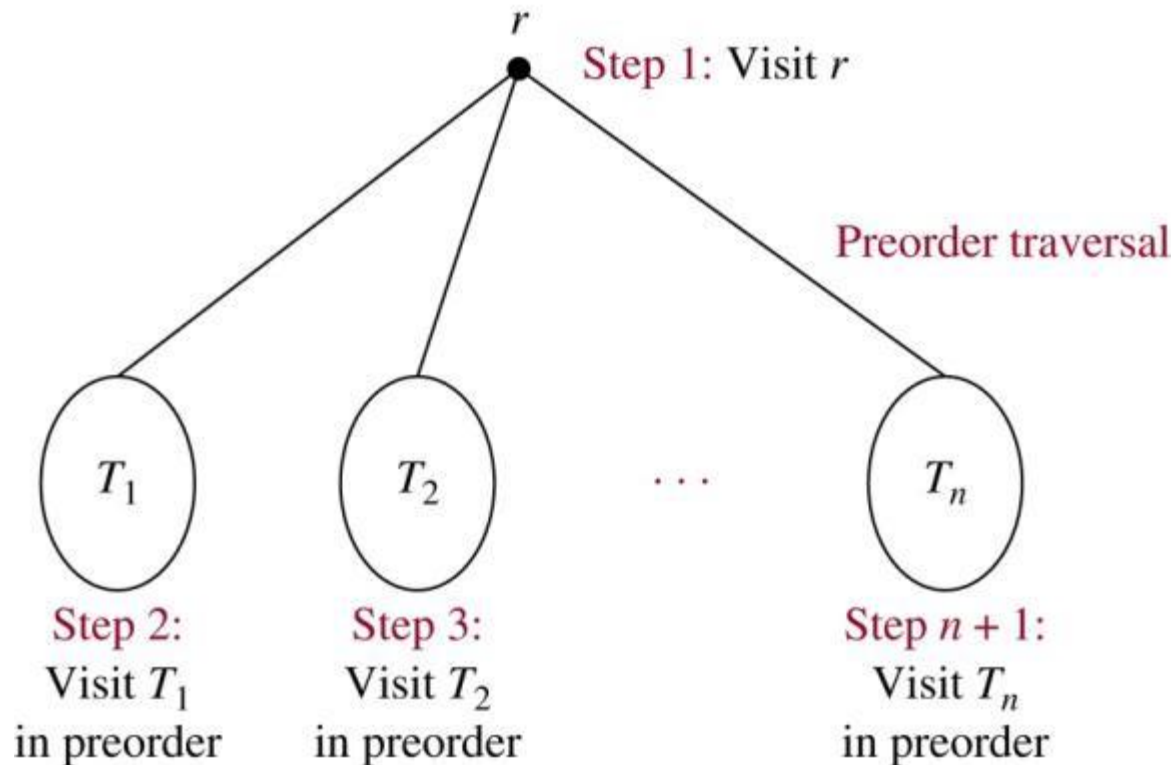


The lexicographic ordering is:

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

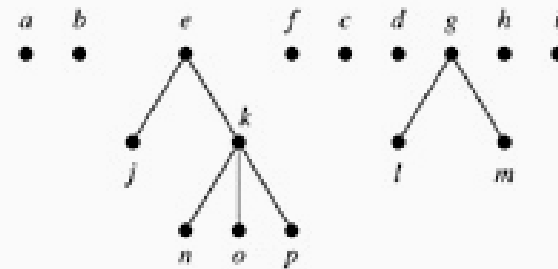
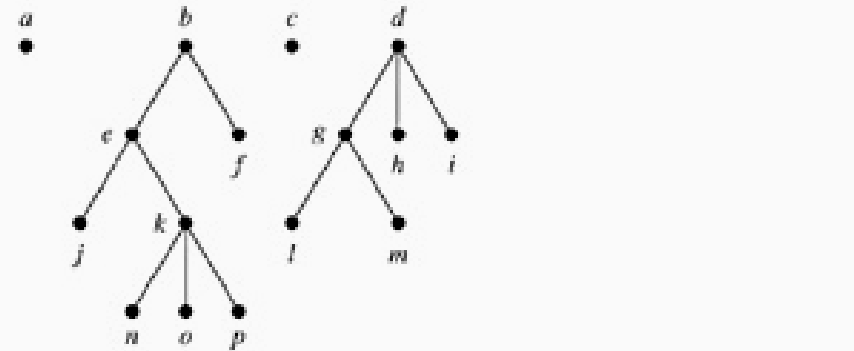
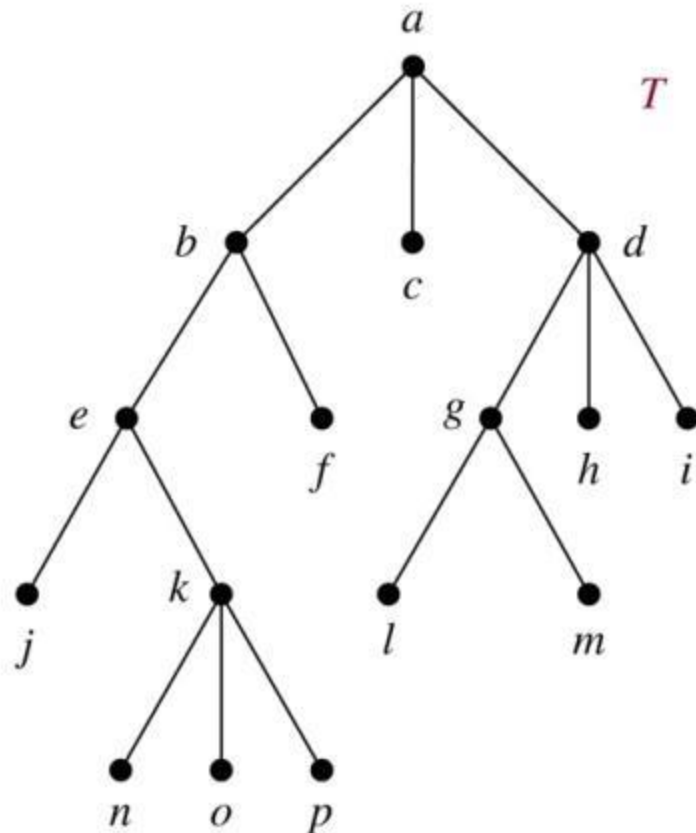
Traversal Algorithms

Preorder traversal (前序)



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

Sol:

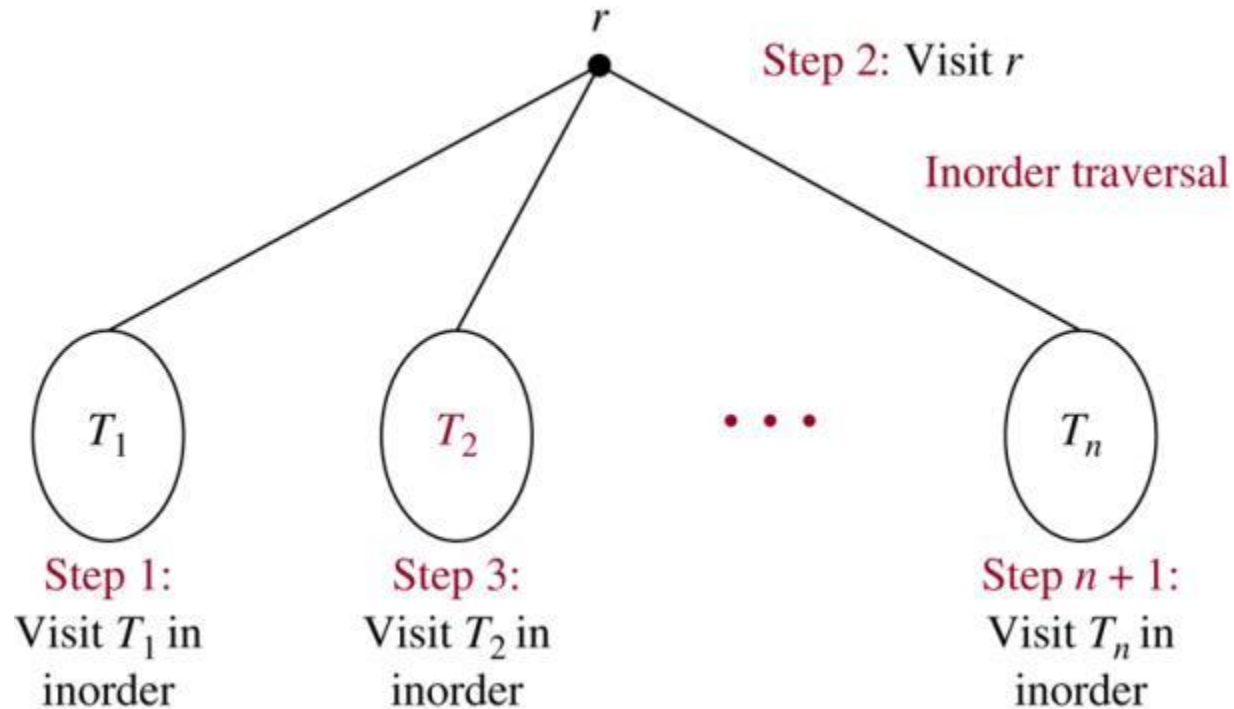


Algorithm 1 (Preorder Traversal)

