<u>Graph</u>: A graph G = (V, E) consists of a nonempty set of vertices (or nodes) V and a set of edges E. Each edge has either one or two vertices associated with it, called its endpoints. An edge is said to connect its endpoints.

<u>UNDIRECTED GRAPH</u>: a set of vertices and a set of undirected edges each of which is associated with a set of one or two of these vertices.

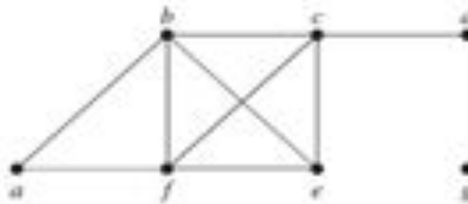


Figure X. an undirected graph. A vertex with zero degree is called isolated e.g. vertex g. A vertex with degree one is called pendent e.g. vertex *a*.

deg (v) (degree of the vertex v in an undirected graph): the number of edges incident with v. A loop contributes twice to the degree of a vertex.

Some results in undirected graphs:

a. **Handshaking theorem**: The sum of the degrees of all vertices in a graph is equal to twice the number of edges. Thus, $2e = \sum \deg(v)\,, for\ all\ v\ \in V.$
   This shows that sum of degrees of vertices in an undirected graph is even. Handshaking theorem applies even if multiple edges and loops are present.
b. An undirected graph has an even number of vertices of odd degree.

<u>DIRECTED GRAPH</u>: is a set of vertices together with a set of directed edges each of which is associated with an ordered pair of vertices.

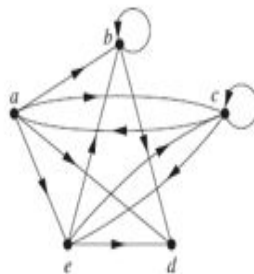


Figure X. A directed graph

When (u, v) is an edge of the graph G with directed edges, u is said to be adjacent to v and v is said to be adjacent from u.

Degree of a vertex in a directed graph:

In-degree of a vertex [$deg^-(v)$] is the number of edges with v as their terminal vertex (i.e. number of edges entering a particular vertex).

Out-degree of a vertex [$deg^+(v)$] is the number of edges with v as their initial vertex (i.e. the number of edges leaving a particular vertex).

The degree of a vertex in directed graph is **sum** of its in-degree and out-degree. A loop at a vertex contributes 1 to both the in-degree and the out-degree of this vertex.

In a directed graph, $\text{sum of indegrees} = \text{sum of outdegrees} = \text{number of edges in the graph.}$

$Thus, \sum_{v \in V} deg^-(v) = \sum_{v \in V} deg^+(v) = e$

SIMPLE GRAPH: an undirected graph with no multiple edges or loops.

Loop: an edge connecting a vertex with itself.

Multi-graph: an undirected graph that may contain multiple edges but no loops.

Pseudo-graph: an undirected graph that may contain multiple edges and loops.

Some special simple graphs

1. K_n (**complete graph** on n vertices): the undirected graph with n vertices where **each pair of vertices** is connected by an edge i.e. there is exactly one edge between each distinct vertex pair.

   $No. of\ edges\ in\ k_n = \frac{n(n-1)}{2} = C(n, 2) = \binom{n}{2}.$
   $Mesh\ network\ of\ n\ nodes\ constitutes\ a\ complete\ graph.$ Therefore, number of edges/connections in this network is equal to $\binom{n}{2}$.

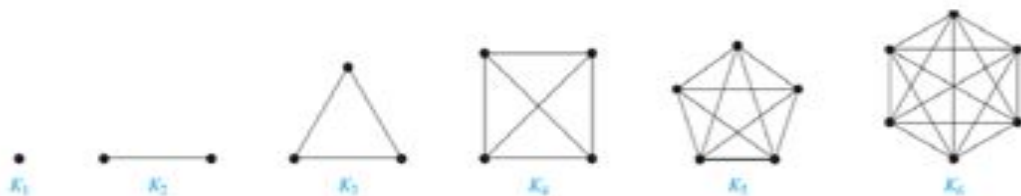
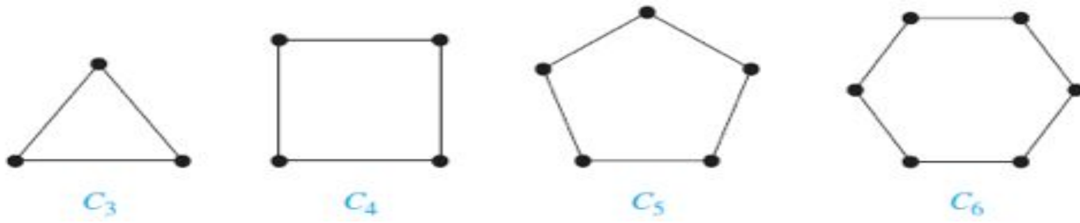


Figure X. Complete graphs on 1, 2, 3, 4, 5, and 6 vertices.

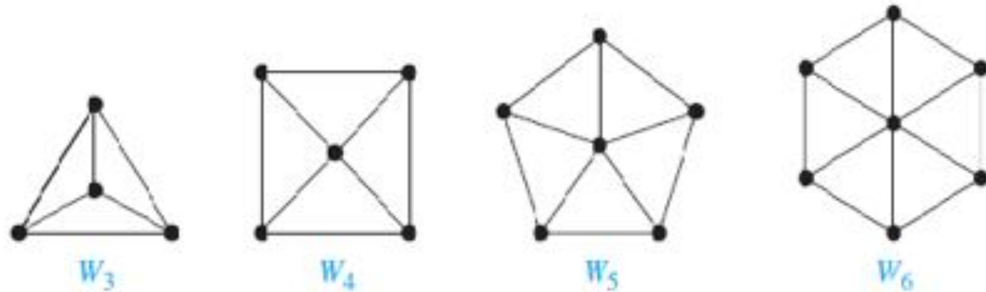
   A **clique** in a simple undirected graph is a **complete** subgraph that is not contained in any larger complete subgraph (i.e. a maximal complete subgraph of a graph).

2. C_n (**cycle** of size n), n ≥ 3: the graph with n vertices $v_1, v_2, \ldots, v_n$ and edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}$.
   $No. of\ edges\ in\ c_n = n.$



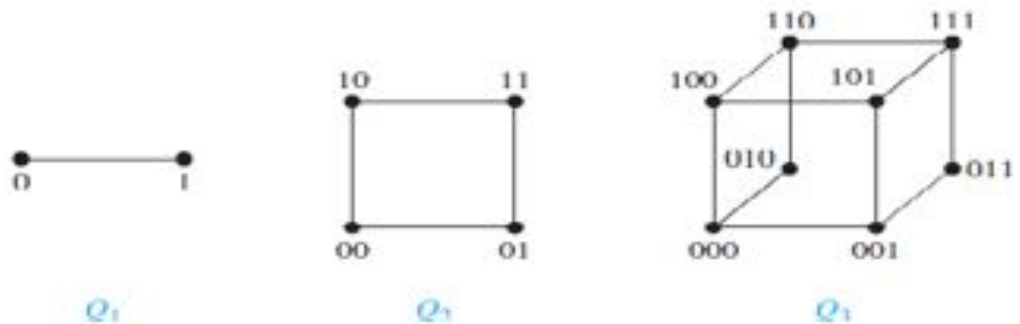
3. W_n (**wheel** of size n), n ≥ 3: the graph obtained from C_n by adding a new vertex and edges from this vertex to each of the n vertices in C_n .

$$No.of\ vertices\ in\ W_n = n + 1\ and\ No.of\ edges = 2n.$$



4. Q_n (n-cube), n ≥ 1: the graph that has the 2^n bit strings of length n as its vertices and edges connecting every pair of bit strings that differ by exactly one bit.

$$No.of\ vertices\ in\ Q_n = 2^n\ and\ No.of\ edges = n2^{n-1}$$



---

5. **BIPARTITE GRAPH** (also known as bigraph): a graph with vertex set that can be partitioned into disjoint subsets V1 and V2 so that each edge connects a vertex in V1 and a vertex in V2 (no edge in G connects either two vertices in V1 or two vertices in V2). The pair (V1, V2) is called a bipartition of V.

A complete graph K_n, n≥3 cannot be bipartite.

Test for bipartition

i. A simple graph is bipartite if and only if it is **2-colorable** i.e. it is possible to assign one of two different colors to each vertex of the graph so that no two adjacent vertices are assigned the same color.

ii. A graph is bipartite if and only if **does not contain any odd length cycles**. That is, it is not possible to start at a vertex and return to this vertex by traversing an odd number of distinct edges.

Examples:

i. C_6 is Bipartite. K_3 is not a bipartite graph.
ii. Graph G (shown below) is bipartite because its vertex set is the union of two disjoint sets, {a, b, d} and {c, e, f, g}, and each edge connects a vertex in one of these subsets to a vertex in the other subset. (Note that for G to be bipartite it is not necessary that every vertex in {a, b, d} be adjacent to every vertex in {c, e, f, g}. For instance, b and g are not adjacent.) Alternatively, you can check it is 2-colorable.
iii. Graph H (shown below) is not bipartite because its vertex set cannot be partitioned into two subsets so that edges do not connect two vertices from the same subset. Alternatively, you can check it is not 2-colorable.

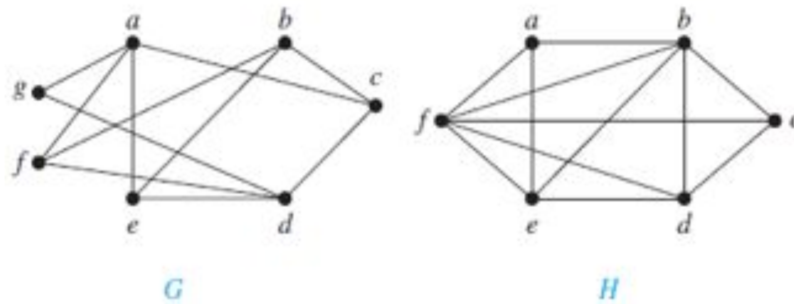
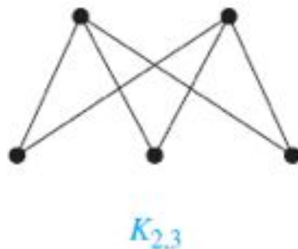
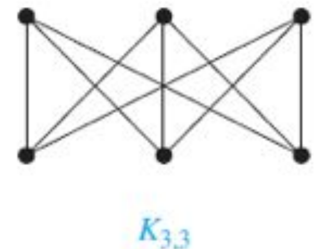


Figure X. G is Bipartite and H is not.

6. **COMPLETE BIPARTITE GRAPH** ($K_{m,n}$): the graph with vertex set partitioned into a subset of m vertices and a subset of n vertices. There is an edge between two vertices if and only if one is in the first subset and the other is in the second subset.

$The\ no.of\ vertices\ in\ K_{m,n} = m + n$ and $no.of\ edges = m \times n$.





REGULAR GRAPH

A simple graph is called regular **if every vertex of this graph has the same degree**. A regular graph is called n-regular if every vertex in this graph has degree n.

It can be seen that:

i. K_n is (n-1)-regular for n≥1 because degree of each vertex in k_n is (n-1).
ii. $K_{m,n}$ is n-regular or m-regular for m=n≥1. $K_{m,n}$ is not regular if m≠n.
iii. C_n is 2-regular for n≥3.
iv. W_n is 3-regular for n=3.
v. Q_n is n-regular for n≥0.

Example: How many vertices does a regular graph of degree four with 10 edges have?

Solution: let the number of vertices and edges be n and e respectively.

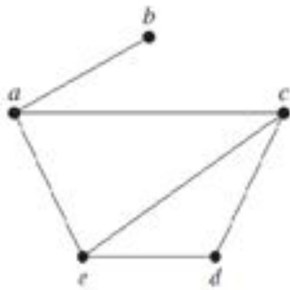
We know, $2e = \sum \deg(v)$

$=> 2 \times 10 = 4.n$

$=> n = \frac{20}{4} = \mathbf{5}\ \boldsymbol{vertices}$

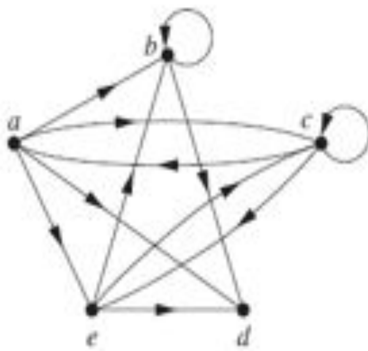
REPRESENTING GRAPHS

1. **Adjacency list**: specifies the vertices that are adjacent to each vertex of a graph. Following example illustrates the adjacency list representation.
   Adjacency list for an undirected graph:



Vertex	*Adjacent Vertices*
a	*b, c, e*
b	*a*
c	*a, d, e*
d	*c, e*
e	*a, c, d*

   Adjacency list for a directed graph:



Initial Vertex	*Terminal Vertices*
a	*b, c, d, e*
b	*b, d*
c	*a, c, e*
d	
e	*b, c, d*

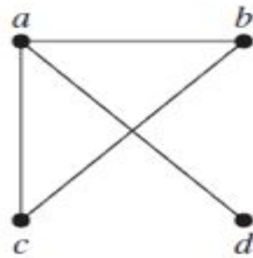
   Adjacency list provides a compact way for representing **sparse** graphs (i.e. graphs containing relatively few edges, $e << n^2$). The representation of a sparse graph using adjacency list results in lot of space savings. Therefore, it is method of choice when graph is sparse.

2. **Adjacency Matrix**:
   Suppose that G = (V, E) is a simple graph where |V| = n. Suppose that the vertices of G are listed arbitrarily as $v_1, v_2, \ldots, v_n$. The adjacency matrix A of G, with respect to this listing of the vertices, is the n x n zero–one matrix with 1 as its (i, j)th entry when v_i and v_j are adjacent, and 0 as its (i, j)th entry when they are not adjacent. In other words,

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

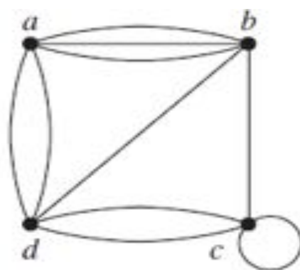
Since an adjacency matrix of a graph is based on the ordering chosen for the vertices. Hence, there may be as many as n! different adjacency matrices for a graph with n vertices, because there are n! different orderings of n vertices.

Adjacency matrix for an undirected simple graph is **symmetric**, that is, $a_{ij} = a_{ji}$



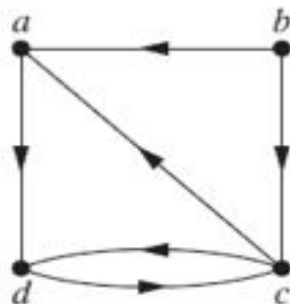
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Adjacency matrix for pseudo-graphs (multiple edges and loops) is no longer a zero-one matrix, for example.



$$\begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}$$

The adjacency matrix can also be used to represent digraphs (with multiple edges and loops). The adjacency matrix for a directed graph does not have to be symmetric, For example:



$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

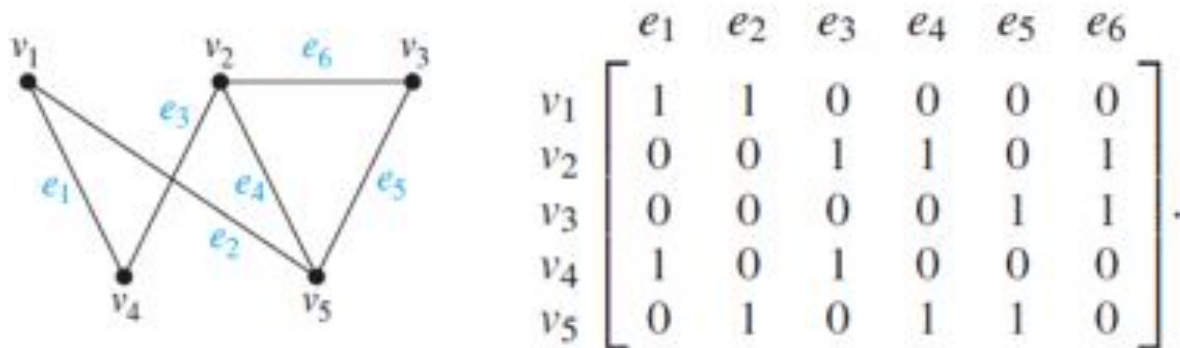
Finally, adjacency Matrix representation is preferred when the graph is **dense (**i.e. a graph containing many edges, **e close to** n^2**).** The representation of a dense graph using adjacency matrix results in efficient graph algorithms.

3. Incidence Matrices: The adjacency list and adjacency matrix are two standard ways to represent graphs. However there is another common way that is incidence matrix.