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

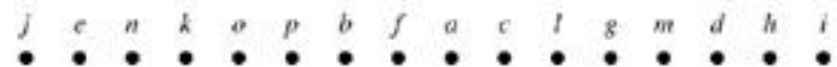
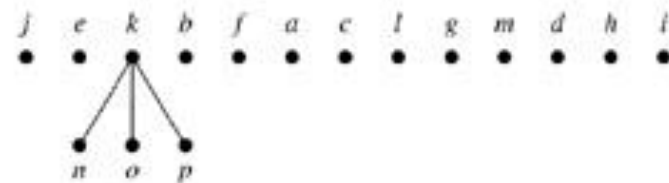
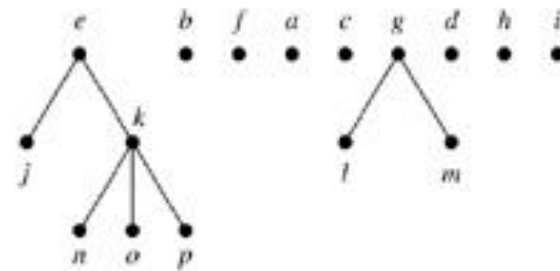
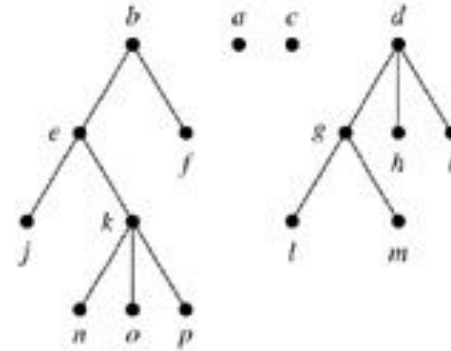
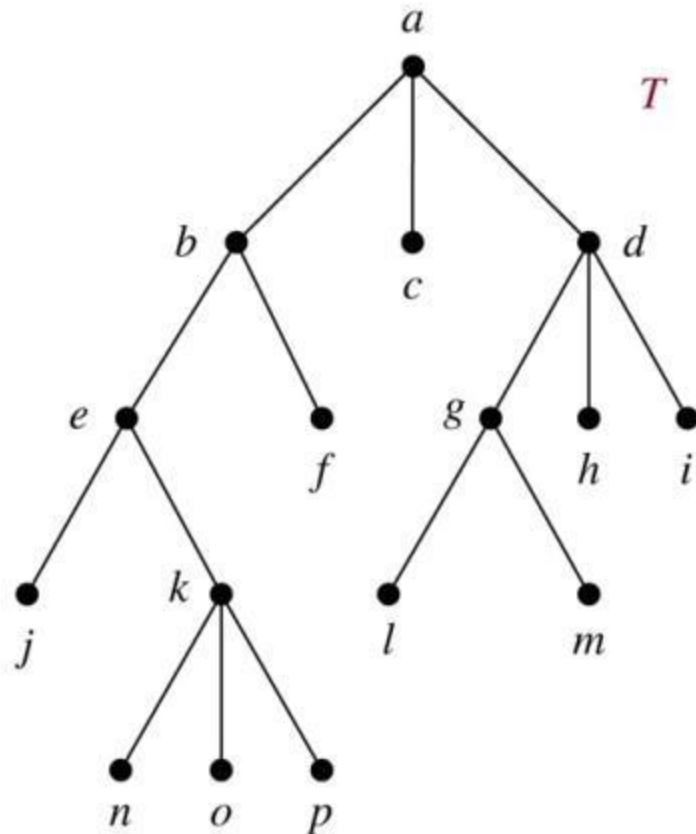
Exercise : 8

Inorder traversal



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

Sol:



Algorithm 2 (Inorder Traversal)

Procedure *inorder*(T : ordered rooted tree)

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

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

else

begin

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

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

inorder($T(l)$)

list r

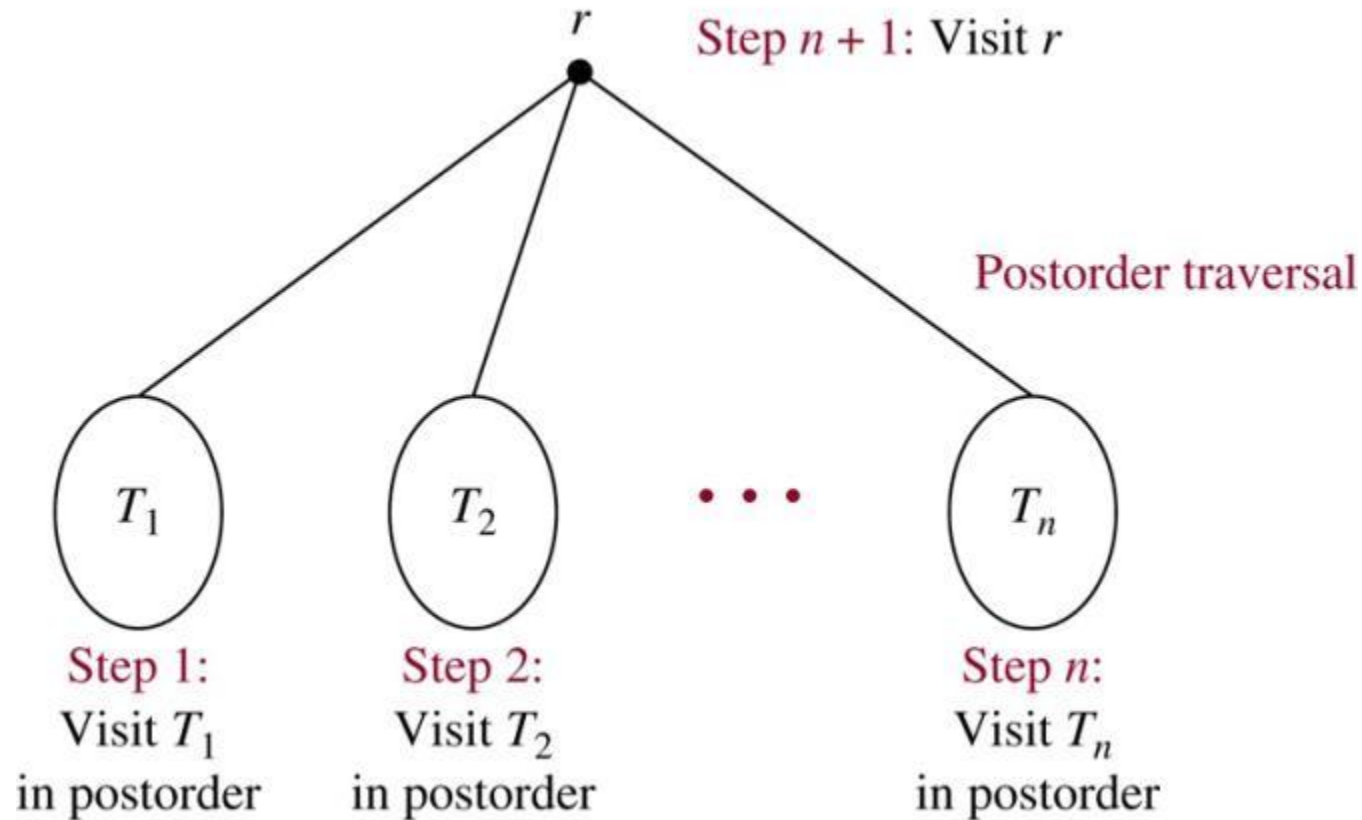
for each child c of r except for l from left to right

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

inorder($T(c)$)