Let G = (V ,E) be an undirected graph. Suppose that $v_1, v_2, \ldots, v_n$ are the vertices and $e_1, e_2, \ldots, e_m$ are the edges of G. Then the incidence matrix with respect to this ordering of V and E

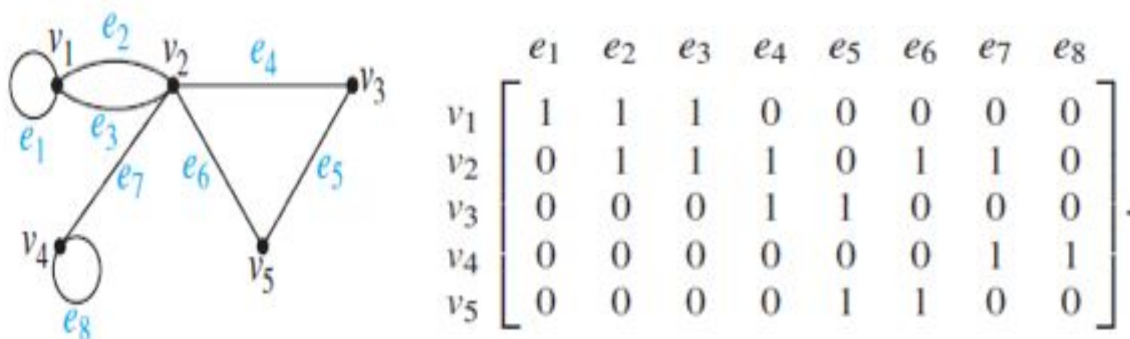
is the n × m matrix M = [m_{ij}], where $m_{ij} = \begin{cases} 1 & \text{when } e_j \text{ is incident with } v_j \\ 0 & \text{otherwise.} \end{cases}$

For example:



	e_1	e_2	e_3	e_4	e_5	e_6
v_1	1	1	0	0	0	0
v_2	0	0	1	1	0	1
v_3	0	0	0	0	1	1
v_4	1	0	1	0	0	0
v_5	0	1	0	1	1	0

Incidence matrices can also be used to represent multiple edges and loops. For example,



	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
v_1	1	1	1	0	0	0	0	0
v_2	0	1	1	1	0	1	1	0
v_3	0	0	0	1	1	0	0	0
v_4	0	0	0	0	0	0	1	1
v_5	0	0	0	0	1	1	0	0

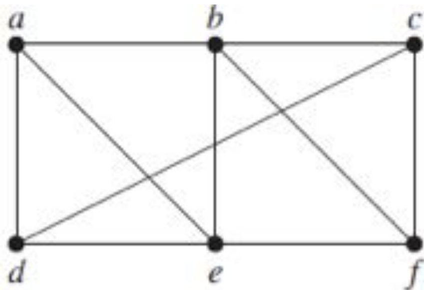
<u>CONNECTIVITY</u>

Path/walk: A path of length n ($n \geq 0$) from a vertex *u* to a vertex *v* in a graph G = (V, E) is a sequence $< x_0, x_1, \ldots, x_{n-1}, x_n >$ of vertices such that $u = x_0$, $v = x_n$ and $(x_{i-1}, x_i) \in E$ for i=1, 2, ... ,n. The path contains the vertices $x_0, x_1, \ldots, x_{n-1}, x_n$ and edges $(x_0,x_1), (x_1,x_2),\ldots, (x_{n-1},x_n)$.

Thus, **path** is a sequence of edges that begins at a vertex of a graph and travels from vertex to vertex along edges of the graph. When the graph is simple we can represent this path using a vertex sequence (because listing these vertices uniquely determines the path). The length of the path is the number of edges in the path. There is always a zero length path from a vertex to itself. If there is a path from u to v, we say that v is **reachable** from u via this path. Further, a path may repeat an edge or vertex more than once.

Circuit/cycle/closed walk: is a path that begins and ends at the same vertex, that is, if u = v, **and** has length greater than zero (i.e. contains at least one edge). A **self-loop** is a cycle of length **one**. A graph with no cycles is called **acyclic** graph.

Simple path/circuit: A path or circuit is simple if it does not contain the same edge more than once.



In the simple graph shown above, < a, d, c, f , e> is a simple path of length 4, because {a, d}, {d, c}, {c, f }, and {f, e} are all edges. However, <d, e, c, a> is not a path, because {e, c} is not an edge. Note that <b, c, f , e, b> is a circuit of length 4 because {b, c}, {c, f }, {f, e}, and {e, b} are edges, and this path begins and ends at b.The path <a, b, e, d, a, b>, which is of length 5, is not simple because it contains the edge {a, b} twice.

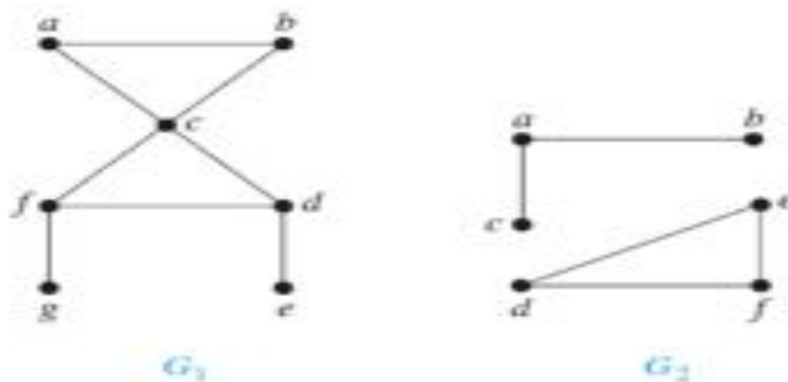
CONNECTIVITY IN UNDIRECTED GRAPHS

An undirected graph is said to be connected **if there is a path between every pair of distinct vertices** of the graph i.e. if every vertex is reachable from all other vertices. An undirected graph that is not connected is called **disconnected.**

When we remove vertices and/or edges, a disconnected graph may result.

Thus, any two computers in the network can communicate if and only if the graph of this network is connected.

Examples: In the figures given below, G_1 is connected and G_2 is disconnected.



Connected component:

A connected component of a graph G is a maximal connected sub-graph of G. That is, a connected component of a graph G is a connected sub-graph of G **that is not a proper sub-graph of another connected sub-graph of G.**

The connected components of a graph are the equivalence classes of vertices under the “is reachable from” relation. The graph given below has three connected components {1, 2, 5}, {3, 6} and {4}. Every vertex in {1, 2, 5} is reachable from every other vertex in {1, 2, 5}.

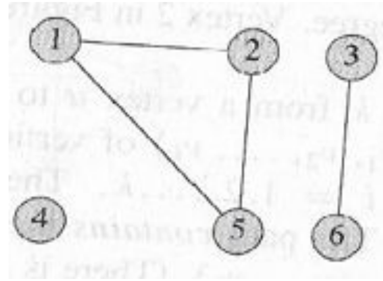
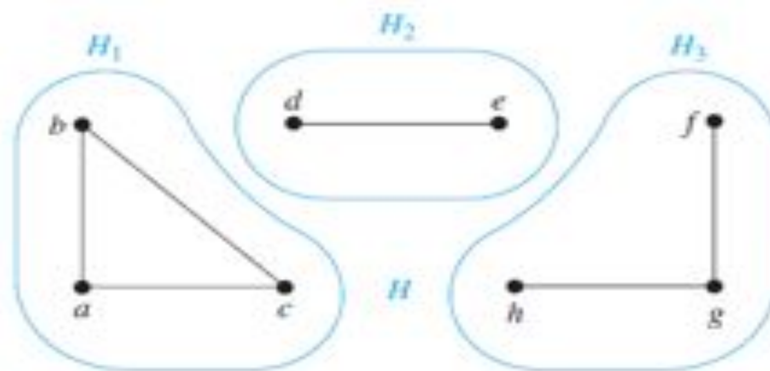


Figure X. A disconnected graph with three connected components {1, 2, 5}, {3, 6} and {4}.

An undirected graph is connected if it has exactly one connected component. A graph G that is not connected has two or more connected components that are disjoint and have G as their union.

The figure given below shows a graph H and its three connected components H1, H2 and H3. The graph H is the union of three disjoint connected sub-graphs H1, H2, and H3.



Cut vertices (or articulation points) and cut-edges (or bridges):

Articulation points and bridges are the measure of a connectedness of a graph i.e. how connected a graph is.

Let G=(V, E) be a connected, undirected graph. A **Cut-vertex (or articulation point)** of G is a vertex whose removal (and all edges incident with this vertex) disconnects G. A bridge or cut-edge of G is an edge whose removal disconnects G. (fig. X)

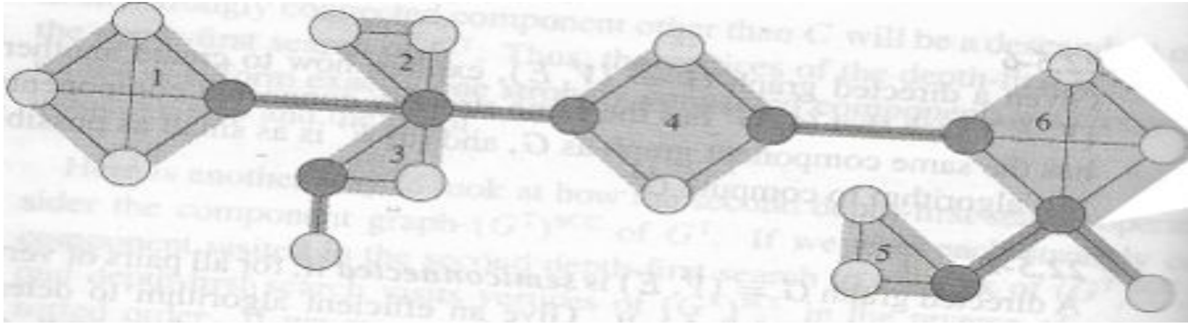


Figure X. The articulation points are the heavily shaded vertices, and the bridges are heavily shaded edges.

A complete graph k_n has no cut-vertex. Removal of any vertex from K_n produces a complete graph K_{n-1}. Connected graphs without cut vertices are called **non-separable** graphs. Thus, non-separable graphs are more connected than those with a cut vertex.

<u>Vertex connectivity</u>

Vertex cut (or separating set): If a graph G has a cut vertex, then we need only remove it to disconnect G. If G does not have a cut vertex, then we look for the smallest set of vertices that can be removed to disconnect G. A subset V' of the vertex set V of G = (V, E) is a **vertex cut**, or separating set, if G – V' is disconnected. Every connected graph, **except a complete graph**, has a vertex cut.

Vertex connectivity of a non complete graph G, denoted by κ(G), is the minimum number of vertices in a vertex cut.

However, when G is a complete graph, it has no vertex cuts, because removing any subset of its vertices and all incident edges still leaves a complete graph. Consequently, we cannot define κ(G) as the minimum number of vertices in a vertex cut when G is complete. Instead, we set $\kappa(K_n) = n - 1$, the number of vertices needed to be removed to produce a graph with a single vertex.

Consequently, for every graph G, κ(G) is minimum number of vertices that can be removed from G to either disconnect G or produce a graph with a single vertex.

We have,

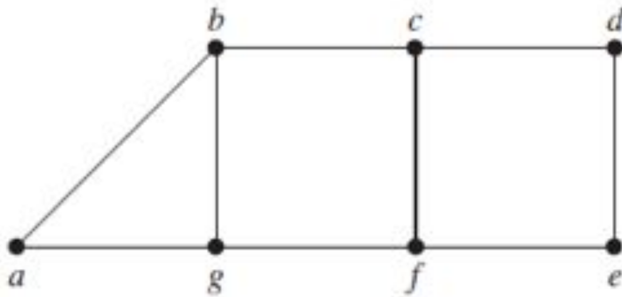
$0 \leq \kappa(G) \leq n - 1$ if G has n vertices,

$\kappa(G) = 0$ if and only if G is disconnected or $G = K_1$,

and $\kappa(G) = n - 1$ if and only if G is complete.

The larger κ(G) is, the more connected G is. Disconnected graphs and K_1 have κ(G) = 0, connected graphs with cut vertices and K_2 have κ(G) = 1, graphs without cut vertices that can be disconnected by removing two vertices and K_3 have κ(G) = 2, and so on. We say that a graph is k-connected (or k-vertex connected), if κ(G) ≥ k. A graph G is 1-connected if it is connected and not a graph containing a single vertex; a graph is 2-connected, or bi-connected, if it is non-separable and has at least three vertices.

Consider the graph G shown as: the graph G has no cut vertices but that {b, g} is a vertex cut. Hence, **κ(G) = 2.**



Edge connectivity

Edge cut: If a graph has a cut edge, then we need only remove it to disconnect G. If G does not have a cut edge, we look for the smallest set of edges that can be removed to disconnect it. A set of edges E' is called an edge cut of G if the sub-graph G – E' is disconnected.

The edge connectivity of a graph G, denoted by λ(G), is the minimum number of edges in an edge cut of G. This defines λ(G) for all connected graphs with more than one vertex because it is always possible to disconnect such a graph by removing all edges incident to one of its vertices.

We have,

$\lambda(G) = 0$ if G is not connected. We also specify that $\lambda(G) = 0$ if G is a graph consisting of a single vertex.

If G is a graph with n vertices, then $0 \le \lambda(G) \le n - 1$.

Also, $\lambda(G) = n - 1$ where G is a graph with n vertices if and only if G = K_n, which is equivalent to the statement that $\lambda(G) \le n - 2$ when G is not a complete graph.

Note that graph G (above) has no cut edges, but the removal of the two edges {b, c} and {f, g} disconnects it.

The following relation holds between vertex and edge connectivity:.

$$\kappa(G) \le \lambda(G) \le \min_{v \in V} \deg(v).$$

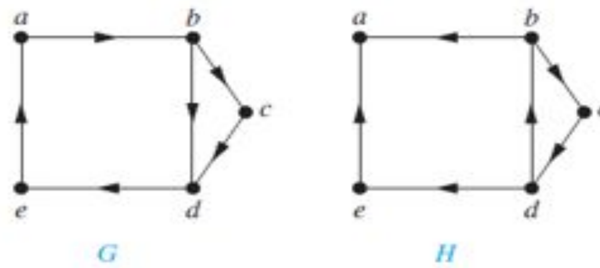
CONNECTIVITY IN DIRECTED GRAPHS

A directed graph can be either strongly connected or weakly connected.

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 i.e. if every two vertices are reachable from each other.

A directed graph is **weakly connected** if there is a path between every two vertices in the underlying undirected graph. That is, a directed graph is weakly connected if and only if there is always a path between two vertices when the directions of the edges are disregarded.

Example: Consider the graphs g and H shown as:

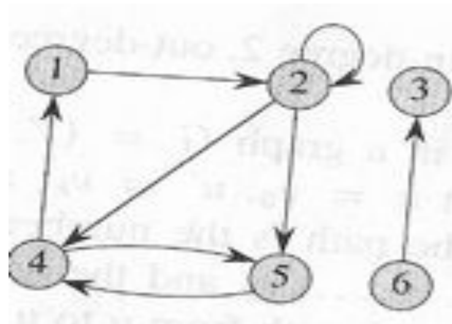


Note that G is strongly connected because there is a path between any two vertices in this directed graph. Hence, G is also weakly connected.

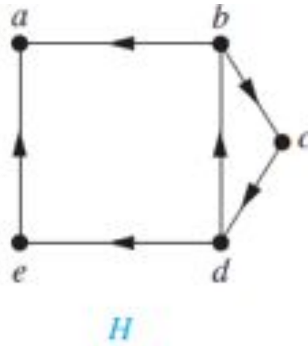
The graph H is not strongly connected. There is no directed path from a to b in this graph. However, H is weakly connected, because there is a path between any two vertices in the underlying undirected graph of H.

Strong connected components of a directed graph: The sub-graphs of a directed graph G that are strongly connected but not contained in larger strongly connected sub-graphs, that is, the maximal strongly connected sub-graphs, are called the strongly connected components or strong components of G. Note that if a and b are two vertices in a directed graph, their strong components are either the same or disjoint.

The strongly connected components of a digraph are the equivalence classes of vertices under the "are mutually reachable" relation. A directed graph is strongly connected if it has only one strongly connected component. The graph given below has three strongly connected components {1, 2, 4, 5}, {3} and {6}. All pairs of vertices in {1, 2, 4, 5} are mutually reachable. The vertices {3, 6} do not form a strongly connected component since vertex 6 cannot be reached from vertex 3.



The graph H in Figure below has three strongly connected components, consisting of the vertex a; the vertex e; and the sub-graph consisting of the vertices b, c, and d and edges (b, c), (c, d), and (d, b).



Some results in connected graphs:

i. Every connected graph with n vertices has at least (n-1) edges. That is, the minimum number of edges in a connected graph equals (n-1).
ii. A simple graph with n vertices and k connected components has at most $\frac{(n-k)(n-k+1)}{2}$ $edges.$
iii. A simple graph with n vertices is connected if it has more than $\frac{(n-1)(n-2)}{2}$ $edges.$
iv.

COUNTING PATHS BETWEEN VERTICES

The number of paths between two vertices in a graph can be determined using its adjacency matrix.

Let G be a graph with adjacency matrix A with respect to the ordering $v_1, v_2, \ldots, v_n$ of the vertices of the graph (with directed or 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)^th^* entry of A^r.

Example 1: How many paths of length four are there from a to d in the given simple graph G.

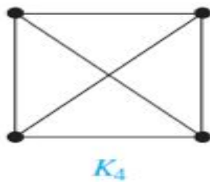
Solution: The adjacency matrix of G (ordering the vertices as a, b, c, d) is

A= $\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$. Therefore A^2 =A.A= $\begin{bmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 2 & 0 \\ 0 & 2 & 2 & 0 \\ 2 & 0 & 0 & 2 \end{bmatrix}$ => $A^4 = A^2 . A^2 =$ $\begin{bmatrix} 8 & 0 & 0 & 8 \\ 0 & 8 & 8 & 0 \\ 0 & 8 & 8 & 0 \\ 8 & 0 & 0 & 8 \end{bmatrix}$

The number of paths of length four from *a to d* is the $(1, 4)^{th}$ entry of A^4. Therefore number of paths equals 8.

Example 2: Find the number of paths of length two between two different vertices in K_4.

Solution: The graph K_4 and adjacency matrix of K_4 is shown as:



K_4

A= $\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$. Here A^4 = $\begin{bmatrix} 3 & 2 & 2 & 2 \\ 2 & 3 & 2 & 2 \\ 2 & 2 & 3 & 2 \\ 2 & 2 & 2 & 3 \end{bmatrix}$

Therefore, the number of paths between any two different vertices is 2.