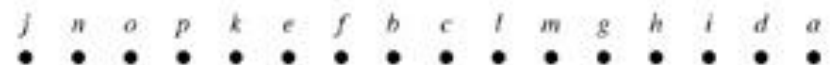
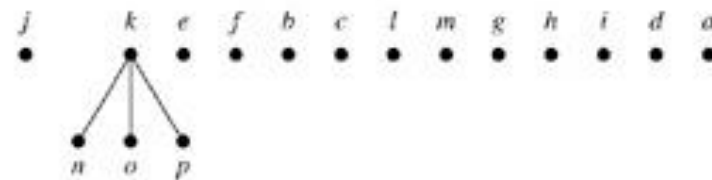
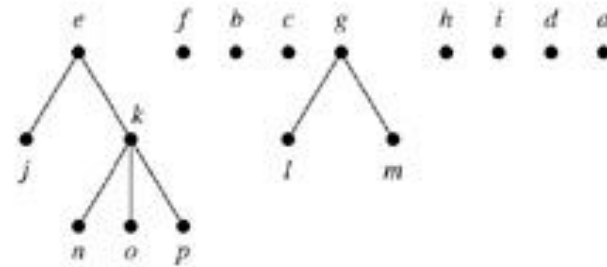
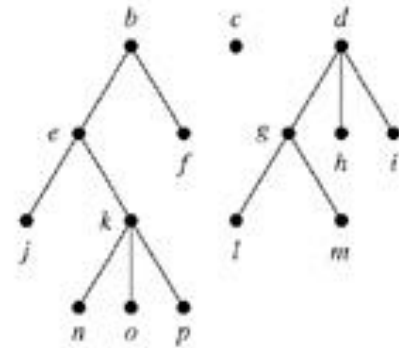
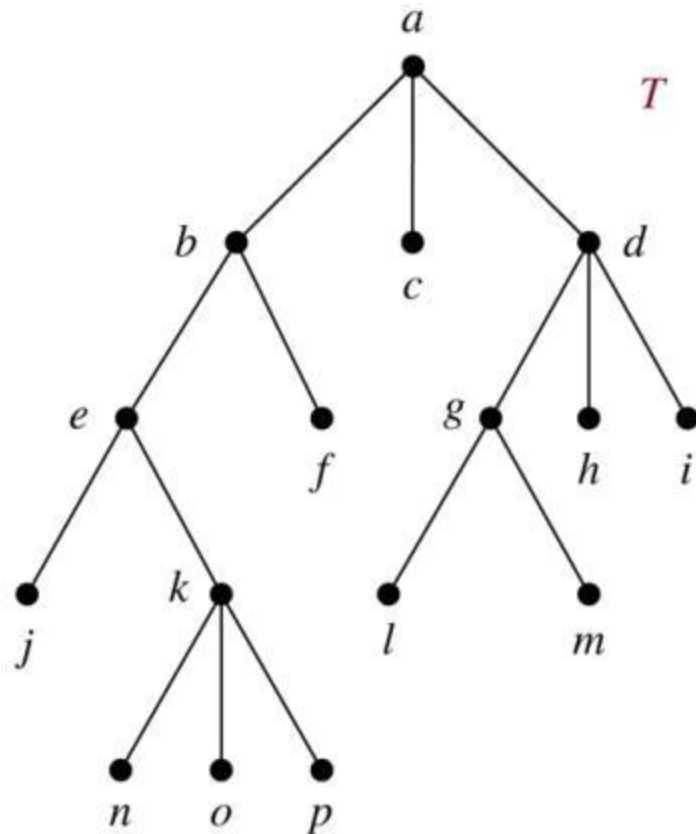
end

Postorder traversal



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

Sol:



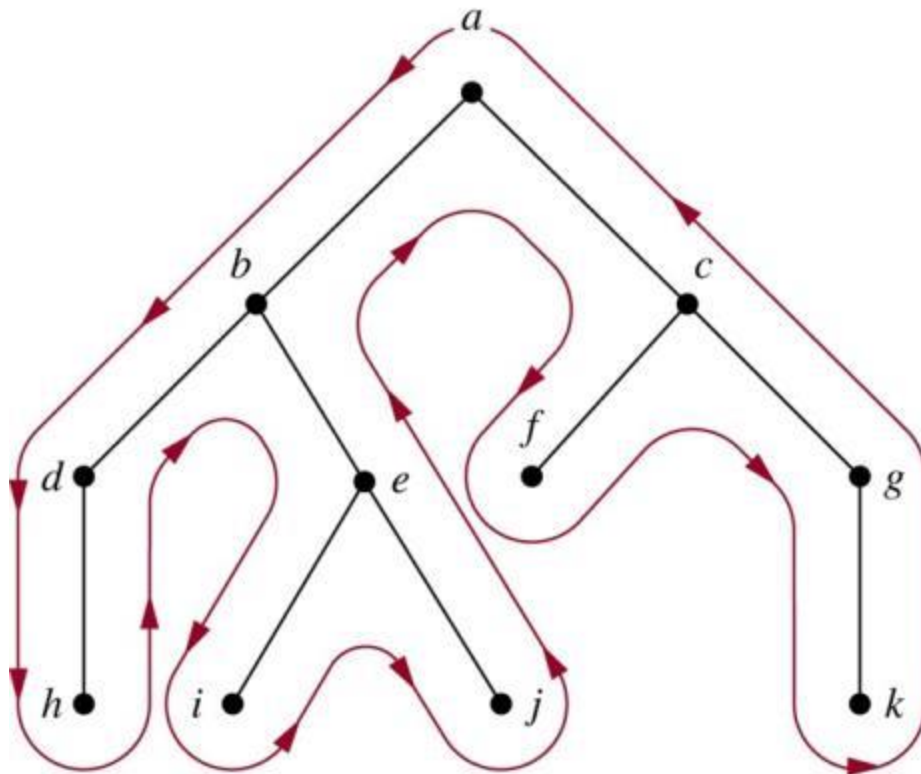
Algorithm 3 (Postorder Traversal)

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

Preorder: curve

Inorder: curve leaf · internal
list

Postorder: curve



Preorder:

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

Inorder:

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

Postorder:

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

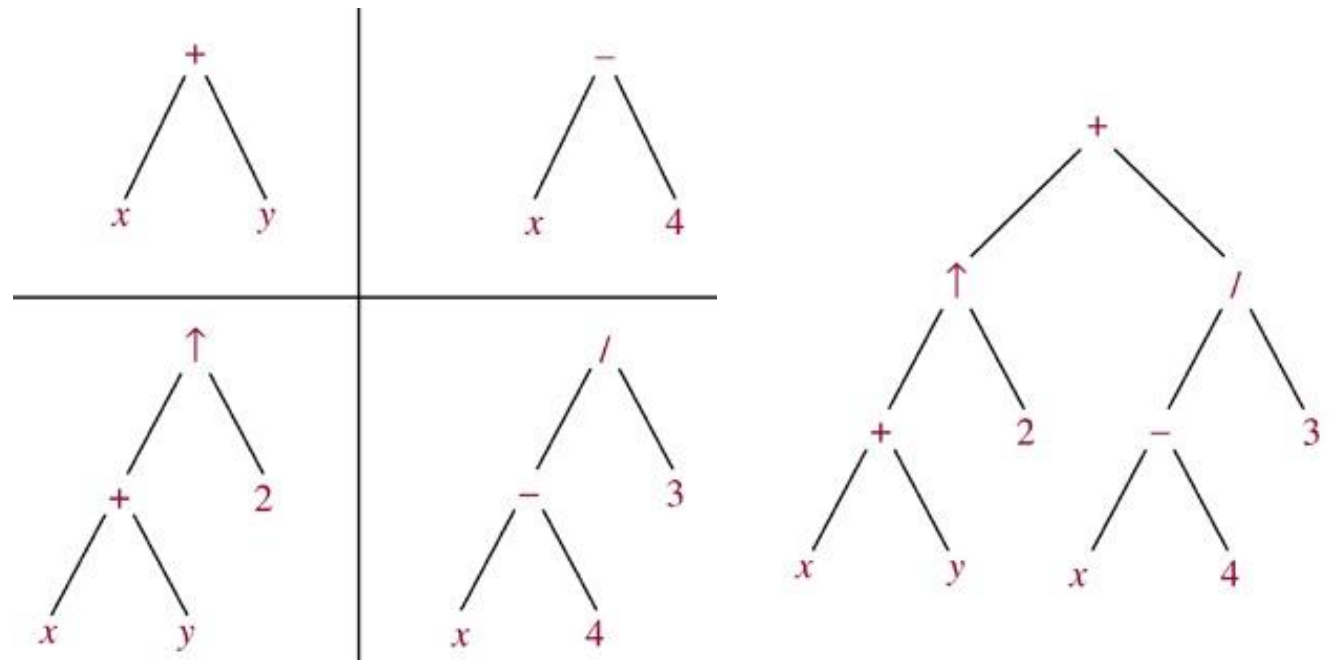
Infix, Prefix, and Postfix Notation

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

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

Sol.

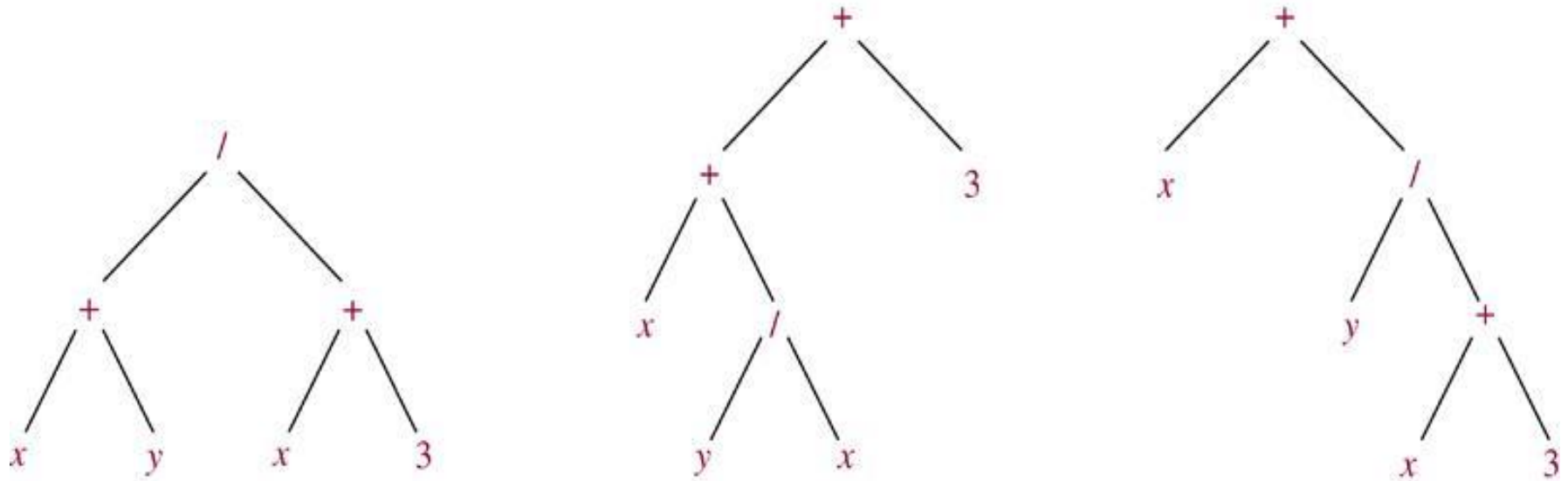
leaf:
variable
internal vertex:
operation on
its left and right
subtrees



The following binary trees represent the expressions:

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

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



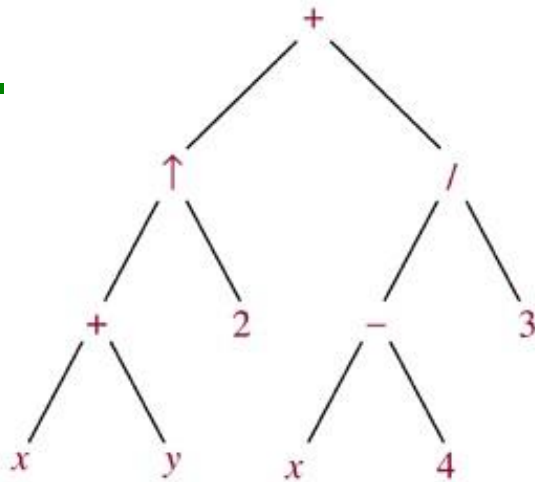
Infix form: An expression obtained when we traverse its rooted tree with inorder.

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

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

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

Sol.



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

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

Sol.

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

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

Example 7 What is the value of the prefix expression
 $+ - * 2 3 5 / \uparrow 2 3 4$?

Sol.

$$\begin{array}{ccccccccccc} + & - & * & 2 & 3 & 5 & / & \uparrow & 2 & 3 & 4 \\ & & & & & & & \underbrace{} & & & \\ & & & & & & & 2 \uparrow 3 = 8 & & & \end{array}$$

$$\begin{array}{ccccccccccc} + & - & * & 2 & 3 & 5 & / & 8 & 4 \\ & & & & & & & \underbrace{} & & & \\ & & & & & & & 8 / 4 = 2 & & & \end{array}$$

$$\begin{array}{ccccccc} + & - & * & 2 & 3 & 5 & 2 \\ & & \underbrace{} & & & & \\ & & 2 * 3 = 6 & & & & \end{array}$$

$$\begin{array}{ccccccc} + & - & 6 & 5 & 2 \\ & \underbrace{} & & & \\ & 6 - 5 = 1 & & & \end{array}$$

$$\begin{array}{ccc} + & 1 & 2 \\ \underbrace{} & & \\ 1 + 2 = 3 & & \end{array}$$

Value of expression: 3

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

Sol.

7 2 3 * - 4 ↑ 9 3 / +

$2 * 3 = 6$

7 6 - 4 ↑ 9 3 / +

$7 - 6 = 1$

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

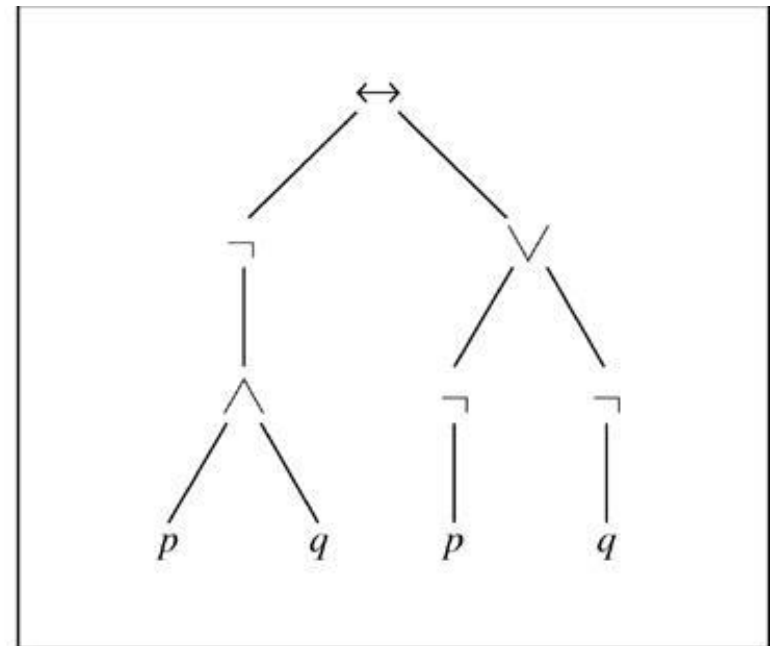
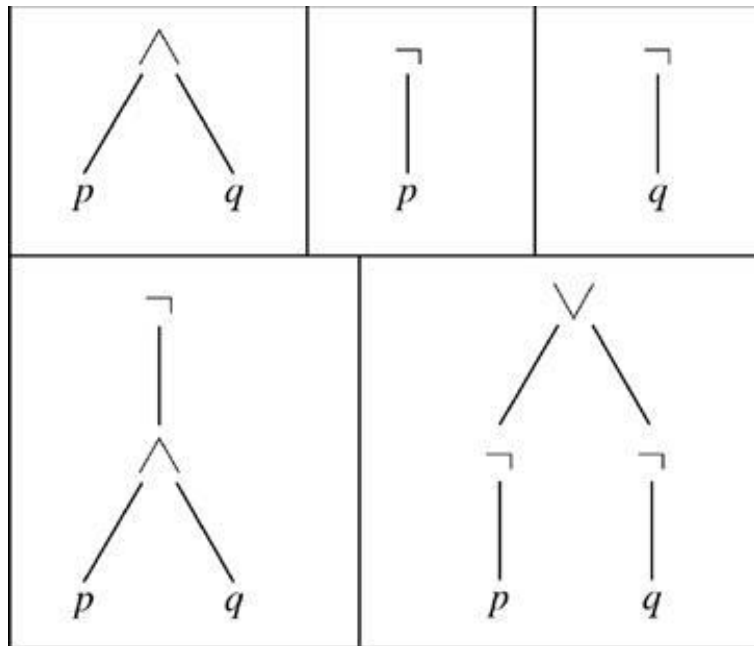
$$\begin{array}{ccccccc} 1 & 9 & 3 & / & + & & \\ & \underbrace{\hspace{1.5cm}} & & & & & \\ & 9/3=3 & & & & & \end{array}$$

$$\begin{array}{r} 1 \quad 3 \quad + \\ \hline 1 + 3 = 4 \end{array}$$

Value of expression: 4

Example 10 Find the ordered rooted tree representing the compound proposition $(\neg(p \wedge q)) \leftrightarrow (\neg p \vee \neg q)$. Then use this rooted tree to find the prefix, postfix, and infix forms of this expression.

Sol.