EULERIAN AND HAMILTONAIN PATHS AND GRAPHS

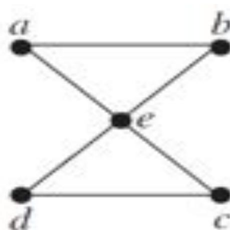
Euler path: An Euler path in a graph G is a path that traverses **each edge** of G exactly once, although it may visit a vertex more than once.

Euler circuit/tour: An Euler circuit in a graph G is a cycle that traverses **each edge** of G exactly once, although it may visit a vertex more than once.

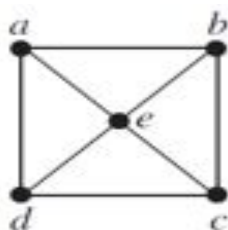
Eulerain Graph: A graph G containing an Euler circuit is called Eulerian. Note that,

i. K_n is Eulerian when n is odd i.e. for $n = 2k + 1$. When n is even, K_n cannot be Eulerian.
ii. C_n is Eulerian $for\ n \geq 3$.
iii. W_n is Eulerian for no value of n.
iv. Q_n is Eulerain for even value of n i.e. $for\ n = 2k$.
v. $K_{m,n}$ is Eulerian if and only if both m and n are even.

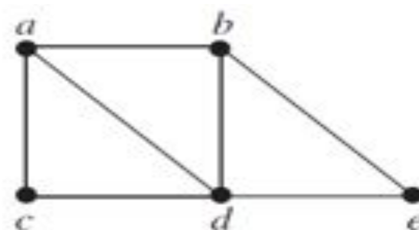
Examples: In the graphs given below, the graph G_1 has an Euler circuit, < a, e, c, d, e, b, a>. Neither of the graphs G_2 or G_3 has an Euler circuit. However, G3 has an Euler path, namely, <a, c, d, e, b, d, a, b>. G2 does not have an Euler path.



G_1



G_2



G_3

Necessary and sufficient conditions for euler circuits and paths in undirected graphs:

1. A connected multi-graph with at least two vertices has an Euler circuit if and only if each of its vertices has even degree.
2. A connected multi-graph has an Euler path but not an Euler circuit if and only if it has exactly two vertices of odd degree.

Necessary and sufficient conditions for euler circuits and paths in directed graphs:

1. A directed multi-graph having no isolated vertices has an Euler circuit if and only if the graph is weakly connected and the in-degree and out-degree of each vertex are equal.
2. A directed multi-graph having no isolated vertices has an Euler path **but not an Euler circuit** if and only if the graph is weakly connected and the in-degree and out-degree of each vertex are equal for all but two vertices, one that has in-degree one larger than its out-degree and the other that has out-degree one larger than its in-degree.

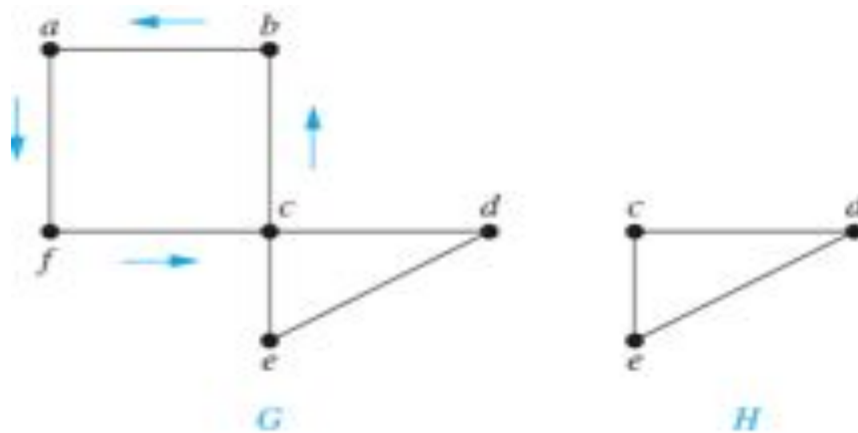
Algorithms for constructing euler circuits:

1.

ALGORITHM Constructing Euler Circuits.

procedure *Euler*(G: connected multigraph with all vertices of even degree)
circuit := a circuit in G beginning at an arbitrarily chosen vertex with edges successively added to form a path that returns to this vertex
H := G with the edges of this circuit removed
while H has edges
    subcircuit := a circuit in H beginning at a vertex in H that also is an endpoint of an edge of *circuit*
    H := H with edges of *subcircuit* and all isolated vertices removed
    circuit := *circuit* with *subcircuit* inserted at the appropriate vertex
return *circuit* {*circuit* is an Euler circuit}

Example: construct an Eulerian circuit in the following graph.



We will form a simple circuit that begins at an arbitrary vertex a of G, building it edge by edge.

Let x_0 = a. First, we arbitrarily choose an edge $\{x_0, x_1\}$ incident with *a* which is possible because G is connected. We continue by building a simple path $\{x_0, x_1\}, \{x_1, x_2\}, \ldots, \{x_{n-1}, x_n\}$, successively adding edges one by one to the path until we cannot add another edge to the path. This happens when we reach a vertex for which we have already included all edges incident with that vertex in the path. For instance, in the graph G given above we begin at *a* and choose in succession the edges {a, f }, {f, c}, {c, b}, and {b, a}.

The path we have constructed must terminate because the graph has a finite number of edges, so we are guaranteed to eventually reach a vertex for which no edges are available to add to the path. The path begins at *a* with an edge of the form {a, x}, and we now show that it must terminate at *a* with an edge of the form {y, a}. To see that the path must terminate at *a*, note that each time the path goes through a vertex with even degree, it uses only one edge to enter this vertex, so because the degree must be at least two, at least one edge remains for the path to leave the vertex. Furthermore, every time we enter and leave a vertex of even degree, there is an even number of edges incident with this vertex that we have not yet used in our path. Consequently, as we form the path, every time we enter a vertex other than a, we can leave it. This means that the path can end only at *a*. Next, note that the path we have constructed may use all the edges of the graph, or it may not if we have returned to *a* for the last time before using all the edges.

An Euler circuit has been constructed if all the edges have been used. Otherwise, consider the sub-graph H obtained from G by deleting the edges already used and vertices that are not incident with any remaining edges. When we delete the circuit <a, f, c, b, a> from the given graph, we obtain the sub-graph labeled as H.

Because G is connected, H has at least one vertex in common with the circuit that has been deleted. Let w be such a vertex. (In our example, c is the vertex.)

Every vertex in H has even degree (because in G all vertices had even degree, and for each vertex, pairs of edges incident with this vertex have been deleted to form H). Note that H may not be connected. Beginning at w, construct a simple path in H by choosing edges as long as possible, as

was done in G. This path must terminate at w. For instance, in given graph,<c, d, e, c> is a path in H. Next, form a circuit in G by splicing the circuit in H with the original circuit in G (this can be done because w is one of the vertices in this circuit). When this is done, we obtain the circuit <a, f, c, d, e, c, b, a>.

Continue this process until all edges have been used. (The process must terminate because there are only a finite number of edges in the graph.) This produces an Euler circuit. The construction shows that if the vertices of a connected multi-graph all have even degree, then the graph has an Euler circuit.

2. **Fleury's Algorithm**:

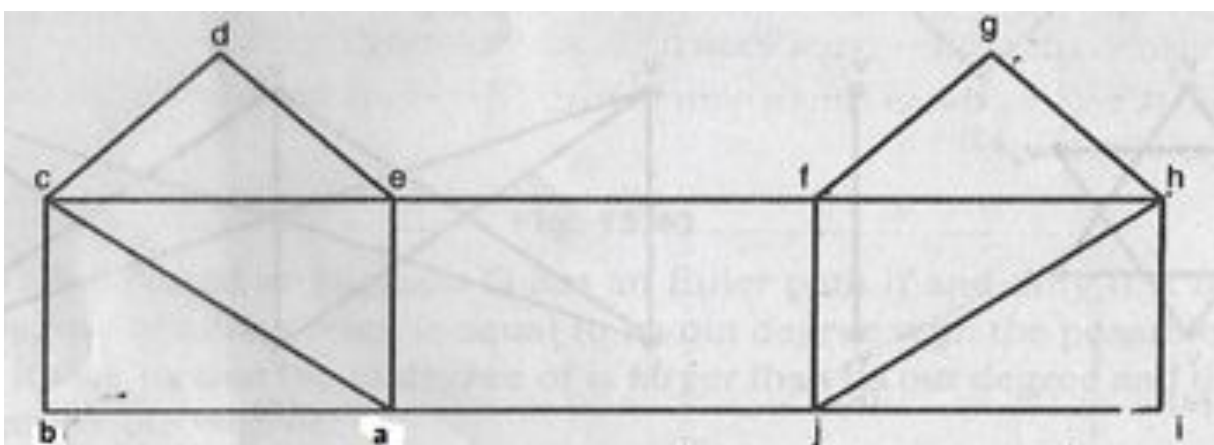
Fleury's algorithm, published in 1883, constructs Euler circuits by first choosing an arbitrary vertex of a connected multi-graph, and then forming a circuit by choosing edges successively. Once an edge is chosen, it is removed. Edges are chosen successively so that each edge begins where the last edge ends, and so that this edge is not a cut edge unless there is no alternative.

Let G=(V, E) be an Eulerian (each vertex has even degree) connected graph. Then use the following steps to construct an Eulerian circuit.

1. Select any vertex u from V as the starting vertex of Euler circuit π. Initialize π to u.
2. Select an edge $e = (u, v)$. If there are many such edges, **select one that is not a bridge**. Extend the path π to πv and set $E = E - \{e\}$. If e is a bridge (select only if there is no alternative), then set $V = V - \{u\}$. Now from vertex v proceed further.
3. Repeat step 2 until E=$\varphi. i.e. until\ there\ is\ no\ edge\ left$.

Note: the same algorithm can be used to find an Euler path with a modification in step 1. We know that when a graph has Euler path but not Euler circuit, it contains exactly two vertices of odd degree. Take one of the odd degree vertex as starting vertex and continue.

Example: use Fleury's algorithm to find Euler circuit in the graph given below:



Solution: since each edge has even degree, therefore Euler circuit exists. The steps of algorithm are shown as:

Current path	**Next Edge**	**Remark**
$\pi = a$	(a, j)	No edge from a is a bridge. Choose(a, j). Add $j\ to\ \pi$ and remove $(a, j) from\ E$.
$\pi = aj$	(j, f)	No edge from j is a bridge. Choose(j, f). Add $f\ to\ \pi$ and remove $(j, f) from\ E$.
$\pi = ajf$	(f, g)	$(f, g)\ is\ not\ a\ bridge.$ $other\ option\ is\ (f, h).$
$\pi = ajfg$	(g, h)	(g, h) is the only edge.
$\pi = ajfgh$	(h, i)	Other option is (h, i)
$\pi = ajfghi$	(i, j)	(i, j) is the only edge.
$\pi = ajfghij$	(j, h)	(j, h) is the only edge.
$\pi = ajfghijh$	(h, f)	(h, f) is the only edge.
$\pi = ajfghijhf$	(f, e)	(f, e) is the only edge.
$\pi = ajfghijhfe$	(e, d)	Other options are $(e, c)(e, a)$
$\pi = ajfghijhfed$	(d, c)	$(d, c)$is the only option.
$\pi = ajfghijhfedc$	(c, b)	Other options are $(c, e)(c, a)$
$\pi = ajfghijhfedcb$	(b, a)	(b, a) is the only option.
$\pi = ajfghijhfedcba$	(a, c)	Other options are (a, e)
$\pi = ajfghijhfedcbac$	(c, e)	(c, e) is the only option.
$\pi = ajfghijhfedcbace$	(e, a)	(e, a) is the only option.
$\pi = ajfghijhfedcbacea$	E=φ. No edge left now.	Euler circuit completed.

HAMILTONIAN PATHS AND GRAPHS

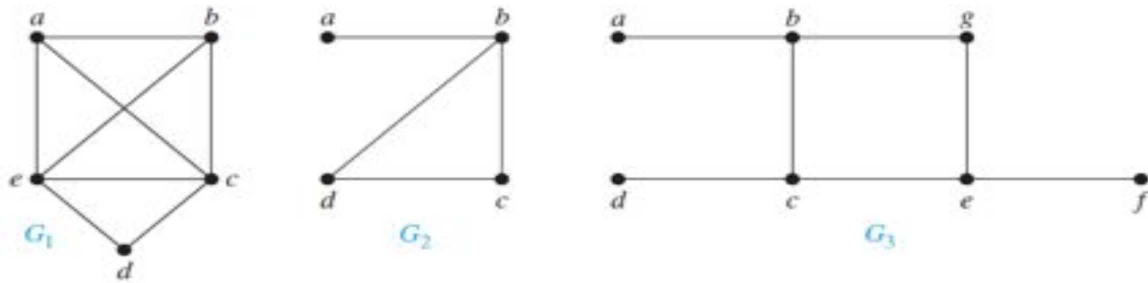
Hamiltonian path: is a **simple path** in a graph G that passes through every vertex exactly once.

Hamiltonian cycle/circuit: is a simple cycle in a graph G that passes through every vertex exactly once.

Hamiltonian graph: A graph that contains a Hamiltonian cycle is said to be Hamiltonian graph, otherwise nonhamiltonian. Note that,

i. K_n is Hamiltonian for $n \geq 3$.
ii. C_n is Hamiltonian for $n \geq 3$.
iii. W_n is Hamiltonian for $n \geq 3$.
iv. Q_n is Hamiltonian for $n \geq 2$.
v. $K_{m,n}$ is Hamiltonian for $m = n \geq 2$.

Examples: In the graphs given below, G1 has a Hamilton circuit: <a, b, c, d, e, a>. There is no Hamilton circuit in G2, but G2 does have a Hamilton path, namely, <a, b, c, d>. G3 has neither a Hamilton circuit nor a Hamilton path., because any path containing all vertices must contain one of the edges {a, b}, {e, f }, and {c, d} more than once.



Conditions for the existence of hamilton circuits:

There are no known simple necessary and sufficient criteria for the existence of Hamilton circuits. However, many theorems give sufficient conditions for the existence of Hamilton circuits. Two among them are:

1. 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 $\frac{n}{2}$, then G has a Hamilton circuit.
2. 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.

Note that, Ore's theorem and Dirac's theorem provide only sufficient conditions for a connected simple graph to have a Hamilton circuit. However, these theorems do not provide necessary conditions for the existence of a Hamilton circuit. That is, a graph may have Hamilton circuit even if none of these theorems hold good.

Also, certain properties can be used to show that a graph has **no** Hamilton circuit. For example, a graph with a vertex of degree one cannot have a Hamilton circuit, because in a Hamilton circuit, each vertex is incident with two edges in the circuit. Moreover, if a vertex in the graph has degree two, then both edges that are incident with this vertex must be part of any Hamilton circuit. Also, note that when a Hamilton circuit is being constructed and this circuit has passed through a vertex, then all remaining edges incident with this vertex, other than the two used in the circuit, can be removed from consideration. Furthermore, a Hamilton circuit cannot contain a smaller circuit within it.

Applications of hamilton circuits:

The best algorithms known for finding a Hamilton circuit in a graph or determining that no such circuit exists have exponential worst-case time complexity(Hamiltonian cycle problem is NP complete).

i. The famous traveling salesperson problem (TSP) reduces to finding a Hamilton circuit in a complete graph.
ii. Gray codes require finding a Hamilton circuit in Q_n.

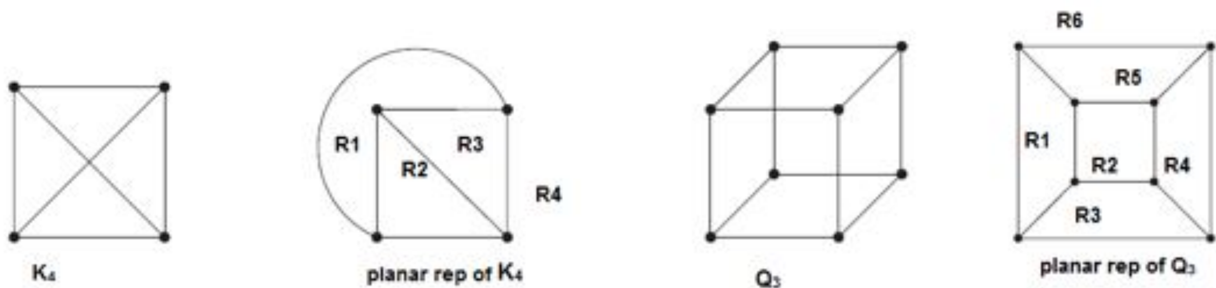
PLANAR GRAPHS

A graph is called planar if it can be drawn in the plane without any edges crossing (where a crossing of edges is the intersection of the lines or arcs representing them at a point other than their common endpoint). Such a drawing is called a planar representation of the graph.

A graph may be planar even if it is usually drawn with crossings, because it may be possible to draw it in a different way without crossings.

Note that, a planar representation of a graph splits the plane into regions, including an unbounded region (e.g. planar rep. of K_4 splits the plane in 4 regions R1, R2, R3 and R4 as shown).Further, all planar representations of a graph split the plane into the same number of regions.

Examples: K_4 and Q_3 are planar because these can be drawn without crossings, as shown:



Similarly, K_5 and $K_{3,3}$ are non-planar because these cannot be drawn in plane without edge crossings.

EULER'S FORMULLA: Let G be a **connected planar simple graph** with e edges and v vertices. Let r be the number of regions in a planar representation of G. Then $\boldsymbol{r = e - v + 2}$.

Given below are some corollaries of Euler's formula that must be satisfied by planar graphs. If a graph fails to satisfy any of these, then it is non-planar. These results can only be used to show that a graph is not planar (when any of the corollaries does not hold on that graph), they cannot be used to show that a graph is planar.

COROLLARY 1 of Euler's formula: If G is a connected planar simple graph with e edges and v vertices, where $v \geq 3$, then $\mathbf{e \leq 3v - 6}$.

Using corollary 1, we can show that **K_5 is not planar**.

COROLLARY 2 of Euler's formula: If G is a connected planar simple graph, then G has a vertex of degree not exceeding five.

COROLLARY 3 of Euler's formula: If a connected planar simple graph has e edges and v vertices with $v \geq 3$ and no circuits of length three, then $e \leq 2v - 4$.

Using corollary 3, we can show that **$K_{3,3}$ is not planar**.

We know that $K_{3,3}$ is not planar. Note, however, that this graph has six vertices and nine edges. This means that the inequality $e = 9 \leq 12 = 3 \cdot 6 - 6$ is satisfied. Consequently, the fact that the inequality $e \leq 3v - 6$ is satisfied does not imply that a graph is planar. Using corollary 3, we can show that $K_{3,3}$ is not planar.

GRAPH HOMEOMORPHISM AND KURATOWSKI'S THEOREM

Homeomorphic graphs: If a graph is planar, so will be any graph obtained by removing an edge {u, v} and adding a new vertex w together with edges {u, w} and {w, v}. Such an operation is called an **elementary subdivision**. The graphs G_1 = (V1,E1) and G_2= (V2,E2) are called **homeomorphic** if they can be obtained from the same graph by a sequence of elementary subdivisions.

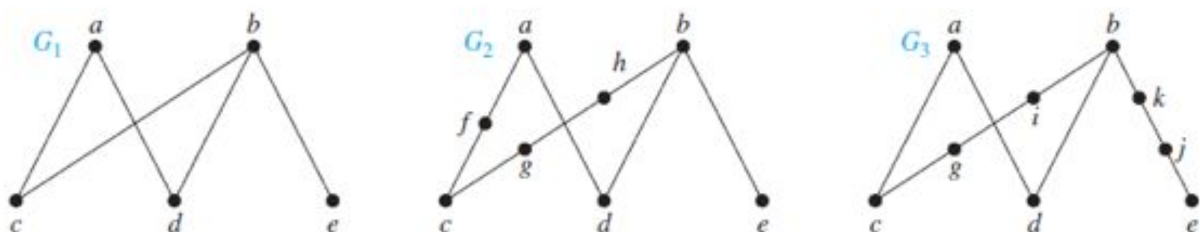
Consider the graphs G_1, G_2 and G_3 given below.

These three graphs are homeomorphic because all three can be obtained from G_1 by elementary subdivisions.

G_1 can be obtained from itself by an empty sequence of elementary subdivisions. To obtain G_2 from G_1 we can use this sequence of elementary subdivisions:

i. remove the edge {a, c}, add the vertex f , and add the edges {a, f } and {f, c};
ii. remove the edge {b, c}, add the vertex g, and add the edges {b, g} and {g, c}; and
iii. remove the edge {b, g}, add the vertex h, and add the edges {g, h} and {b, h}.

Similarly, G_3 can be obtained from G_1.

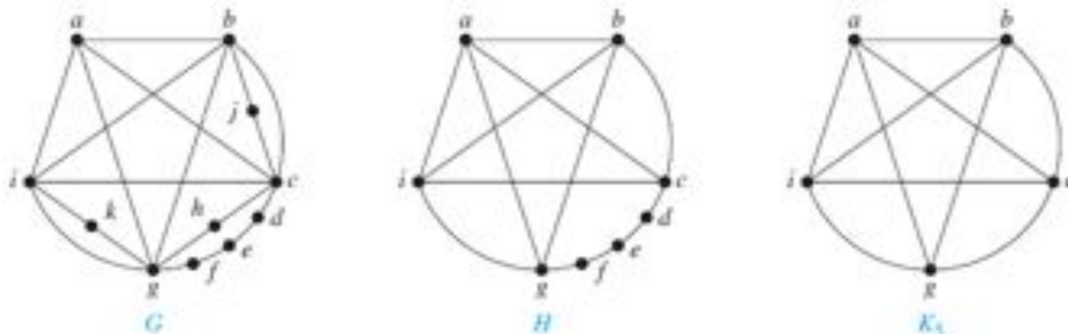


KURATOWSKI'S THEOREM:

A graph is **non-planar** if and only if it contains a sub-graph homeomorphic to $K_{3,3}$ or K_5.

Thus, all non-planar graphs must contain a sub-graph that can be obtained from $K_{3,3}$ or K_5 using certain permitted operations.

Example: consider the graph given below. We will show that this graph is not planar because its subgraph H is homeomorphic to K_5. H is obtained by deleting h, j , and k and all edges incident with these vertices. H is homeomorphic to K5 because it can be obtained from K5 (with vertices a, b, c, g, and i) by a sequence of elementary subdivisions, adding the vertices d, e, and f.



The Undirected Graph G, a Subgraph H Homeomorphic to K_5, and K_5.

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.

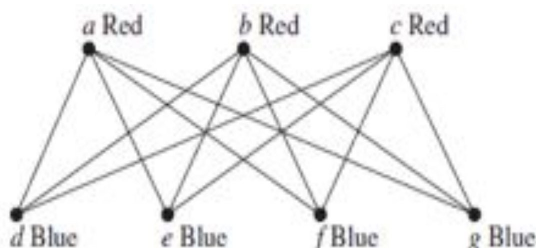
A graph can be colored by assigning a different color to each of its vertices. However, for most graphs a coloring can be found that uses fewer colors than the number of vertices in the graph. So the problem is to find the least number of colors required.

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

Because every two vertices of a complete graph are adjacent, the chromatic number of K_n is n. That is, $(K_n) = n$.

The chromatic number of a bipartite graph is **two** (because all vertices in one set can be assigned same color and all vertices in other set can also be assigned same color different from first). Thus, a bipartite graph is 2-colorable and if a graph is 2-colorable, then it is bipartite.

Because $K_{m,n}$ is a bipartite graph, its chromatic number is also equal to two i.e. $\chi(K_{m,n}) = 2$. This means that we can color the set of m vertices with one color and the set of n vertices with a second color. For example,



A Coloring of $K_{3,4}$.

Also note that, $\chi(C_n) = \begin{cases} 2, & \text{if n is an even positive integer with n } \geq 4 \\ 3, & \text{if n is an odd positive integer with n } \geq 3. \end{cases}$

$$\chi(W_n) = \begin{cases} 3, & \text{if n is even} \\ 4, & \text{if n is odd} \end{cases}$$

A graph G whose edge set E is empty has chromatic number equal to one.

THE FOUR COLORTHEOREM: The chromatic number of a planar graph is no greater than four.

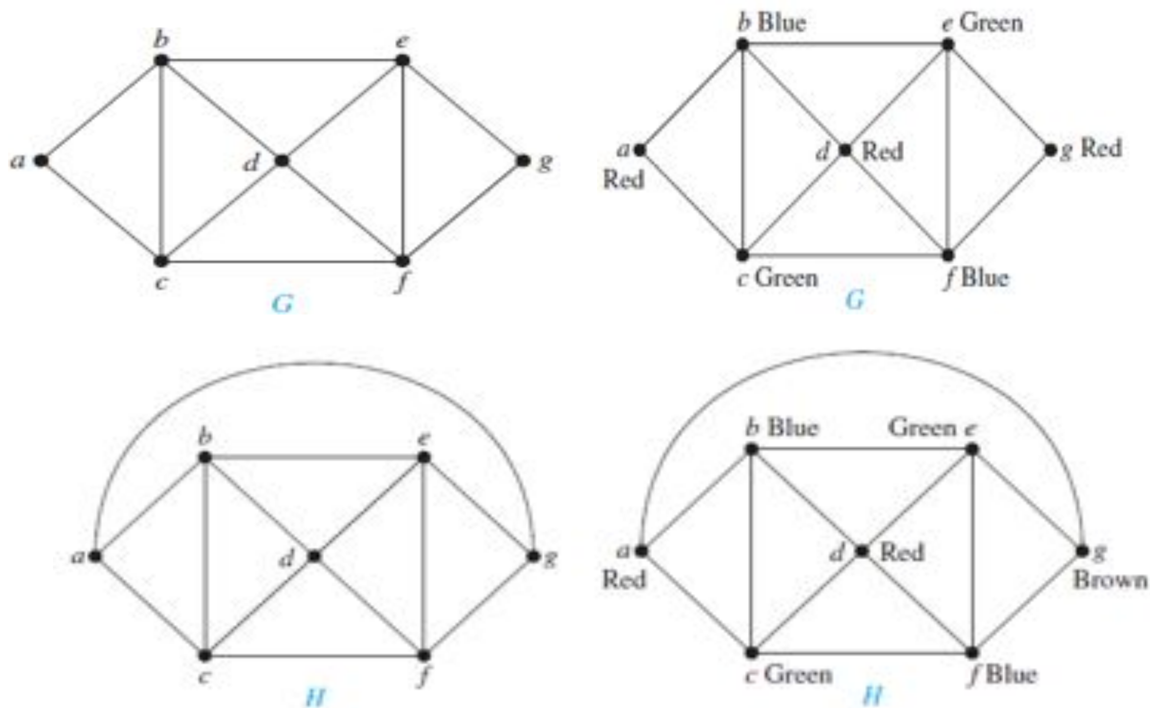
Two things are required to show that the chromatic number of a graph is k.

First, we must show that the graph can be colored with k colors. This can be done by constructing such a coloring. Second, we must show that the graph **cannot** be colored using fewer than k colors.

Examples: consider the following graphs:

The coloring of G using 3 colors and coloring of H using 4 colors is shown alongside the graphs G and H below. Note that, no two adjacent vertices are assigned the same color.

The minimum number of colors used for coloring of G is 3 i.e. $\chi(G) = 3$ because the vertices a, b, c must be assigned different colors. And minimum number of colors used for coloring of H is 4 i.e. $\chi(H)=4$.



BINARY OPERATION

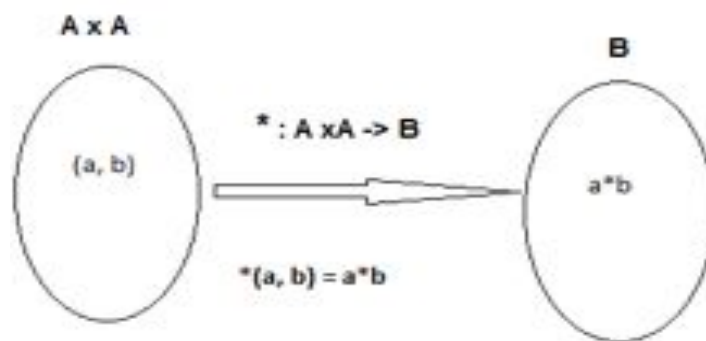
Let A and B be two non-empty sets, then a function from $A \times A$ to B is called a binary operation (or composition) on A.

If a binary operation on A is denoted by '*', then the unique element of B associated with the ordered pair (a, b) of $A \times A$ is denoted by a*b.

Thus, a binary operation '*' on A is a rule which associates every ordered pair (a, b) of $A \times A$, a unique element a*b of B.

i.e. *: $A \times A \rightarrow B$ such that *(a, b) = a*b ∀ (a, b) ∈ $A \times A$

Note: if B ⊆ A, the A is said to be closed under the binary operation.



Let A be a nonempty set and '*' be a binary operation on A, then

i. A is called **closed** under the operation '*' if and only if a*b ∈ A ∀ a, b ∈ A. (**Closure property)**
ii. The operation is called **commutative** if and only if a*b = b*a, ∀ a, b ∈ A.
iii. The operation is called **associative** if and only if (a*b)*c = a*(b*c), ∀ a, b, c ∈ A.
iv. The operation is called
    a. **left cancellative** if and only if a*b=a*c ⇒ b = c, ∀ a, b, c ∈ A.
    b. **Right cancellative** if b*a=c*a ⇒ b = c, ∀ a, b, c ∈ A.
    c. **Cancellative** if and only if it is left and right cancellative.
v. An element e is called **identity element** if and only if a*e = a = e*a ∀a ∈ A.
vi. An element $a \in A$ is called **inverse element** if and only if ∃ b ∈ A such that a*b = e = b*a. Here b is called inverse of A.

Example: Addition and multiplication are binary operations on the set N of natural numbers. Further, set N is closed with respect to addition and multiplication because the addition or multiplication of two natural numbers is a natural number. However, the set N is not closed under subtraction, for example 5-7=-2, which is an integer and ∉ N.

For set N of natural numbers there is no identity element for addition operation but 1 is identity element for multiplication operation.

A binary operation on a finite set can be represented by a table called composite table. For example, the composite table for a binary operation multiplication on a set S = {1, 2, 3} is shown as:

×	1	2	3
1	1	2	3
2	2	4	6
3	3	6	9

Reading the composite table, you can conclude the various properties/laws:

Closure property: If all entries in the table are elements of given set S, then S is closed under '*'. Note that the above set is not closed under multiplication, e.g. $9 \notin S$.

Commutative law: if transpose of the matrix representing the elements is same as the original table, then '*' is commutative. Note that the binary operation, multiplication on {1, 2, 3} holds commutative property.

Identity element: If the row headed by element 'a' of S coincides with the top row, then 'a' is identity element. In the above example, 1 is the identity element under multiplication.

Inverse element: if the table contains identity element 'e' at the intersection of row 'a' and column 'b', then b is inverse of a and we write $a^{-1} = b$. In the above example, table contains identity element 1 at only one place, that is, at intersection of row headed by 1 and column headed by element 1. Thus, 1 is inverse of 1.

ALGEBRAIC STRUCTURE

A nonempty set together with one or more binary operations is called algebraic structure. For example, (N, +), (Z, +), R(+, .) are algebraic structures. The identity element of any algebraic structure, if it exists, is unique. Similarly, the inverse of an element is unique, if it exists and inverse of identity element is the identity element itself.

Group

A nonempty set G, together with a binary operation '*' is called group, denoted by (G, *), if the following conditions are satisfied:

i. Closure law, i.e. $a*b \in G \ \forall a, b \in G$
ii. Associative law, i.e. $(a*b)*c = a*(b*c), \forall a, b, c \in G$.
iii. Existence of identity element e, i.e. $a*e = a = e*a \ \forall a \in G$
iv. Existence of inverse element a^{-1}, i.e. for every $a \in G$, $\exists a^{-1} \in G$, *such* that $a* a^{-1} = e = a^{-1} *a$. (here, $a^{-1} \neq \frac{1}{a}$)

Abelian Group (or Commutative Group) is named after the great Norwegian mathematician N. Able.

A group (G, *) is called Abelian group if it satisfies the commutative law, i.e. $a*b = b*a, \ \forall a, b \in G$.

Groupoid: A nonempty set S, together with a binary operation '*' is called **groupoid** if S is **closed** under ' *' i.e. $a*b \in S \ \forall a, b \in S$. For example, the algebraic structure (N, +) is a groupoid because the set N is closed under addition. But the set of odd integers under '+' is not a groupoid because it is not closed under addition, e.g. 5+5=10, which is even.

Semi-Group: A nonempty set S, together with a binary operation '*' is called **semi-group,** if the following conditions are satisfied:

i. Closure law, i.e. $a*b \in S \ \forall a, b \in S$
ii. Associative law, i.e. $(a*b)*c = a*(b*c), \forall a, b, c \in S$.

Note: every group is a semi-group but the converse may not be true. For example, (N, +) is semi-group but not a group because neither identity element nor inverse element exists for this algebra.

Similarly, (Z, +) and (Z, .) are semigroups.

The set Q of rational numbers is not a group under multiplication. Although the identity element for all rational numbers under multiplication is 1, but the inverse (or reciprocal) of zero is not defined.

Monoid:A nonempty set S, together with a binary operation '*' is called **monoid** if the following conditions are satisfied:

i. Closure law, i.e. $a*b \in S \ \forall a, b \in S$
ii. Associative law, i.e. $(a*b)*c = a*(b*c), \forall a, b, c \in S$.
iii. Existence of identity, i.e. for some $e \in S$, $a*e = a = e*a \ \forall a \in S$

Thus, a monoid is semigroup that has an identity element. For example, (Z, +) is a monoid with identity element 0 and (Z, .) is a monoid with 1 as identity element.

Cyclic Group

A group (G, *) is said to be cyclic if there exists an element 'a' in G such that $G=\{a^n : n \in Z\}$. The element 'a' is said to be the generator of the cyclic group. The cyclic group generated by a is denoted by the symbol [a] or (a). Also note that every cyclic group is Abelian.

PROPERTIES OF A GROUP

1. The identity element in a group is unique.
2. The inverse of each element of a group is unique.
3. If the inverse of a is a^{-1}, then the inverse of a^{-1} is a i.e. $(a^{-1})^{-1} = a$.
4. $(ab)^{-1} = b^{-1}a^{-1} \ \forall a, b \in G$.
5. Cancellation laws hold good in a group i.e.
    a. if $a*b=a*c \Rightarrow b = c, \forall a, b, c \in A$.(**left cancellation law**)
    b. if $b*a=c*a \Rightarrow b = c, \forall a, b, c \in A$. (**Right cancellation law**)
6. if a and b are any two elements of a group G, then the equations $a*x=b$ and $y*a=b$ have unique solutions in G. The solutions are $x=a^{-1}*b$ and $y= b* a^{-1}$.