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

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

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

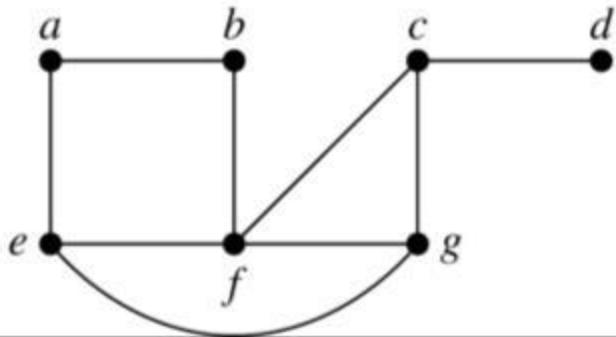
Exercise : 17, 23, 24

10.4 Spanning Trees

Introduction

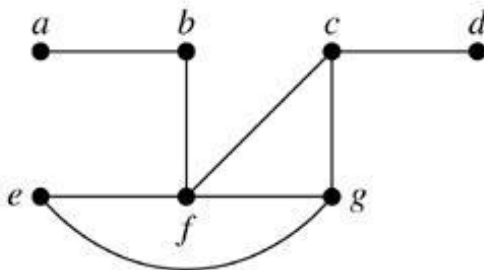
Def. Let G be a simple graph. A **spanning tree** of G is a subgraph of G that is a tree containing every vertex of G .

Example 1 Find a spanning tree of G .



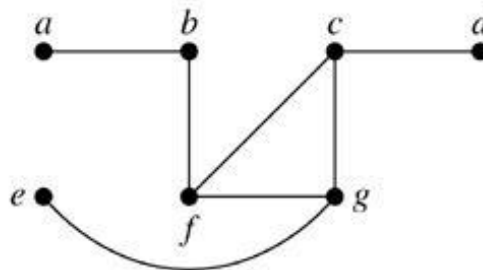
Sol.

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



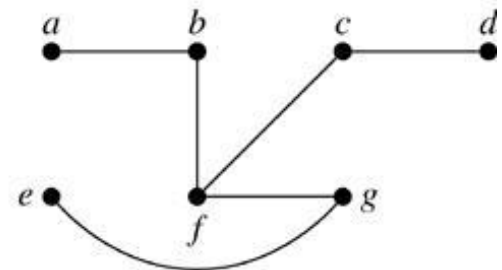
Edge removed: $\{a, e\}$

(a)



$\{e, f\}$

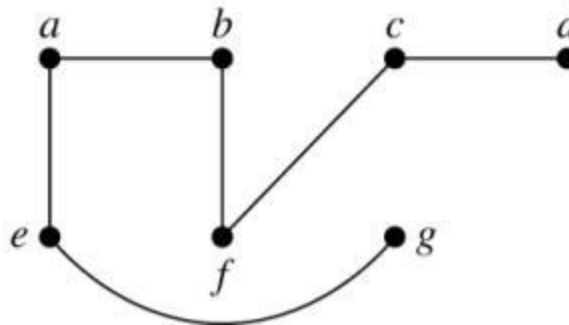
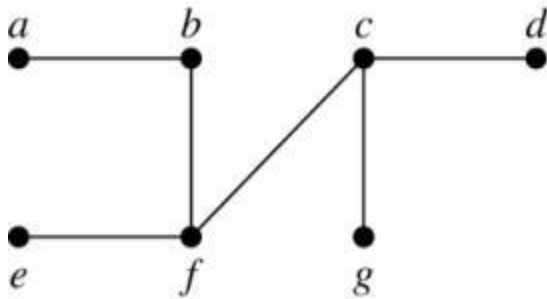
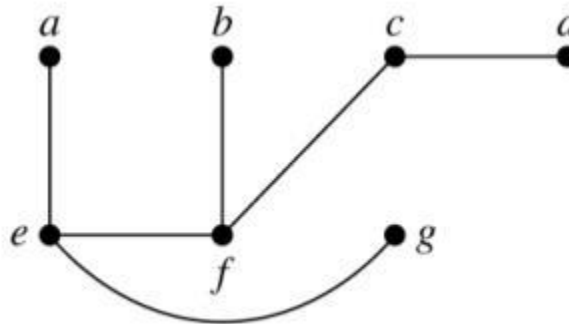
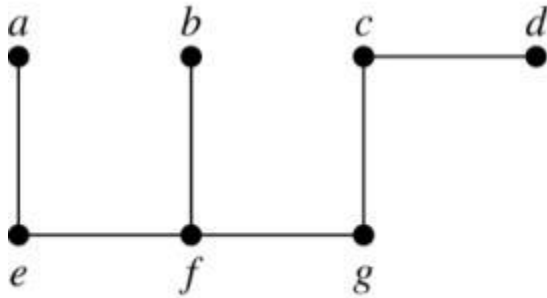
(b)



$\{c, g\}$

(c)

Four spanning trees of G :



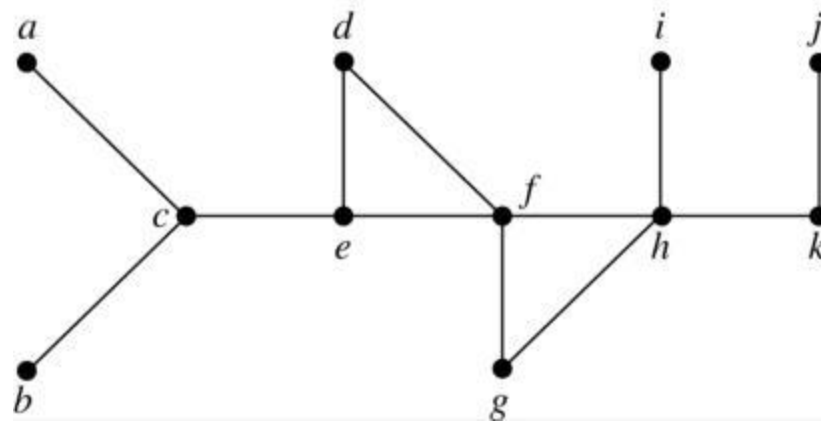
Exercise : 1, 8, 11

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

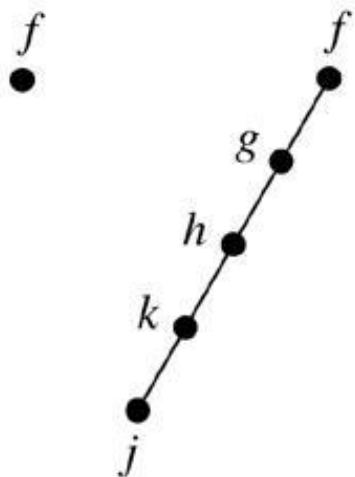
Exercise : 24, 25

Depth-First Search (DFS)

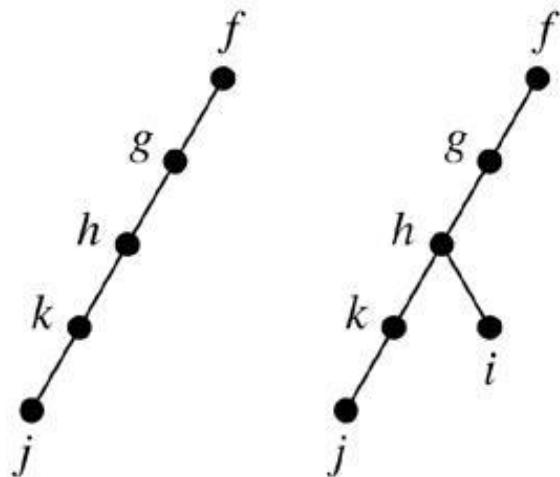
Example 3 Use depth-first search to find a spanning tree for the graph.



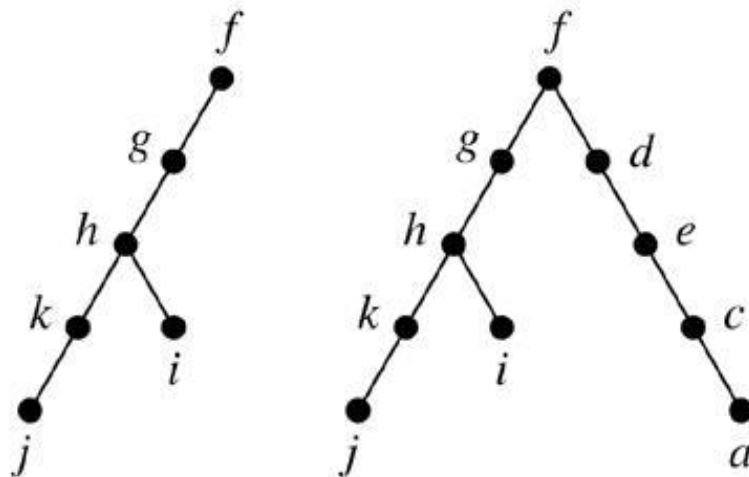
Sol. (arbitrarily start with the vertex f)