Example 1. The set Z of all integers form an abelian group w.r.t. usual addition of integers.

Sol. To prove that (Z, +) is a group.

(*i*) **Closure axiom.** We know that sum of two integers is again an integer *i.e.* $a + b \in Z$ $\forall a, b \in Z$

$\Rightarrow$ Z is closed under addition.

(*ii*) **Associative law.** $(a + b) + c = a + (b + c)$ $\forall a, b, c \in Z$.

(*iii*) **Existence of identity.** $\exists\ 0 \in Z$ s.t. $a + 0 = a = 0 + a$ $\forall a \in Z$

$\therefore$ 0 is identity element of Z.

(*iv*) **Existence of inverse.** For each $a \in Z$, $\exists -a \in Z$ s.t.

$a + (-a) = 0 = (-a) + a$

$\therefore$ $-a$ is inverse of a.

(*v*) **Commutative law.** $a + b = b + a$ $\forall a, b \in Z$

Also Z contains infinite number of elements.

$\therefore$ (Z, +) is infinite abelian group.

Example 2. The set $Q^* = Q - \{0\}$ of all non-zero rational numbers form an abelian group w.r.t. usual multiplication of rational numbers.

Sol. (*i*) **Closure axiom.** We know that multiplication of two non-zero rational number is a non-zero rational number.

$\therefore ab \in Q^*$ $\forall a, b \in Q^*$.

(*ii*) **Associaitve law.** $(ab)\,c = a\,(bc)$ $\forall a, b, c \in Q^*$.

(*iii*) **Existence of identity.** $\exists\ 1 \in Q^*$ s.t. $a.1 = 1.a = a$ $\forall a \in Q^*$.

$\therefore$ 1 is identity element of Q^*.

(*iv*) **Existence of inverse.** For each $a \in Q^*$, $\exists\ \frac{1}{a} \in Q^*$

$$\text{s.t. } a.\frac{1}{a} = 1 = \frac{1}{a}.a$$

$\therefore$ $\frac{1}{a}$ is multiplicative inverse of a.

(v) commutative law: $ab = ba$, $\forall a, b \in Q^*$.

Thus, $(Q^*, \times)$ is an abelian group.

Example 3. Show that the set G = {1, –1, i, –i} where $i=\sqrt{-1}$, is an abelian group w.r.t. multiplication.

Sol. Give G = {1, –1, i, –i}

Firstly, we make the following composition table :

•	1	–1	i	$-i$
1	1	–1	i	$-i$
–1	–1	1	$-i$	i
i	i	$-i$	–1	1
$-i$	$-i$	i	1	–1

(*i*) **Closure Axiom.** Now $ab \in G \ \forall\ a, b \in G$. ...[From Table]

(*ii*) **Associative law.** $(ab)c = a(bc) \ \forall\ a, b, c \in G$.

...[∵ $(1.\,i)(-i) = i(-i) = 1$ & $1\,(i.\,-i) = 1.1 = 1 \Rightarrow (1\cdot i)(-i) = 1\cdot(i.\,(-i))$ similarly others]

(*iii*) **Existence of identity.** $\exists\ 1 \in G$ s.t.

$a\cdot 1 = a = 1\cdot a \ \forall\ a \in G$

∴ 1 is identity element of G.

(*iv*) **Existence of Inverse.** Since $1\cdot 1 = 1 \Rightarrow 1$ is inverse of 1

$(-1)(-1) = 1 \Rightarrow -1$ is inverse of –1

$i(-i) = 1 = (-i)\,i \Rightarrow i$ is inverse of $-i$

and $-i$ is inverse of i

Hence for each $a \in G$, $\exists\ b \in G$ s.t. $ab = 1 = ba$

∴ b is inverse of a.

(*v*) **Commutative law.** $ab = ba \ \forall\ a, b \in G$

Hence (G, •) is an abelian group.

Order of an element of a group: Let (G, *) be a group and let 'g' be an element of G. The order of an element 'g' in G, denoted by $O(g)$, is the **smallest** positive integer n such that $g^n = e$. If no such integer exists, then 'g' has infinite order.

Thus, to find order of a group element 'g', compute the sequence of products $g,\ g^2,\ g^3, g^4$,...until an identity element is obtained first time. The exponent of 'g' at this time gives the order of 'g'.

Example: Let G= {1, -1, *i, -i* } be a multiplicative group. Find the order of every element? (multiplicative group means binary operation is multiplication)

Solution: Clearly, 1 is identity element in this group.

i. $1^1 = 1 \Rightarrow O(1) = 1$
ii. $-1^2 = 1 \Rightarrow O(-1) = 2$
iii. $i^4 = i^2 \times i^2 = -1 \times -1 = 1 \Rightarrow O(i) = 4$
iv. $-i^4 = -i^2 \times -i^2 = 1 \times 1 = 1 \Rightarrow O(-i) = 4$

Integral powers of an element:

$a^0 = e$, $a^1 = a$, ..., $a^n = a*a*...*a$.(n factors)

$a^{-n} = (a^{-1})^n = a^{-1}*a^{-1}*...*a^{-1}$(n factors)]

Properties of order:

i. $O(a) = O(a^{-1})$
ii. if $O(a) = n$, and $e^m = e$, then n is divisor of m.
iii. If $O(a) = n$, then a, a^2, ... , a^n(=e) are distinct elements of G.
iv. If $O(a) = n$ and p is prime to n, then $O(a^p) = n$.
v. If $O(a)$ is infinite and p be a positive integer, then $O(a^p)$ *is infinite*.

Order of a group: The number of elements in a group is called order of a group, denoted by O(G). A group which contains finite number of elements is called finite group, otherwise infinite group.

Subgroup

Let (G, *) be a group and H be a nonempty subset of G. Then, (H, *) is said to be subgroup of G if (H, *) is also a group by itself.

Because every set is a subset of itself, every group is a subgroup of itself. Further, the subset of G containing only e (identity element) is also a subgroup of G. These two subgroups (G, *) and ({e}, *) of the group (G, *) are called trivial/ improper subgroups, others are called proper or non-trivial subgroups.

Example: The group ({1,-1}, .) i.e. multiplicative group on {1,-1} is a subgroup of multiplicative group ({1, -1, i, -i}, .)

Similarly, the additive group of **even** integers is a subgroup of the additive group of all integers.

Properties of Subgroups:

i. The identity element of a subgroup is same as that of the group.
ii. For any a ∈ H, if a^{-1} is the inverse of a in G, then a^{-1} is also the inverse of a in H.
iii. A nonempty subset H of a group (G, *) is a subgroup of G if and only if
    a. for all a, b ∈ H, a*b ∈ H.
    b. for all a ∈ H, a^{-1} ∈ H.

iv. Let (G, *) be group, then a nonempty subset H of G forms a subgroup of (G, *) if and only if $a \in H$ and $b \in H$ imply $a*b^{-1} \in H$. (this is a necessary and sufficient condition for subset H to be a subgroup of G)

v. Let (G, *) be group and H be a nonempty subset of G. If H is closed under '*', then H forms a subgroup of (G, *).

vi. Let H and K be **subgroups** of (G, *). Then $H \cap K$ is a subgroup of (G, *) but $H \cup K$ is not necessarily a subgroup.

vii. Let G be a group and a be an element of G. Then, the set $\{a^n : n \in Z\}$ forms a subgroup of G and it is the smallest subgroup of G containing the element a.

viii. Every subgroup of a cyclic group is cyclic.

Cosets

Let H be a **subgroup** of a group (G, *) and let $a \in G$. Then the set $\{a*h : h \in H\}$ is called the **left coset** generated by a and H and is denoted by aH.

Thus, $aH = \{a*h : h \in H\}$.

Similarly, $Ha=\{h*a : h \in H\}$ is called the **right coset**.

Both aH and Ha are subsets of G. Also, if e be the identity element of G, then $e \in H$ and He=H=eH. Therefore, H is itself a left as well as right coset.

Any left coset of an abelian group is equal to the corresponding right coset.

In additive notation, a left coset is denoted by $a+H = \{ a+h: h \; h \in H \}$ and right coset by $H+a = \{ h+a: h \; h \in H \}$

Example: Let G be an additive group of integers i.e G = ({..., -3, -2, -1, 0, 1, 2, 3, ...}, +)

And let H be a **subgroup** of G obtained by multiplying each element of G by 3,

then H=({..., -9, -6, -3, 0, 3, 6, 9, ...}, +)

The right and left cosets of H in G are:

For $0 \in G$, H+0 = {..., -9, -6, -3, 0, 3, 6, 9, ...}, and 0+H = H+0 = {..., -9, -6, -3, 0, 3, 6, 9, ...}

For $1 \in G$, H+1= {..., -8, -5, -2, 1, 4, 7, 10, ...} and 1+H = {..., -8, -5, -2, 1, 4, 7, 10, ...}

For $2 \in G$, H+2 = {..., -7, -4, -1, 2, 5, 8, 11,...} and 2+H ={..., -7, -4, -1, 2, 5, 8, 11,...}

Note that, the left or right cosets are all distinct and disjoint sets.

Since the group G is Abelian (easy to check), that is why any left coset is equal to the corresponding right coset.

Properties of cosets

Let H be a subgroup of G and let a, b ∈ G. Then,

i. a∈ aH.
ii. aH = bH or aH ∩ bH=φ(empty set).
iii. aH = bH if and only if $a^{-1}b$ ∈H.

Normal Subgroup

A subgroup H of a group G is said to be a normal subgroup of G if Ha=aH, for all a ∈ G. Clearly, every subgroup of an Abelian group is normal subgroup.

HOMOMORPHISM AND ISOMORPHISM OF GROUPS

Let (G, ∘) and (G', *) be two groups. A mapping (function) ϕ: (G, ∘) → (G', *) is said to be homomorphism if $\phi(a \circ b) = \phi(a) * \phi(b)$ for all a,b ∈ G.

A **ono-to-one** homomorphism is said to be **monomorphism.**

An **onto** homomorphism is said to be **epimorphism .**

A homomorphism is said to be **isomorphism** if and only if it is both one-to-one and onto i.e. both monomorphism and epimorphism. Thus, if (G, ∘) and (G', *) are two groups, then a mapping ϕ: (G, ∘) → (G', *) is called isomorphism if

i. ϕ *is homomorphism.*
ii. ϕ *is one − to − one*
iii. ϕ *is onto.*

Two groups (G, ∘) and (G', *) are said to be isomorphic if there exists an isomorphism ϕ: (G, ∘) → (G', *) and symbolically we write G ≅ G'.

If the group G is finite, then G can be isomorphic to G' if and only if G' is finite and O(G') = O(G). Further, If G ≅ G', then G and G' are abstractly identical and there is no difference between them.

Properties of isomorphism:

Suppose that f is an isomorphic mapping from G into G'. Then,

if e is the identity of G, then $f(e)$ is the identity of G'.

if e is the identity of G and e' is the identity of G', then $f(e) = e'$.

$f(a^{-1}) = [f(a)]^{-1}, a \in G$.

The order of an element a of G is equal to the order of its image $f(a)$.

Finally, if we want to prove that group G is isomorphic to G', then we must find a mapping f: G->G' which is one-one and onto and also preserves compositions.

Example 2. If R is the additive group of real numbers and R^+ is the multiplicative group of positive reals, prove that mapping $f : R \to R^+$ defined by $f(x) = e^x \ \forall \ x \in R$ is an isomorphism of R onto R^+.

Sol. If x is any real no, positive, zero or negative, then e^x is always a positive number. Also e^x is unique. Therefore if $f(x) = e^x$, then $f : R \to R^+$

(*i*) ***f* is one one.**

Suppose $x_1, x_2 \in R$ and

$$f(x_1) = f(x_2)$$

$$\therefore \quad e^{x_1} = e^{x_2}$$

$$\Rightarrow \quad \log e^{x_1} = \log e^{x_2} \Rightarrow x_1 \log e = x_2 \log e$$

$$\Rightarrow \quad x_1 = x_2$$

Thus the two elements in R have the same f-images in R^+ only if they are equal. Consequently different elements in R have different images in R^+. Therefore f is one one.

(*ii*) **f is onto.** Let $y \in R^+$ be any element

$\therefore$ y is positive real number.

Then $\log y$ is a real no. $\in R$. Then $f(\log y) = e^{\log y} = y$. Therefore for any positive $y \in R^+$, $\exists \log y \in R$ such that $f(\log y) = e^{\log y} = y \Rightarrow f$ is onto.

(*ii*) ***f* is homomorphism**

Let x_1 and $x_2 \in R$

$$\therefore \quad f(x_1 + x_2) = e^{x_1 + x_2} = e^{x_1} \cdot e^{x_2}$$

$$= f(x_1) f(x_2)$$

$\Rightarrow f : R \to R^+$ an isomorphism of R onto R^+.

Hence $R \cong R^+$.

Example 3. Let R^+ be the multiplicative group of all positive real numbers and R be the additive group of all real numbers. Show that the mapping $g : R^+ \to R$ defined by $g(x) = \log x \ \forall \ x \in R^+$ is an isomorphism.

Sol. If x is any positive real, then $\log x$ is a real number and also $\log x$ is unique. Therefore if

$$g(x) = \log x, \text{ then } g : R^+ \to R$$

(*i*) ***f* is one-one**

Let $x_1, x_2 \in R^+$ such that

$$g(x_1) = g(x_2)$$

$$\Rightarrow \quad \log x_1 = \log x_2 \Rightarrow e^{\log x_1} = e^{\log x_2}$$

$$\Rightarrow \quad x_1 = x_2$$

$\Rightarrow$ g is one one

(*ii*) ***f* is onto**

Let y be any real number of R.

$\therefore$ e^y is always a positive real $\in R^+$

$$\therefore \quad g(e^y) = \log e^y = y \log e = y$$

therefore for any $y \in R$, $\exists \ e^y \in R^+$ such that $g(e^y) = \log e^y = y \log e = y$

$\Rightarrow$ mapping is onto.

(*iii*) ***f* is homomorphism**

Let x_1 and x_2 be any two elements $\in R^+$

$$\therefore \quad g(x_1 x_2) = \log(x_1 x_2) = \log x_1 + \log x_2 = g(x_1) + g(x_2)$$

$\Rightarrow$ g is an isomorphism of R^+ onto R.

$$R^+ \cong R.$$

Example 4. Show that the additive group of integers G = {......., – 3, – 2, – 1, 0, 1, 2, 3, ...} is isomorphic to the additive group.

G′ = {........., – 3m, – 2m, – m, 0, m, 2m, 3m, ...}

where m is any fixed integer not equal to zero.

Solution. If $x \in G$, then $mx \in G'$ is fixed integer not equal to zero.

Let $f : G \to G'$ defined by $f(x) = mx \ \forall \ x \in G$

(i) f is one-one

Let $x_1, x_2 \in G$ such that

$$f(x_1) = f(x_2)$$

$$\therefore \quad mx_1 = mx_2 \Rightarrow x_1 = x_2$$

$\Rightarrow$ f is one-one.

***(ii) f* is onto**

Let y be any element $\in G'$ $\therefore \exists \ \frac{y}{m} \in G$ s.t.

$$f\left(\frac{y}{m}\right) = m\left(\frac{y}{m}\right) = y$$

$\Rightarrow$ f is onto

***(iii) f* is homomorphism**

Let x_1, x_2 be any two elements $\in G$.

$$\therefore \quad f(x_1 + x_2) = m(x_1 + x_2) \quad \text{[By def. of } f\text{]}$$

$$= mx_1 + mx_2$$

$$= f(x_1) + f(x_2)$$

Therefore, f is homomorphism.

Hence G is isomorphic to G′.

FREE TREES

A free tree is a **connected, acyclic, undirected** graph (The adjective free is often omitted when we say that a graph is a tree).

If an undirected graph is acyclic but possibly disconnected, it is called a **forest**. Each connected component of a forest is a tree. Thus, every tree is a forest (containing only one tree) but every forest is not a tree.

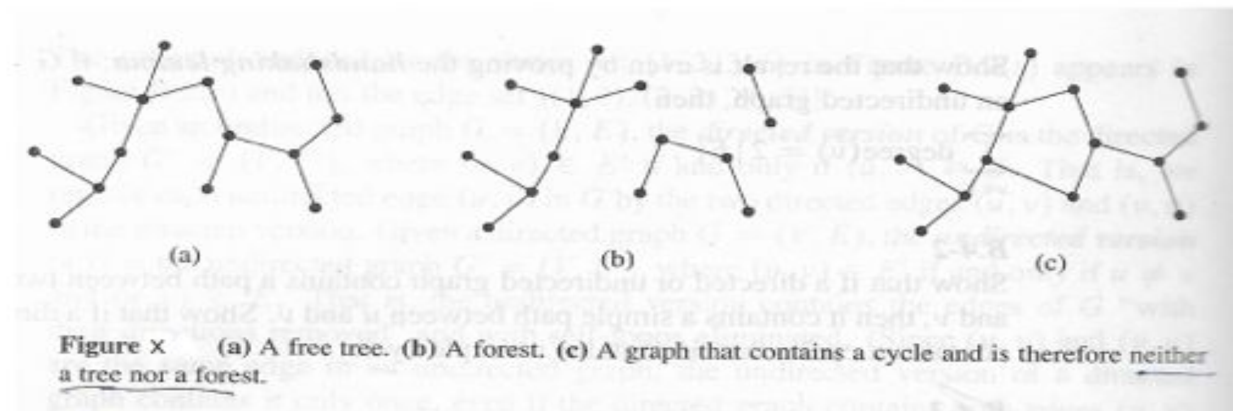


Figure x (a) A free tree. (b) A forest. (c) A graph that contains a cycle and is therefore neither a tree nor a forest.