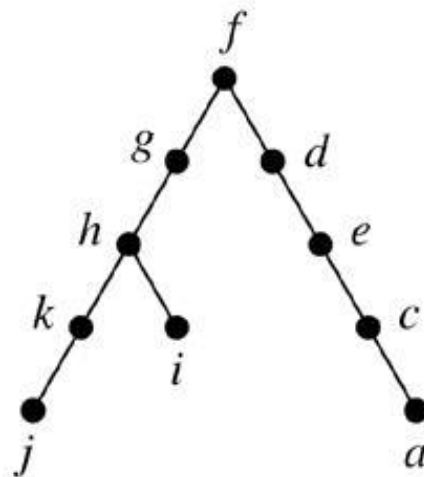
(a)



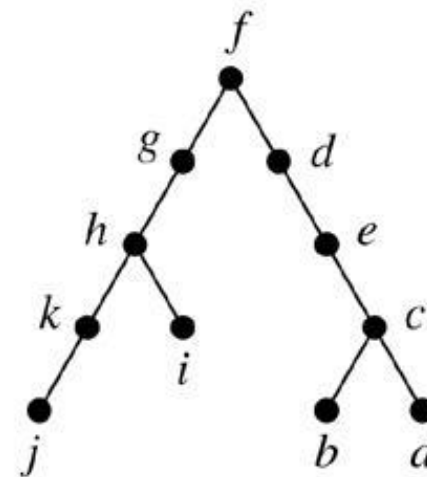
(b)



(c)



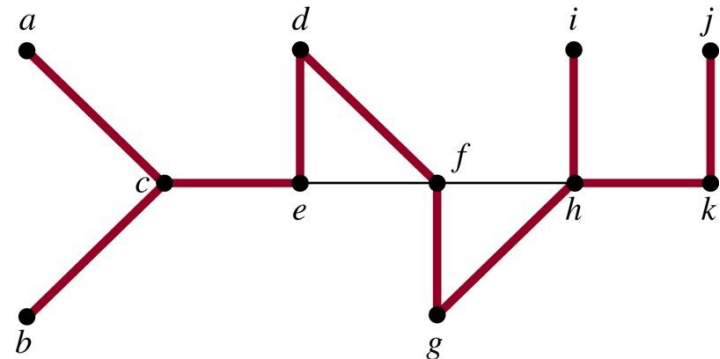
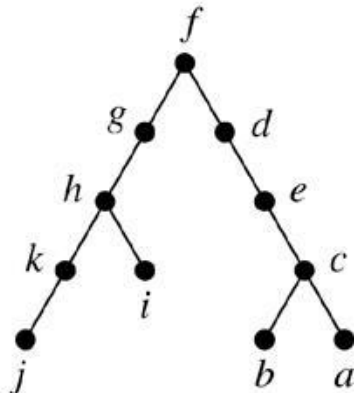
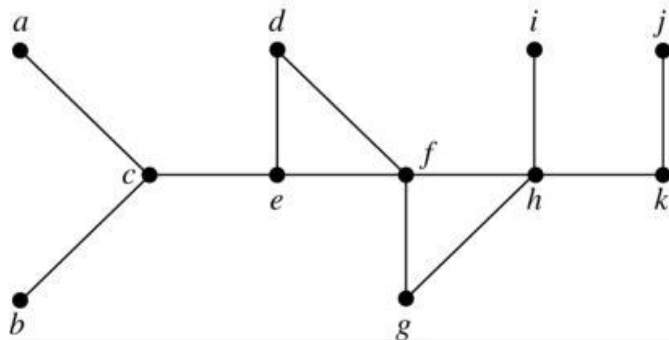
(d)



(e)

The edges selected by DFS of a graph are called **tree edges**. All other edges of the graph must connect a vertex to an ancestor or descendant of this vertex in the tree. These edges are called **back edges**.

Example 4



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

Algorithm 1 (Depth-First Search)

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

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

visit(v_1)

procedure *visit*(v : vertex of G)

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

begin

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

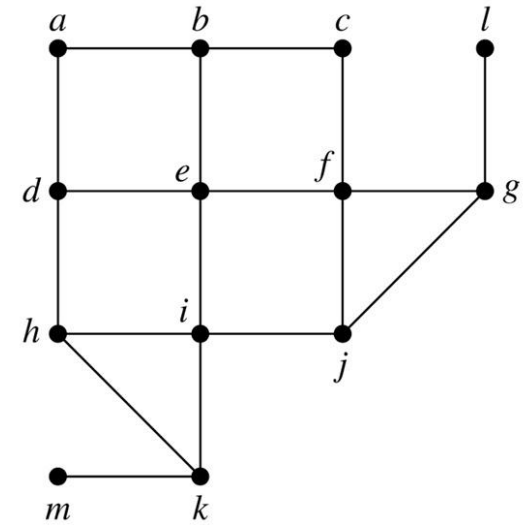
visit(w)

end

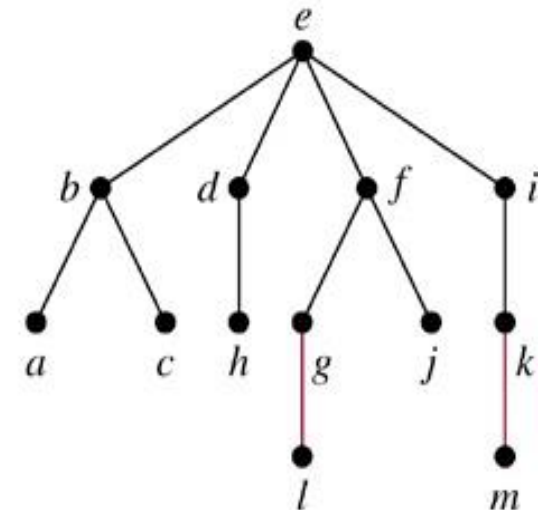
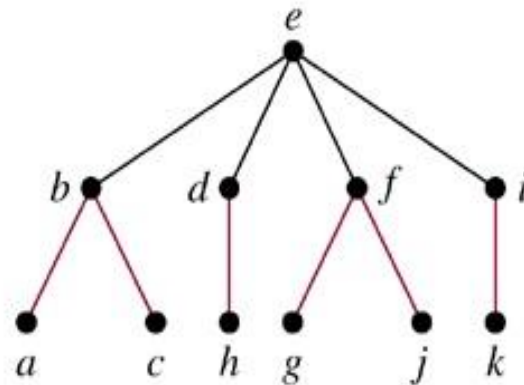
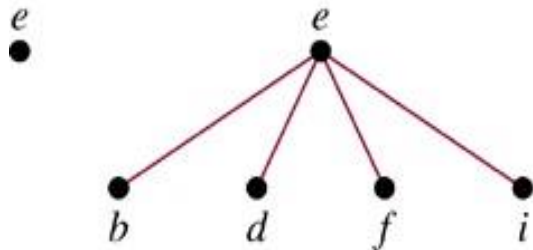
Exercise : 13

Breadth-First Search (BFS)

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



Sol. (arbitrarily start with the vertex e)



Algorithm 2 (Breadth-First Search)