PROPERTIES OF FREE TREES

Let $G = (V, E)$ be an undirected graph on n vertices (or nodes) and e edges, then the following statements are equivalent:

i. G is a free tree.
ii. Any two vertices in G are connected by a **unique** simple path (i.e. G is connected and there is one and only one path between any vertex pair).
iii. G is connected but if any edge is removed from E, the resulting graph is disconnected.
iv. G is connected and $e = n - 1$.
v. G is acyclic and $e = n - 1$.
vi. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle.

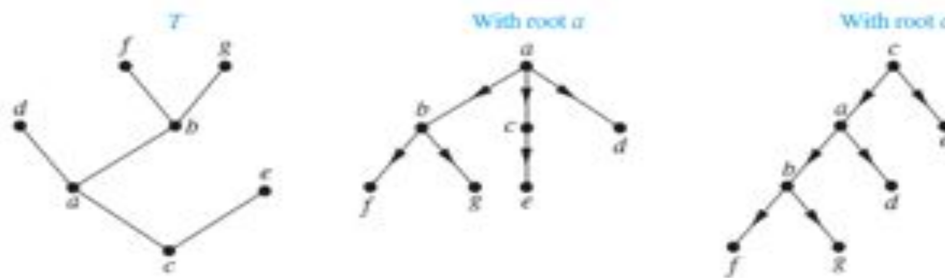
ROOTED TREES

A rooted tree is a free tree in which one of the vertices is distinguished from the others. The distinguished vertex is called the root of the tree.

Once we specify a root, we can assign a direction to each edge. Because there is a unique path from the root to each vertex of the graph, we direct each edge away from the root. Thus, a tree together with its root produces a directed graph called a rooted tree.

We can change an unrooted tree into a rooted tree by choosing any vertex as the root. Note that different choices of the root produce different rooted trees. For example, consider the tree T and two rooted trees produced from this tree shown below. The arrows indicating the directions of the

edges in a rooted tree can be omitted, because the choice of root determines the directions of the edges.



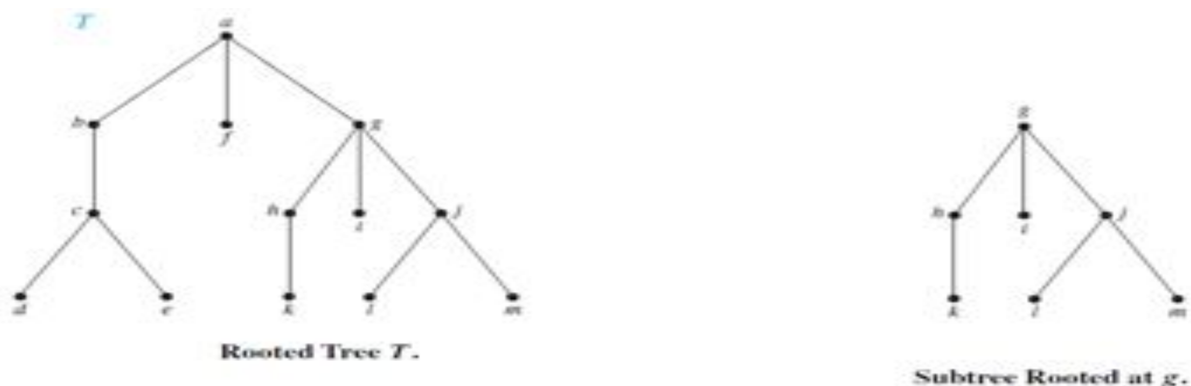
A Tree and Rooted Trees Formed by Designating Two Different Roots.

Suppose that T is a rooted tree. If v is a vertex in T other than the root, the parent of v is the unique vertex u such that there is a directed edge from $u \ to \ v$. When u is the **parent** of v, v is called a **child** of u. Root has no parent. Vertices with the same parent are called **siblings.** The **ancestors** of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root (that is, its parent, its parent's parent, and so on, until the root is reached). The **descendants** of a vertex v are those vertices that have v as an ancestor. A vertex of a rooted tree is called a **leaf** if it has no children. Vertices that have children are called **internal vertices**. The root is an internal vertex unless it is the only vertex in the graph, in which case it is a leaf.

If a is a vertex in a tree, the subtree with a as its root is the subgraph of the tree consisting of a and its descendants and all edges incident to these descendants.

Consider the following rooted tree.

In this tree, the parent of c is b. The children of g are h, i, and j. The siblings of h are i and j. The ancestors of e are c, b, and a. The descendants of b are c, d, and e. The internal vertices are a, b, c, g, h, and j . The leaves are d, e, f , i, k, l, and m. The **subtree** rooted at g is shown alongside.



Rooted Tree T.

Subtree Rooted at g.

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

DEGREE, HEIGHT, LEVEL, DEPTH

Degree: The **number of children** of a node (or a vertex) in a rooted tree is referred to as degree of a node.

Depth: The length of the simple path from the root to a node is called the depth of a node. The depth of root node is zero.

Level: A level of a tree consists of all nodes at the same depth. (Some authors use level instead of depth i.e. the level of a vertex v in a rooted tree is the length of the unique path from the root to this vertex).

Height: the height of a node in a tree is the number of edges on the **longest** simple downward path from the node to a leaf and the height of tree is the height of its root. The height of a tree is also equal to the largest (maximum) depth of any node in the tree. The height of a leaf node is zero.

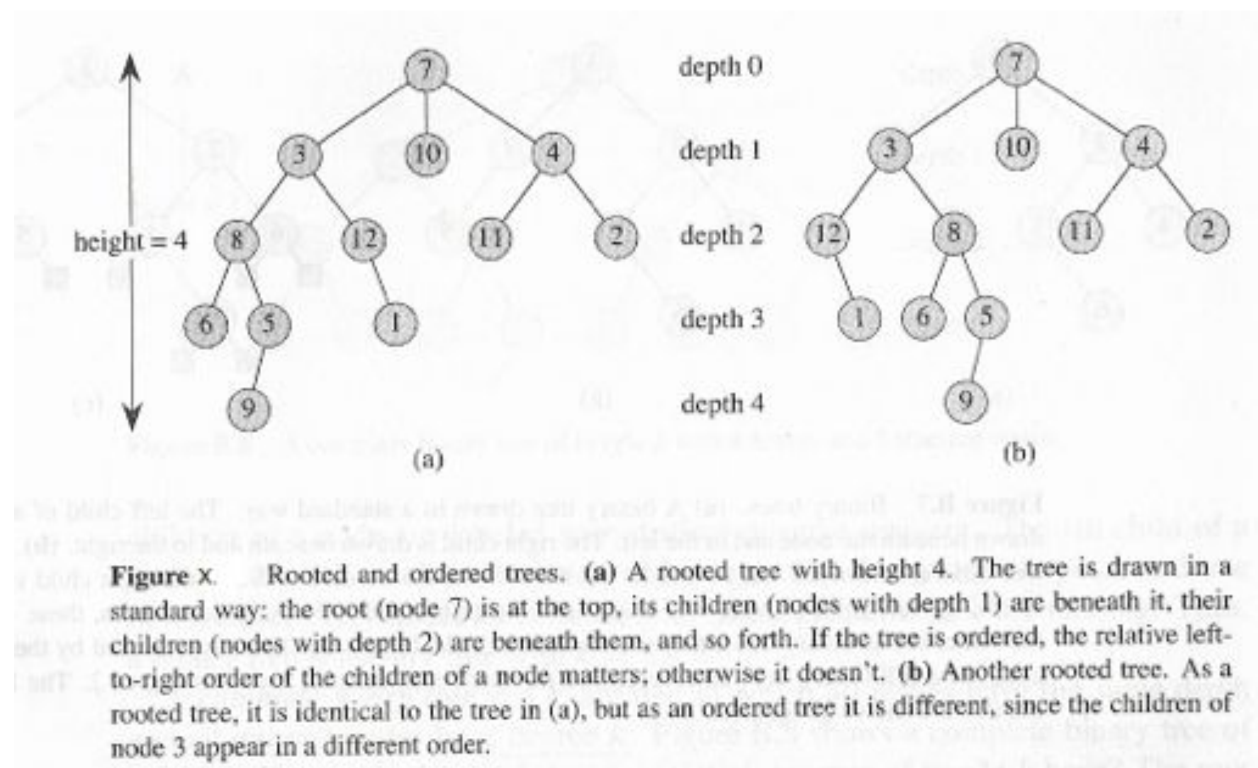


Figure X. Rooted and ordered trees. **(a)** A rooted tree with height 4. The tree is drawn in a standard way: the root (node 7) is at the top, its children (nodes with depth 1) are beneath it, their children (nodes with depth 2) are beneath them, and so forth. If the tree is ordered, the relative left-to-right order of the children of a node matters; otherwise it doesn't. **(b)** Another rooted tree. As a rooted tree, it is identical to the tree in (a), but as an ordered tree it is different, since the children of node 3 appear in a different order.

ORDERED TREES

An ordered tree is a rooted tree in which the children of each internal vertex are ordered. Ordered trees are drawn so that the children of each internal vertex are shown in order from left to right. That is, if a vertex (or node) has k children, then there is first child, a second child, . . ., a kth child.

In an ordered binary tree (usually called just a **binary tree**), if an internal vertex has two children, the first child is called the left child and the second child is called 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 the vertex.

BINARY TREE

A binary tree is a rooted tree in which each internal vertex has **at most two children**. Each child is designated as either **left child** or **right child.**

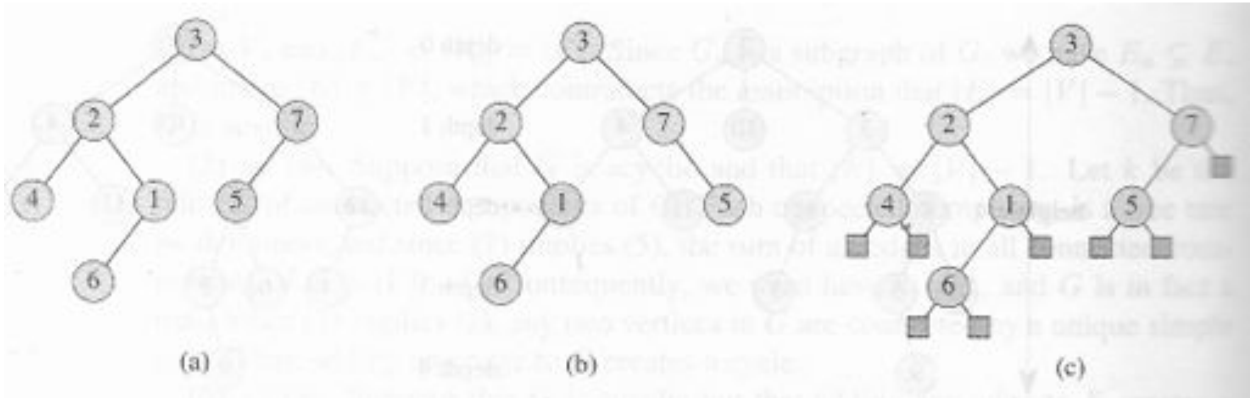


Figure X: Binary trees (a) A binary tree drawn in a standard way. The left child of a node is drawn beneath the node and to the left. A right child is drawn beneath and to the right. (b) A Binary tree different from the one in (a). In (a), the left child of a node 7 is 5 and the right child is absent. In (b), the left child of node 7 is absent and the right child is 5. As ordered trees, these trees are the same, but as binary trees, they are distinct.(c) The binary tree in (a) represented by the internal nodes of a full binary tree: an ordered tree in which each internal node has degree 2.The leaves in the tree are shown as squares**.**

A **full binary** is a tree in which each internal node has exactly two children. That is, each node has either zero children or two children.

A **complete binary** tree is a binary tree in which all leaves have the same depth and degree of all internal nodes is two (degree= number of children).

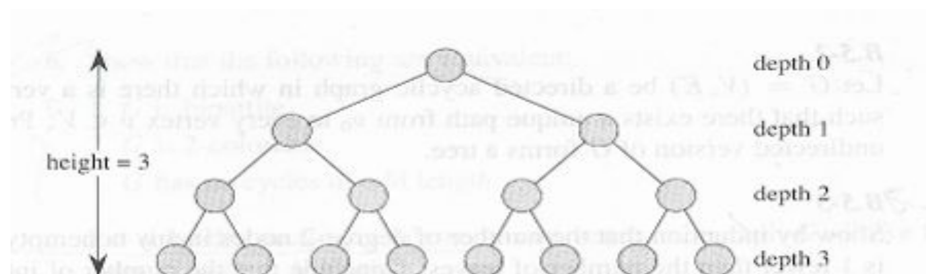


Figure X. A complete binary tree of height 3 with 8 leaves and 7 internal nodes.

<u>Complete *k-ary* tree</u> is a tree in which all leaves have the same depth and all internal nodes have degree k. To count the number of leaf nodes in a k-ary tree of height h, note that the root has k children at depth 1, each of the nodes at depth 1 has k children at depth 2 and so on. Thus, the number of leaves at depth h is k^h. Consequently, the height of a complete k-ary tree with n leaves is $\log_k n$.The number of internal nodes of a complete k-ary tree of height h is

$1 + k + k^2 + \ \ldots + k^{h-1} = \frac{k^h - 1}{k-1}$.

Therefore, a complete binary tree (k=2) has $2^h - 1\ internal\ nodes$. Thus, total nodes= $2^h + 2^h - 1 = 2^{h+1} - 1$.

Properties of binary trees:

i. The number of nodes/vertices in a binary tree is always odd.
ii. The maximum number of vertices at depth h of a binary tree is 2^h.
iii. The maximum number of vertices in a binary tree of height h is $2^{h+1} - 1$.
iv. The number of internal nodes **in a full binary** tree is 1 fewer than the number of leaves.

SOME IMPORTANT RESULTS (consider that a tree is rooted, because we can always designate one of the vertices as root).

i. A tree with n vertices contains $n - 1$ edges ($e = n - 1$).
ii. A forest with n vertices and having t trees in it, contains a total of $(n - t)$ edges.
    Proof: Suppose a tree T_i $(i = 1, 2, \ldots, t)$ contains x_i vertices. Then number of edges in each tree $T_i = (x_i - 1)$. Therefore, total number of edges in this forest $= \sum_{i=1}^{t}(x_i - 1) = \sum_{i=1}^{t} x_i + (-1)(t\ times) = n - t$.
iii. A full $m - ary$ tree with i internal vertices contains $\boldsymbol{n = mi + 1}$ vertices.
iv. A full $m - ary$ tree with
    a. n vertices has $i = (n - 1)/m$ internal vertices and $l = [(m - 1)n + 1]/m$ leaves,
    b. i internal vertices has $n = mi + 1$ vertices and $l = (m - 1)i + 1$ leaves,
    c. l leaves has $n = (ml - 1)/(m - 1)$ vertices and $i = (l - 1)/(m - 1)$ internal vertices.
v. There are at most m^h leaves in an *m-ary* tree of height h.
vi. If an *m-ary* tree of height h has l leaves, then $h \geq cieling(\log_m l)$. If the *m-ary* tree is full and balanced, then $h = cieling(\log_m l)$ (ceiling(x) returns the smallest integer greater than or equal to x.)

BINARY SEARCH TREE

Binary search tree is a binary tree in which each vertex is labeled with a **key**. The vertices are assigned keys so that the key of a vertex is both larger than the keys of all vertices in its left subtree and smaller than the keys of all vertices in its right subtree.

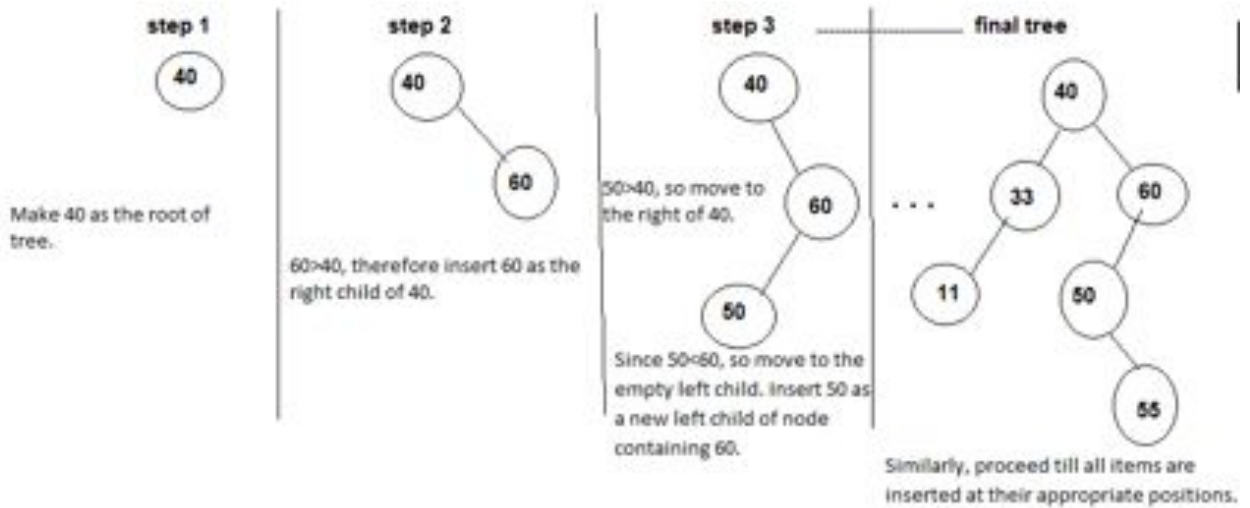
Constructing a binary search tree for a given list of items:

1. Make the first item in the list as root node N.
2. Compare the subsequent item with the root node N
    a. If item < N, move to the left child of N.
    b. If item ≥ N, move to the right child of N.
3. Repeat step 2 until you find an empty left or right subtree. Insert the item in place of empty subtree.

Note: Binary search tree provides efficient searching method. With slight modification to above steps, you can search an element in a binary search tree. Given an element to be searched, follow step 2 repeatedly until either you find a match or search fails.

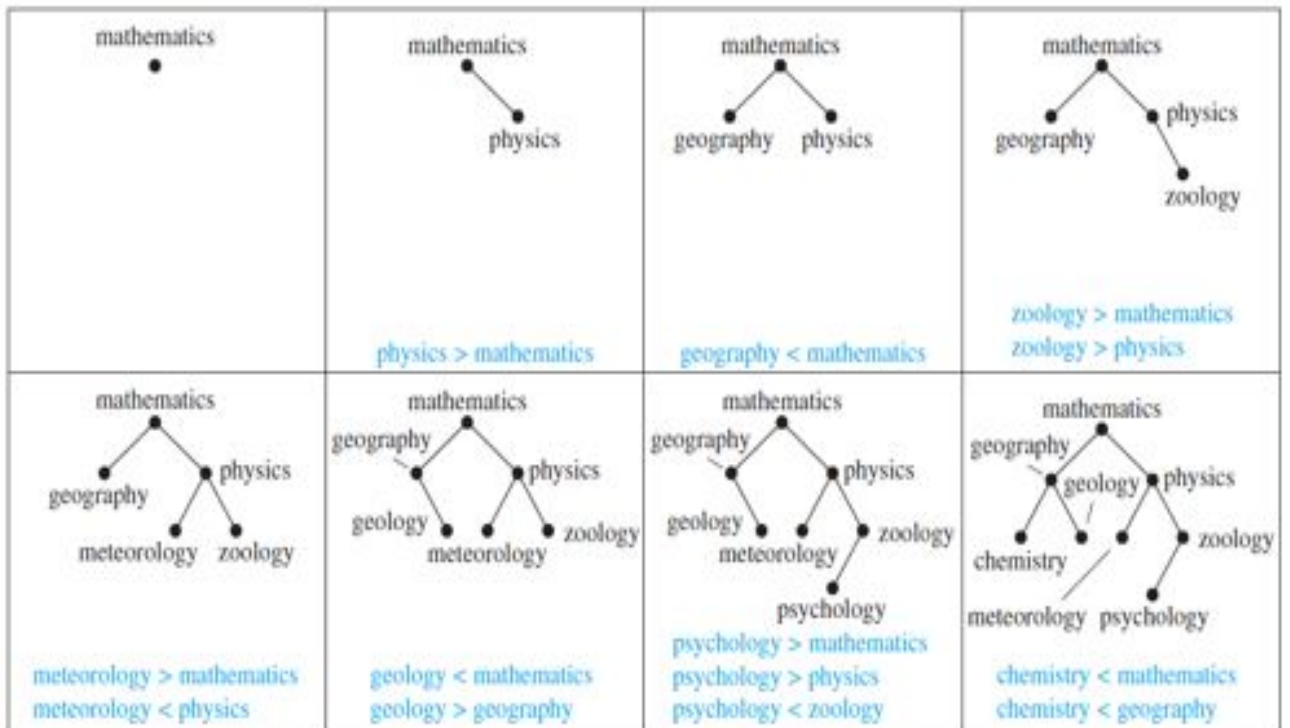
Example: construct the binary search tree for following numbers in the this order: 40, 60, 50, 33, 55, 11.

Solution: Note that inorder traversal of final tree yields sorted list.



Example 2: Form a binary search tree for the words: mathematics, physics, geography, zoology, meteorology, geology, psychology, and chemistry (using alphabetical order).

Solution:



PREFIX CODES/HUFFMAN CODES

Prefix codes are used in data compression. Data is a sequence of characters. To represent this data, we can represent each character by a binary string (codeword). There are two possible ways:

i. Fixed-length code
ii. Variable-length code

To illustrate, consider a 100,000-character data file that we wish to store compactly. Suppose the characters in the file occur with the frequencies given in following table:

character	**a**	**b**	**c**	**d**	**E**	**f**
Frequency(in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

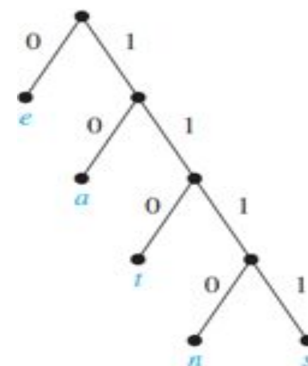
If we use a fixed-legth code, we need 3 bits to represent 6 characters: a= 000, b= 001, ..., f=101. This method requires 300,000 bits to code the entire file.

A variable length code can do considerably better than a fixed length code, by giving frequent characters short codewords and infrequent characters long codewords. The above figure shows one such code. Here, a= 0, b= 101 and so on. This code requires

$(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 224{,}000\ bits.$ to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file. This shows the significance of variable-length codes such as prefix or Huffman codes.

Prefix codes are variable length codes in which no codeword occurs as a prefix of some other codeword. A prefix code always gives the optimal data compression among any character code.

A prefix code can be represented using a **binary tree**, where the characters (or symbols) are the labels of the leaves in the tree. The edges of the tree are labeled so that an edge leading to a left child is assigned a 0 and an edge leading to a right child is assigned a 1. 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. For instance, the tree in Figure given alongside represents the encoding of e by 0, a by 10, t by 110, n by 1110, and s by 1111.



An optimal code for a file is always represented by a full binary tree (in which every internal node has two children).

Prefix codes simplify **decoding** because no codeword is a prefix of any other. Simply use the following steps to decode a bit string back into character string:

1. Start from root and traverse along edges as per bits in the bit string (i.e. if 1 bit is encountered move right, otherwise left).
2. Stop when leaf is encountered. The leaf-label gives the character encoded by the bit substring.
3. Start again from the root, continuing with the next bit in the bit string.

Executing the above steps on bit string 11111011100, we obtain the character string “sane”.

Algorithm: Constructing a Huffman code/ prefix code.

Given a set of symbols (or characters) and their frequencies, the objective is to construct a binary tree where the symbols are labels of leaf nodes. Follow steps:

1. Construct a forest of trees each consisting of one vertex, where each vertex has a symbol as its label and where the **weight of this vertex** equals the frequency of the symbol.
2. Combine two trees having the **least total weight** into a single tree by introducing a **new root** and placing the tree with larger weight as its left subtree and the tree with smaller weight as its right subtree. Furthermore, assign the sum of the weights of the two subtrees of this tree as the total weight of the tree.
3. **Repeat step 2** until the forest is reduced to a single tree. The algorithm is finished when it has constructed a tree

Huffman coding is greedy algorithm. Its time complexity on a set of n characters is $O(n \log n)$.

Example: 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?

Solution: Figure X given below illustrates the steps of algorithm.

The encoding produced encodes A by 111, B by 110, C by 011, D by 010, E by 10, and F by 00. The average number of bits used to encode a symbol using this encoding is

$$3 \cdot 0.08 + 3 \cdot 0.10 + 3 \cdot 0.12 + 3 \cdot 0.15 + 2 \cdot 0.20 + 2 \cdot 0.35 = 2.45.$$

(Constructing Huffman codes, example)

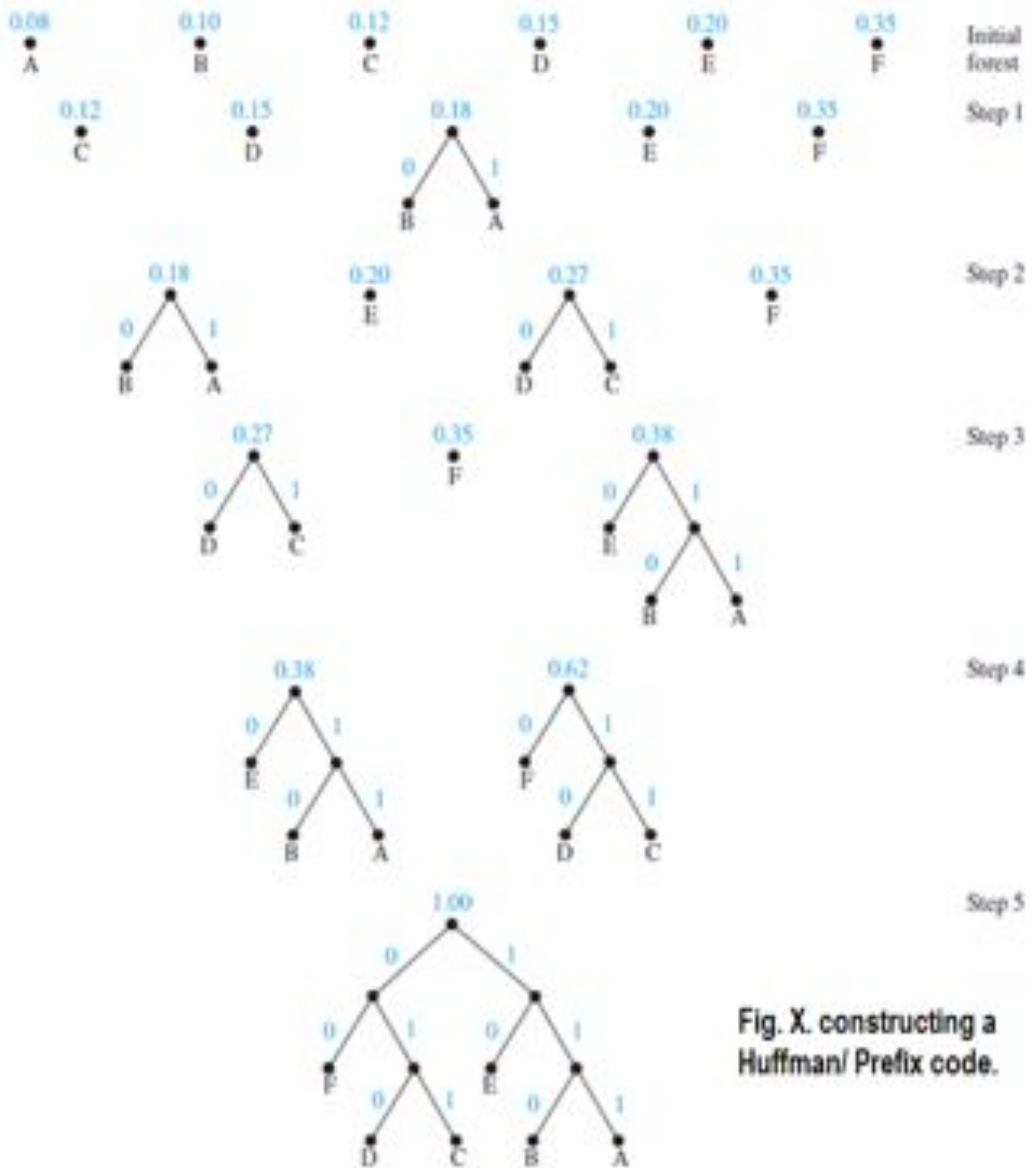


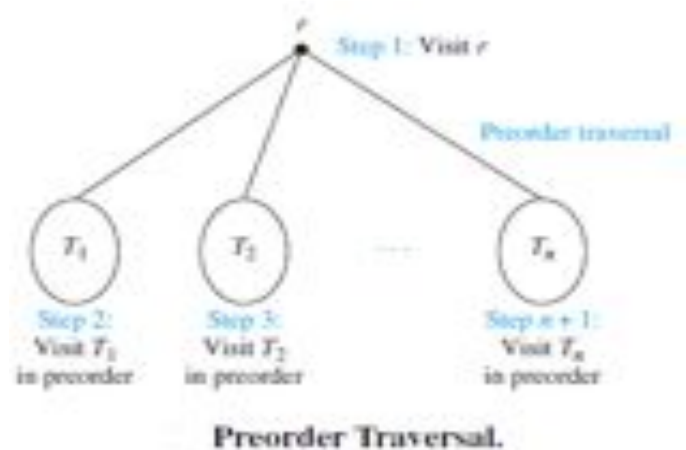
Fig. X. constructing a Huffman/ Prefix code.

TREE TRAVERSAL

Ordered rooted trees are often used to store information. Therefore, we need procedures for systematically **visiting each vertex** of an ordered rooted tree to access data. These procedures are called **traversal algorithms**. Three common recursive algorithms for visiting all the vertices of a tree are:

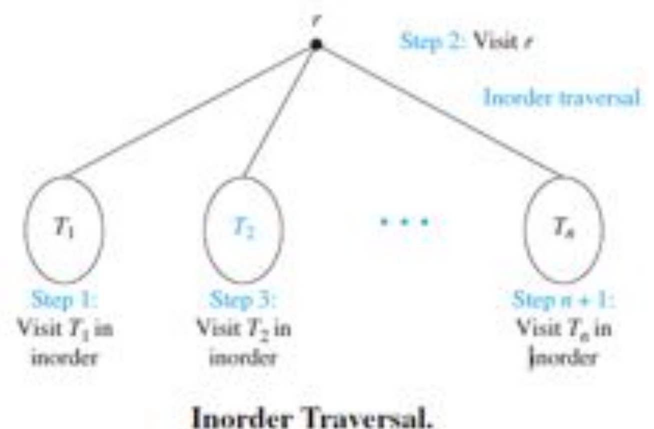
1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

1. **Preorder traversal**: 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 $at\ \boldsymbol{r}$ from left to right in T . The preorder traversal begins by visiting r. It continues by traversing T_1 in preorder, then T_2 in preorder, and so on, until T_n is traversed in preorder. For example, the preorder traversal of tree T given below is shown in figure Y:



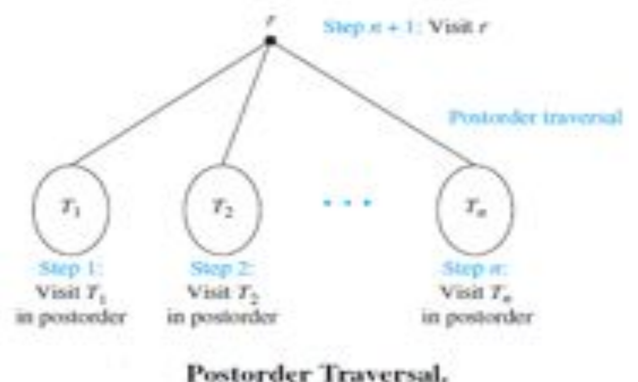
Preorder Traversal.

2. **Inorder traversal**: 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 at r from left to right. The inorder traversal begins by traversing T_1 in inorder, then visiting r. It continues by traversing T_2 in inorder, then T_3 in inorder, . . . , and finally T_n in inorder. For example, the preorder traversal of tree T given below is shown in figure Z.

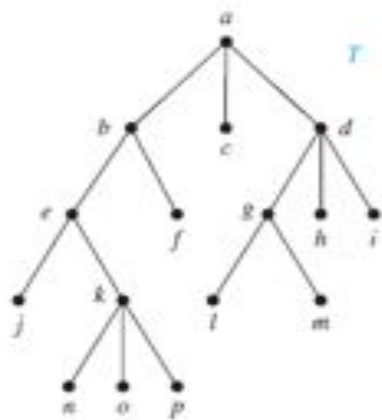


Inorder Traversal.

3. Postorder traversal: 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 at r from left to right. The postorder traversal begins by traversing T_1 in postorder, then T_2 in postorder, . . . , then T_n in postorder, and ends by visiting r. For example, the preorder traversal of given tree T is shown in figure V.

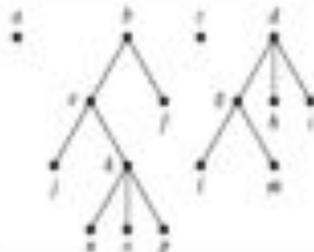


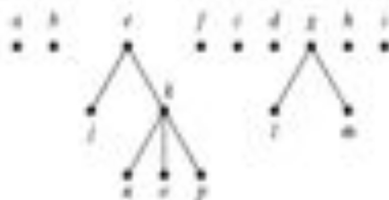
Postorder Traversal.



ordered rooted tree T

Preorder traversal: Visit root, visit subtrees left to right





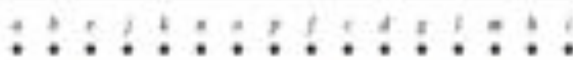
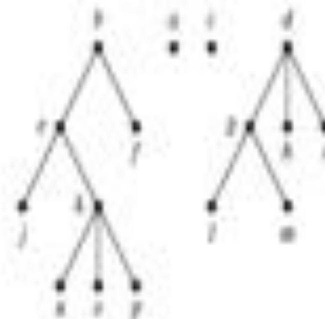
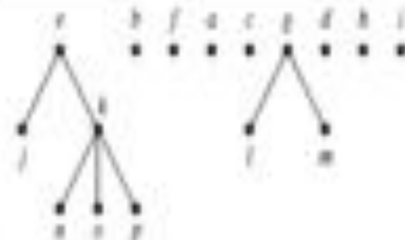


Figure Y. preorder traversal of tree T.

Inorder traversal: Visit leftmost subtree, visit root, visit other subtrees left to right





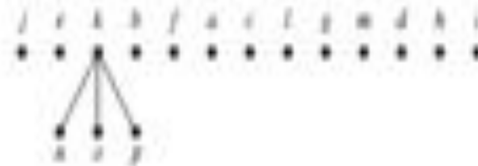


Fig Z. The Inorder Traversal of T.