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

Exercise : 16



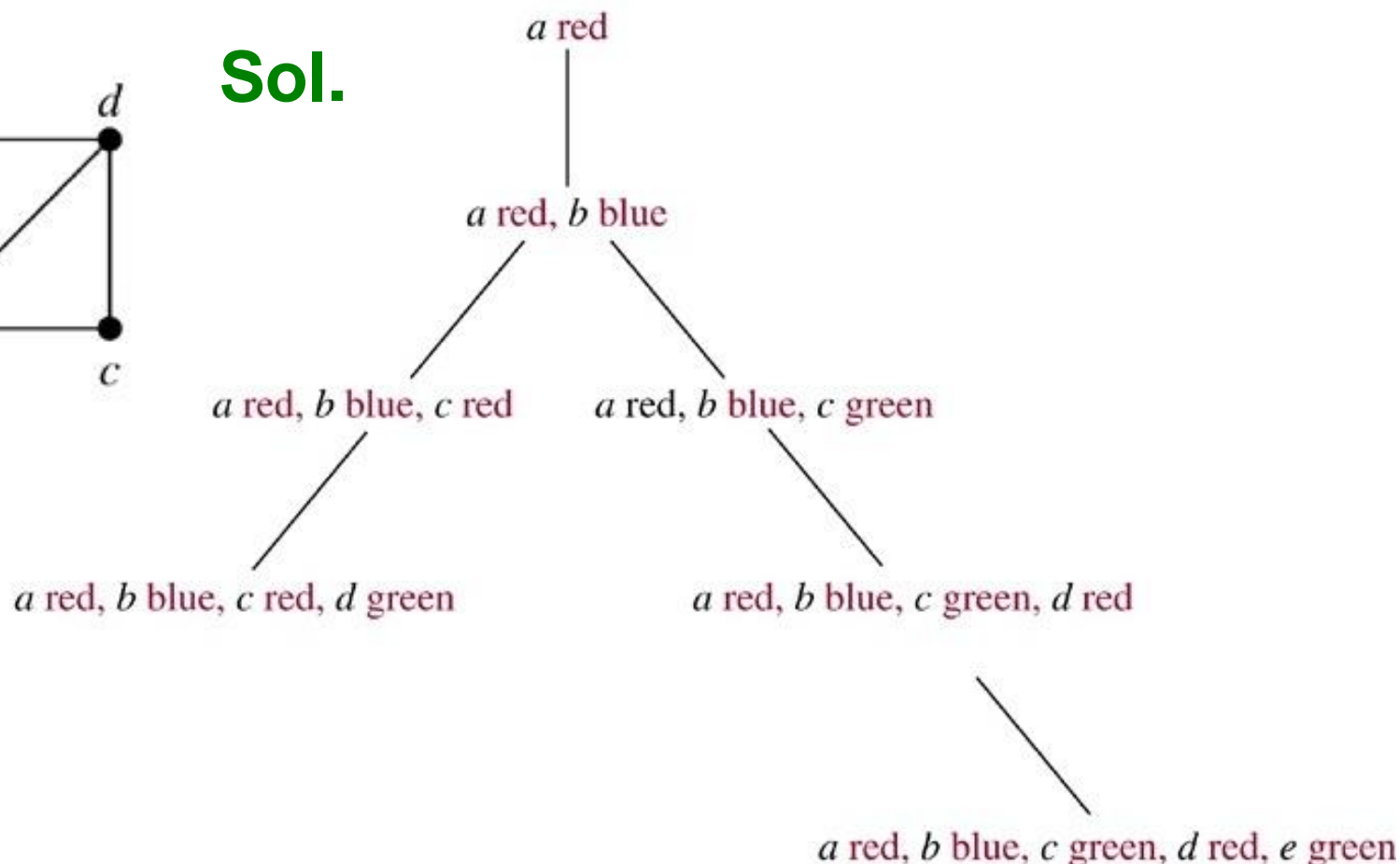
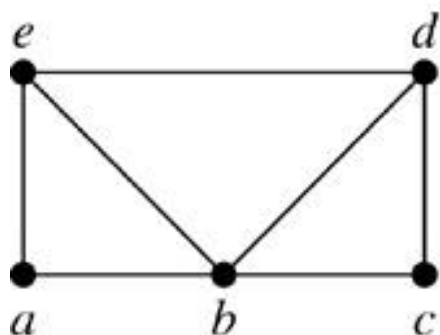
Backtracking Applications

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

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

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

Sol.

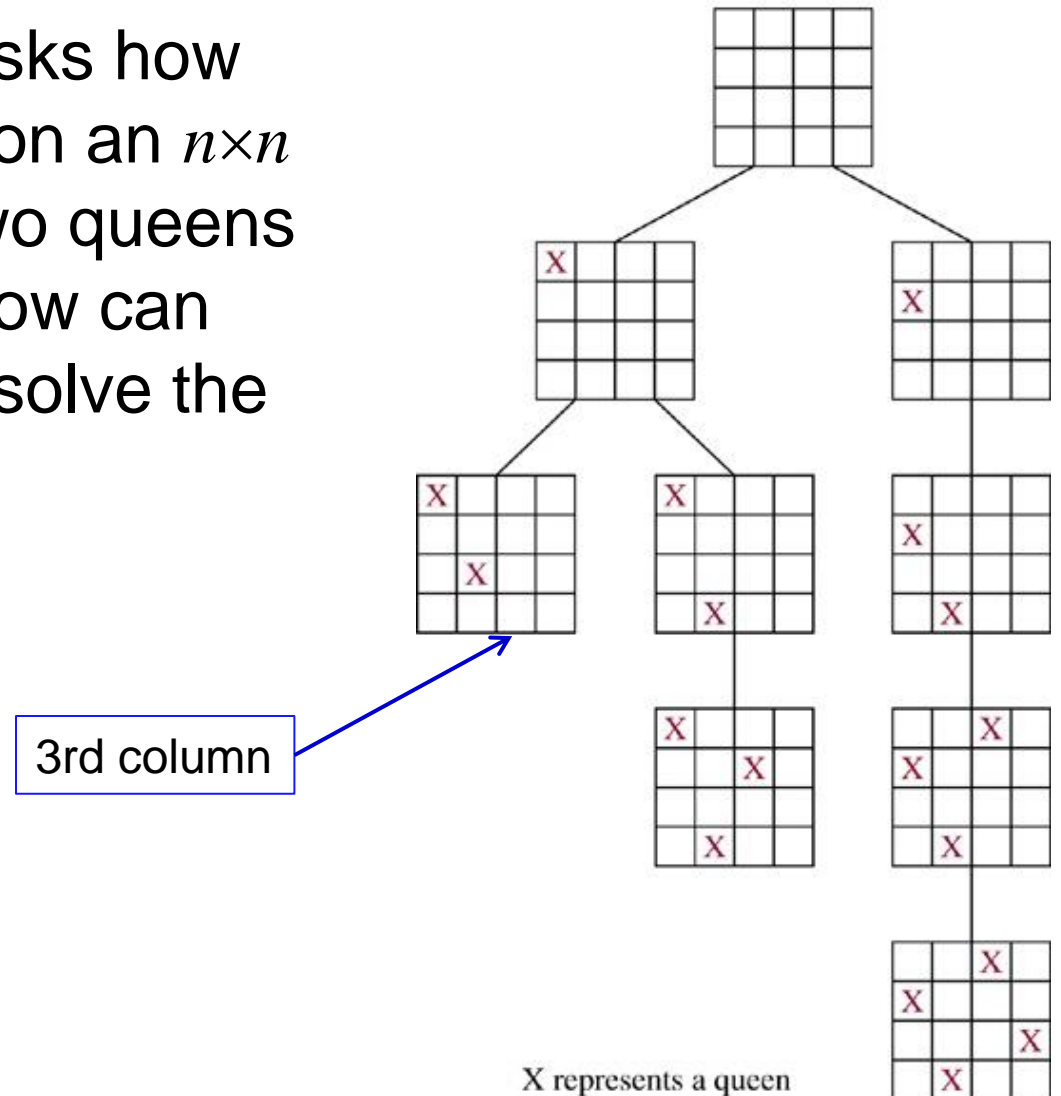


Example 7

(The n -Queens Problem)

The n -queens problem asks how n queens can be placed on an $n \times n$ chessboard so that no two queens can attack on another. How can backtracking be used to solve the n -queens problem.

Sol.



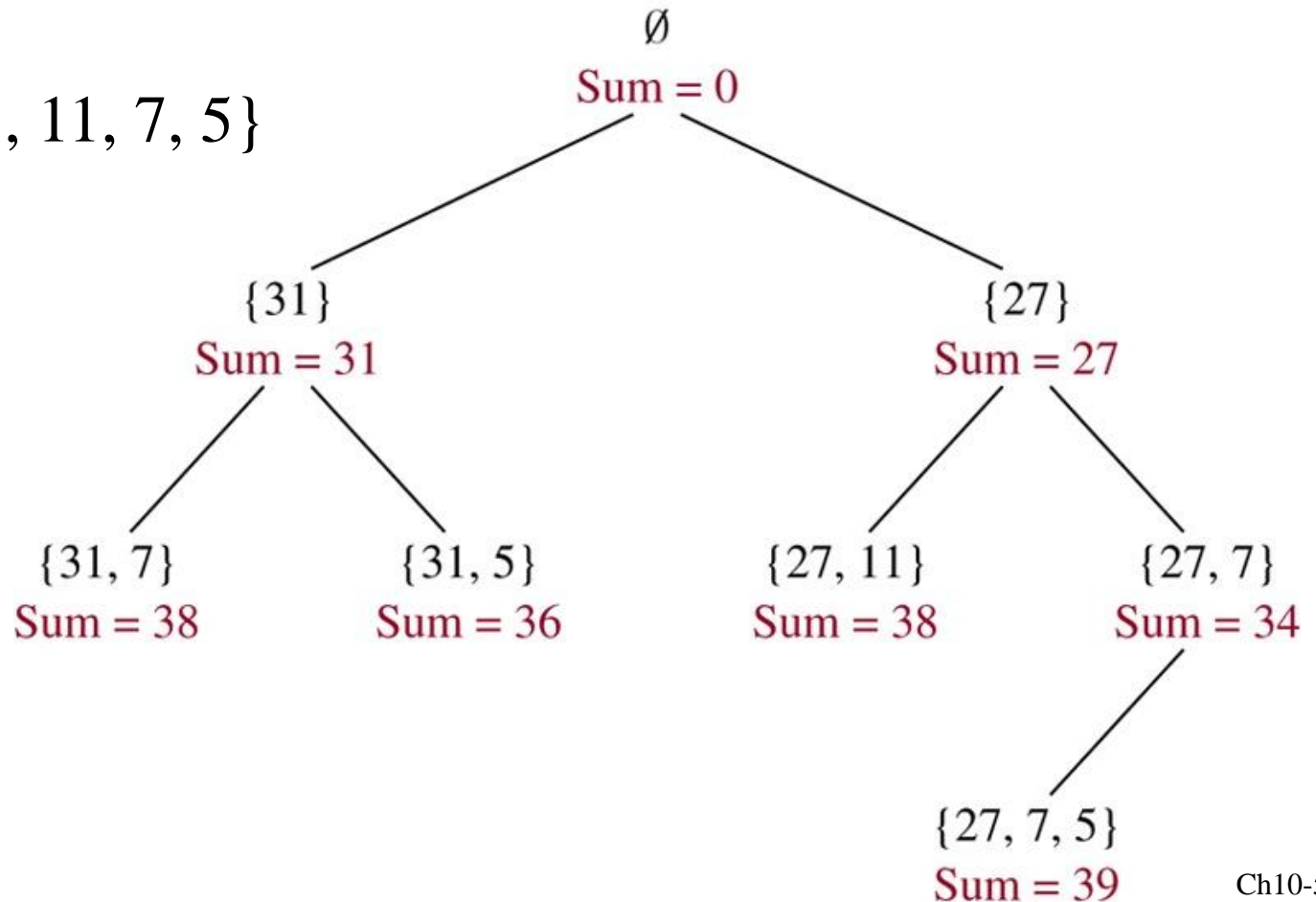
Example 8 (Sum of Subsets)

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

Sol.

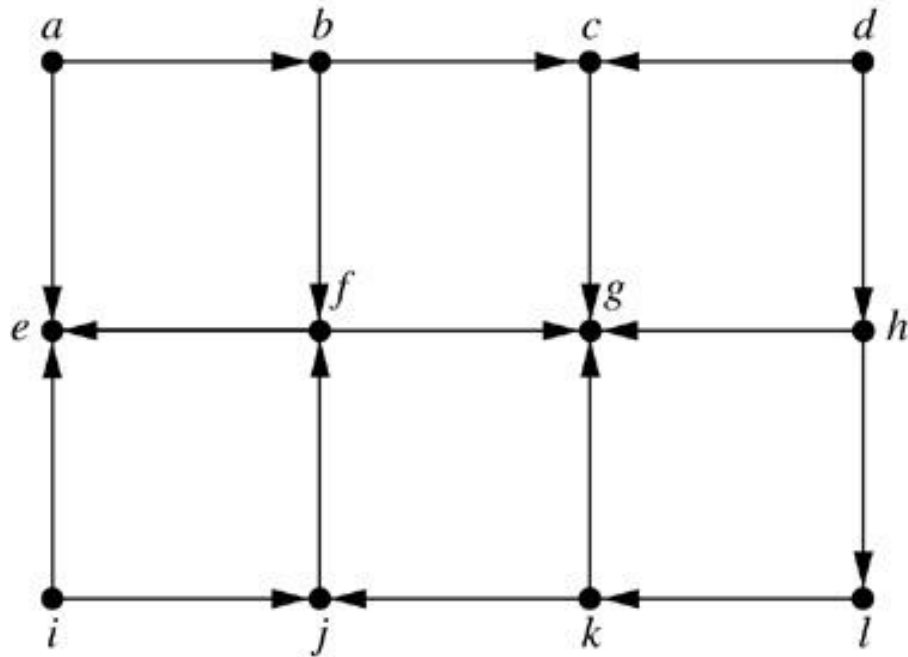
$$S = \{31, 27, 15, 11, 7, 5\}$$

$$M = 39$$



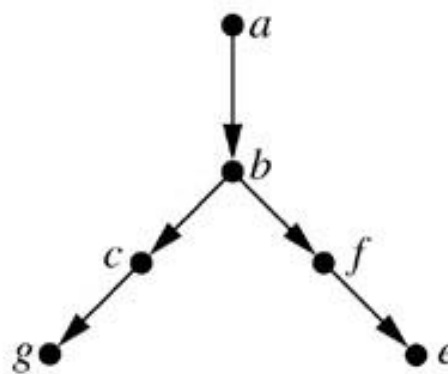
Depth-First Search in Directed Graphs

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



(a)

Sol.



(b)



• i

10.5 Minimum Spanning Trees

G : connected weighted graph (each edge has an weight ≥ 0)

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

Algorithms for Minimum Spanning Trees

Algorithm 1 (Prim's Algorithm)

Procedure *Prim*(G : connected weighted undirected graph with n vertices)

$T :=$ a minimum-weight edge

for $i := 1$ **to** $n-2$

begin

$e :=$ an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T

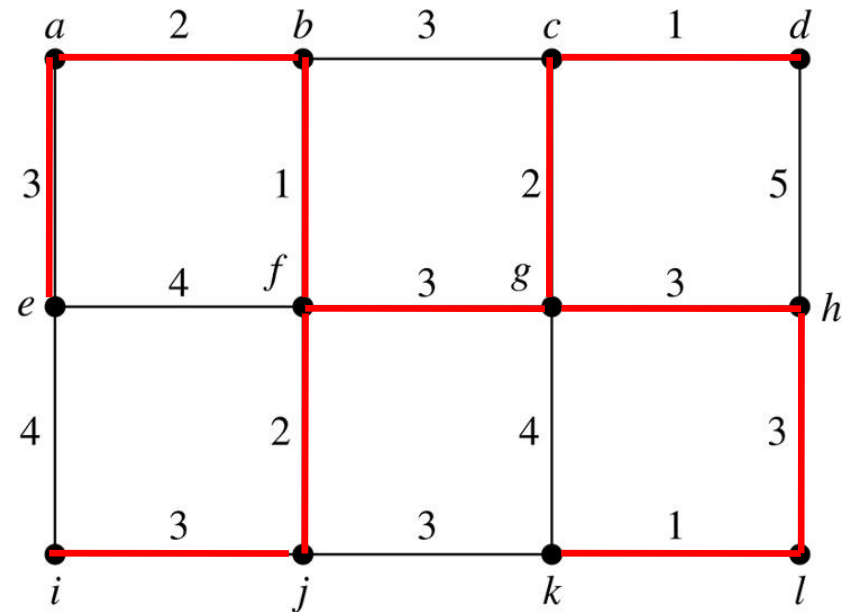
$T := T$ with e added

end { T is a minimum spanning tree of G }

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

Sol.

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



Exercise: 3

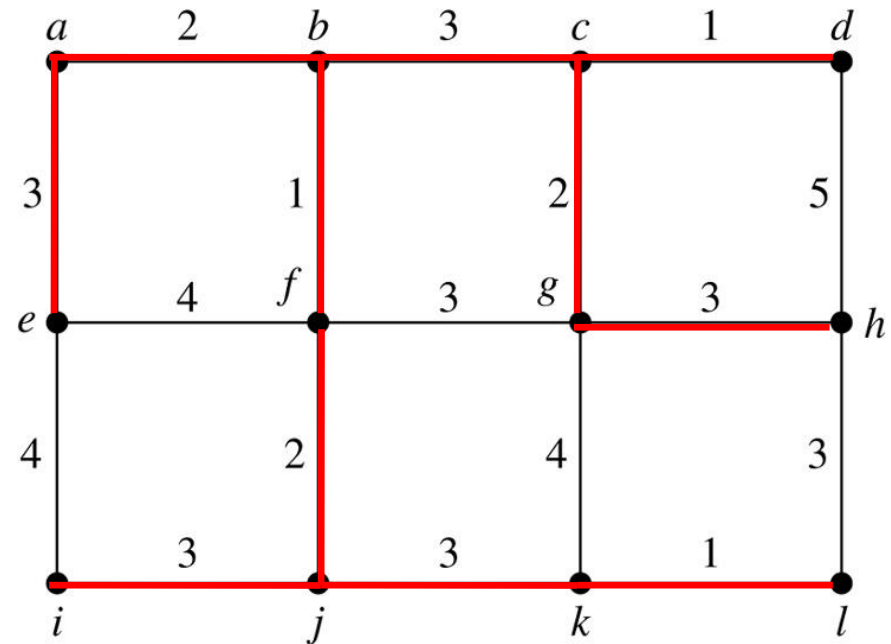
Algorithm 2 (Kruskal Algorithm)

```
Procedure Kruskal( $G$ : connected weighted undirected graph with  $n$  vertices)  
 $T :=$  empty graph  
for  $i := 1$  to  $n-1$   
begin  
     $e :=$  any edge in  $G$  with smallest weight that does not form a simple  
        circuit when added to  $T$   
     $T := T$  with  $e$  added  
end { $T$  is a minimum spanning tree of  $G$ }
```

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

Sol.

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



Exercise: 7