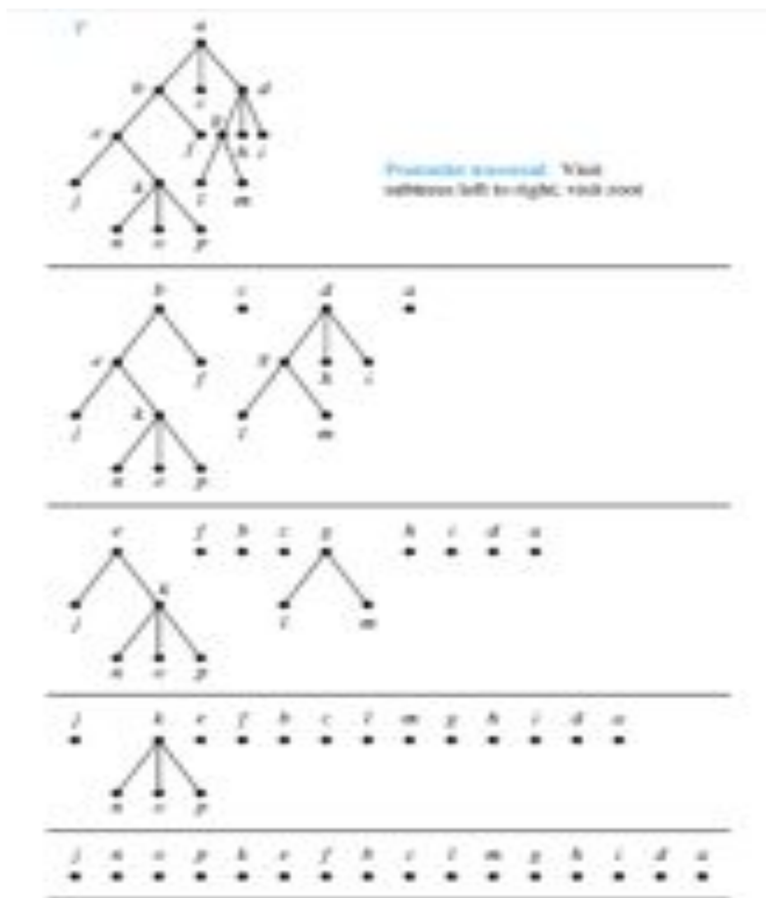


Fig. V. The Postorder Traversal of T.

Note: We can uniquely draw a binary tree when its preorder and inorder traversal are given or when its inorder and postorder traversal are given.

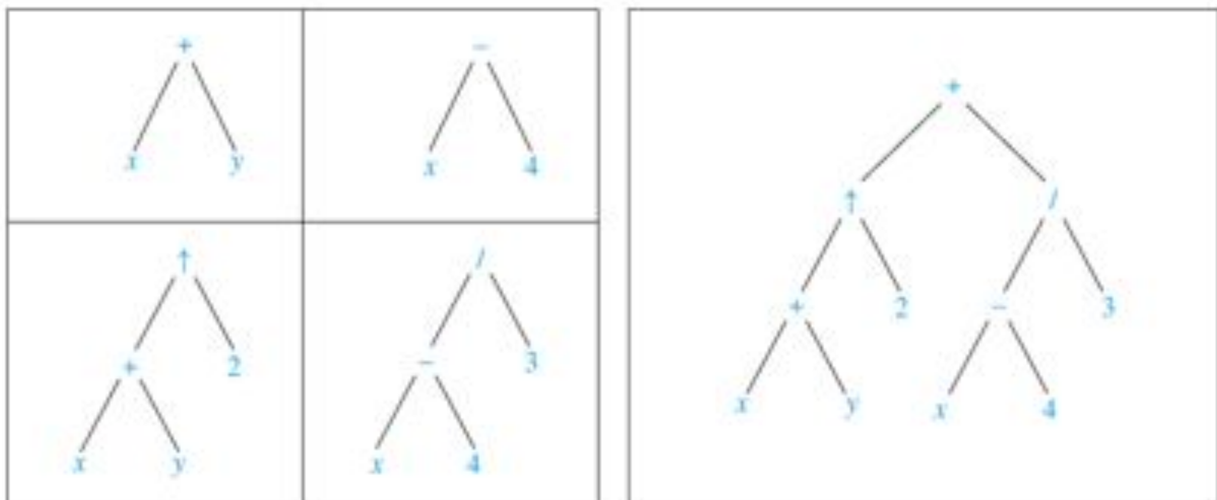
REPRESENTING EXPRESSIONS USING TREES

We can represent complicated expressions, such as compound propositions, combinations of sets, and arithmetic expressions using ordered rooted trees, where the internal vertices represent **operations**, and the leaves represent the variables or numbers. Each operation operates on its left and right subtrees (in that order).

Operators: + (addition), – (subtraction), $*$ (multiplication), / (division), and $\uparrow$ (exponentiation).

Example 1: Represent the expression $((x + y) \uparrow 2) + ((x - 4)/3)$ using a tree.

Solution: We build the tree representing given expression in bottom up fashion as follows:



A Binary Tree Representing $((x + y) \uparrow 2) + ((x - 4)/3)$.

Example 2: Figure X given below shows the trees representing $(\boldsymbol{x} + \boldsymbol{y})/(\boldsymbol{x} + \mathbf{3})$, $(\boldsymbol{x} + (\boldsymbol{y}/\boldsymbol{x})) + \mathbf{3}$, and $\boldsymbol{x} + (\boldsymbol{y}/(\boldsymbol{x} + \mathbf{3}))$.

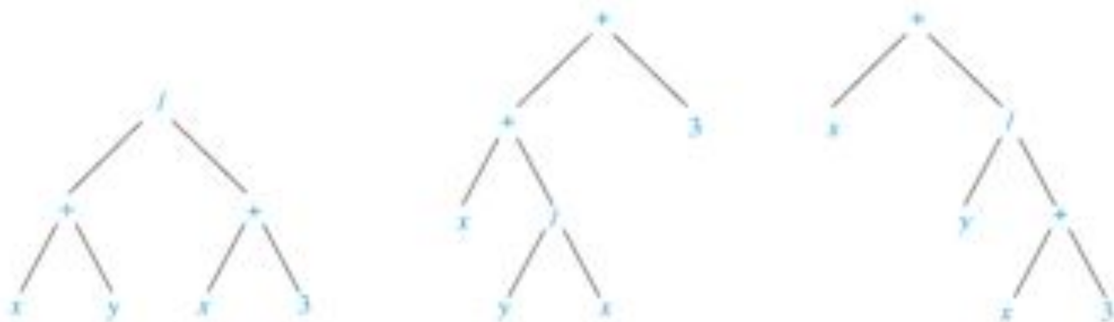


Figure X. **Rooted Trees Representing $(x + y)/(x + 3)$, $(x + (y/x)) + 3$, and $x + (y/(x + 3))$.**

An inorder traversal of the binary tree representing an expression produces the original expression with the elements and operations in the same order as they originally occurred, except for **unary operations**, which instead immediately follow their operands. For instance, inorder traversals of the above binary tree, which represent the expressions (x + y)/(x + 3), (x + (y/x)) + 3, all lead to the infix expression x + y/x + 3. To make such expressions unambiguous it is necessary to include parentheses in the inorder traversal whenever we encounter an operation. The fully parenthesized expression obtained in this way is said to be in **infix form.**

We obtain the **prefix form** (or Polish notation, named after the Polish logician Jan Lukasiewicz.) of an expression when we traverse its rooted tree in preorder.

We obtain the **postfix form** (or **reverse Polish notation**) of an expression by traversing its binary tree in postorder.

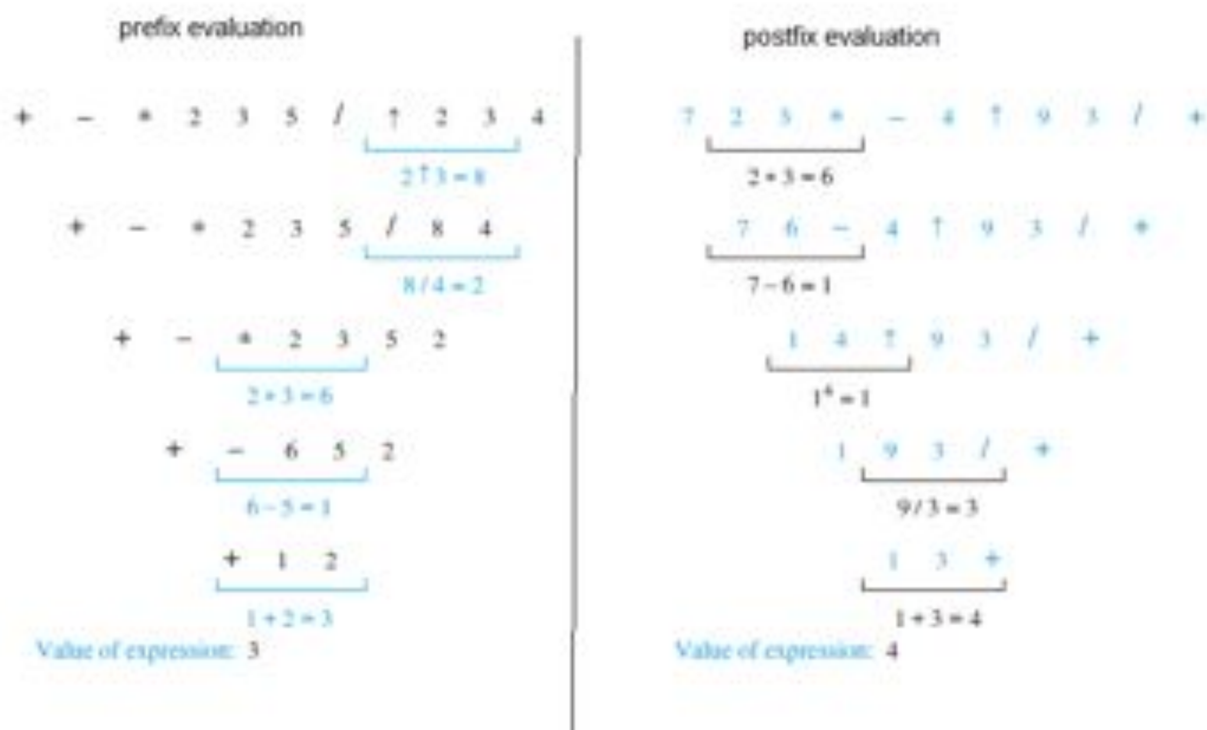
Note: Both prefix and postfix forms are unambiguous, and therefore do not require parenthesis. They do not require precedence and associativity rules and thus, are easy to parse. Furthermore, they can be evaluated easily without scanning back and forth. So, they are used extensively in computer science. Such expressions are especially useful in the construction of compilers.

EVALUATION OF EXPRESSIONS

Evaluating a prefix expression: In the prefix form, a binary operator, such as +, precedes its two operands. Hence, we can evaluate an expression in prefix form by working from **right to left**. When we encounter an operator, we perform the corresponding operation with the two operands immediately to the right of this operand. Also, whenever an operation is performed, we consider the result a new operand.

Evaluating a postfix expression:In the postfix form of an expression, a binary operator follows its two operands. So, work from **left to right**, carrying out operations whenever an operator follows two operands. After an operation is carried out, the result of this operation becomes a new operand.

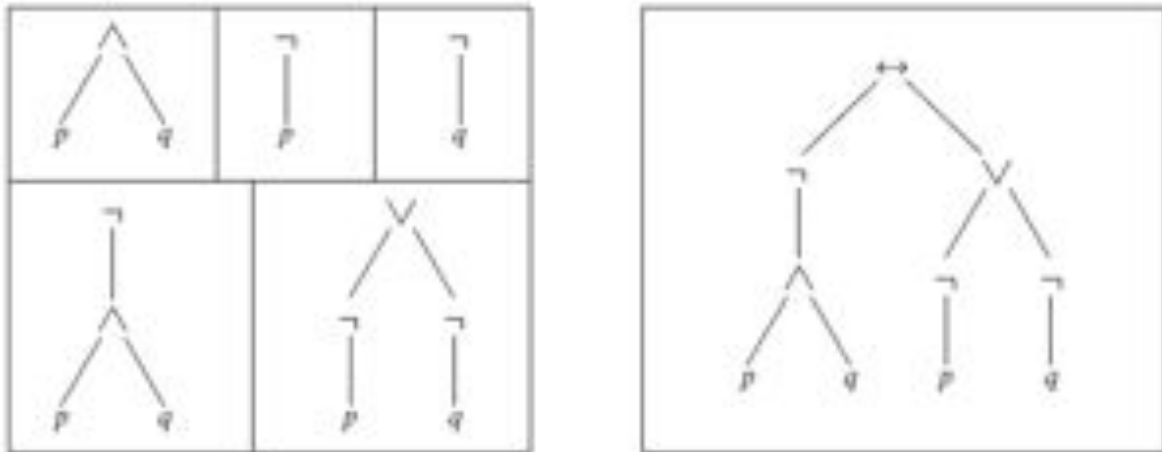
The figure given below illustrates both the methods:



Note: in order to find the prefix or postfix form for a given expression, represent the expression using a tree, then traverse the tree as per requirement.

Example: 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.

Solution: construct the rooted tree as shown:



Now traversing the tree in preorder yields the prefix expression: $\leftrightarrow \neg \wedge pq \vee \neg p \neg q$

And traversing the tree in postorder yields the postfix expression: $pq \wedge \neg p \neg q \neg \vee \leftrightarrow$

And traversing the tree in inorder (including parentheses), yields the infix expression: $(\neg(p \wedge q)) \leftrightarrow ((\neg p) \vee (\neg q))$

SPANNING TREES

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.**

A simple graph with a spanning tree **must be connected**, because there is a path in the spanning tree between any two vertices. The converse is also true; that is, every connected simple graph has a spanning tree.

Example: Find a spanning tree of the simple graph G shown in Figure X.

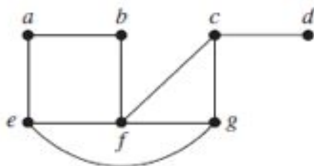


fig X.

Solution: The graph G is connected, but it is not a tree because it contains simple circuits. Remove the edge $\{a, e\}$. This eliminates one simple circuit, and the resulting subgraph is still connected and still contains every vertex of G. Next remove the edge $\{e, f\}$ to eliminate a second simple circuit. Finally, remove edge $\{c, g\}$ to produce a simple graph with no simple circuits. This subgraph is a spanning tree, because it is a tree that contains every vertex of G. The sequence of edge removals

used to produce the spanning tree is illustrated in Figure Y.

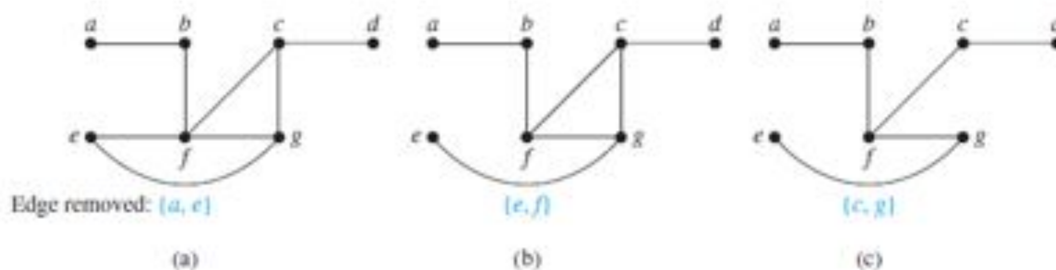


Fig Y. Producing a Spanning Tree for G by Removing Edges That Form Simple Circuits.

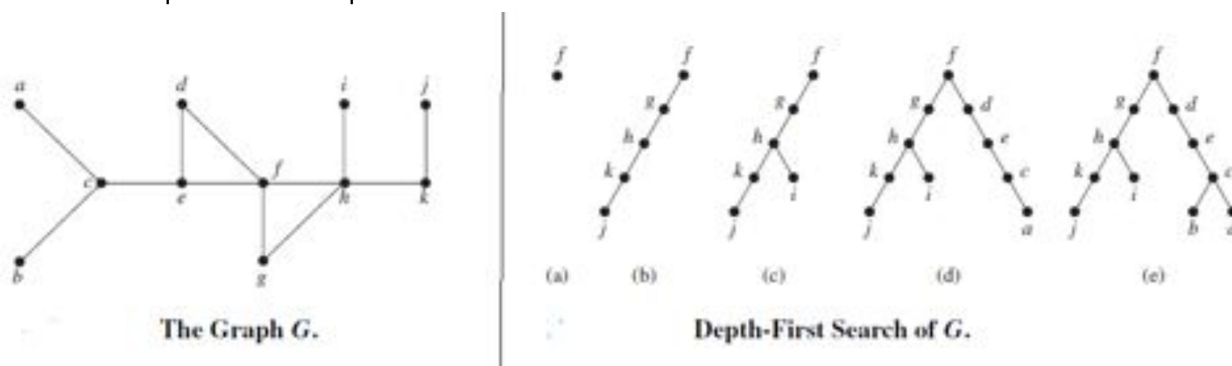
Algorithms for producing spanning trees:

Finding a spanning tree by removing edges (as was done in above example) is inefficient, because it requires that simple circuits be identified. Instead of constructing spanning trees by removing edges, spanning trees can be built up by successively adding edges. Two algorithms based on this principle are:

1. **DFS** (depth-first search)

<u>Procedure</u>: Arbitrarily choose a vertex of the graph as the root. Form a path starting at this vertex by successively adding vertices and edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding vertices and edges to this path as long as possible. If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree. However, if the path does not go through all vertices, more vertices and edges must be added. Move back to the next to last vertex in the path, and, if possible, form a new path starting at this vertex passing through vertices that were not already visited. If this cannot be done, move back another vertex in the path, that is, two vertices back in the path, and try again.

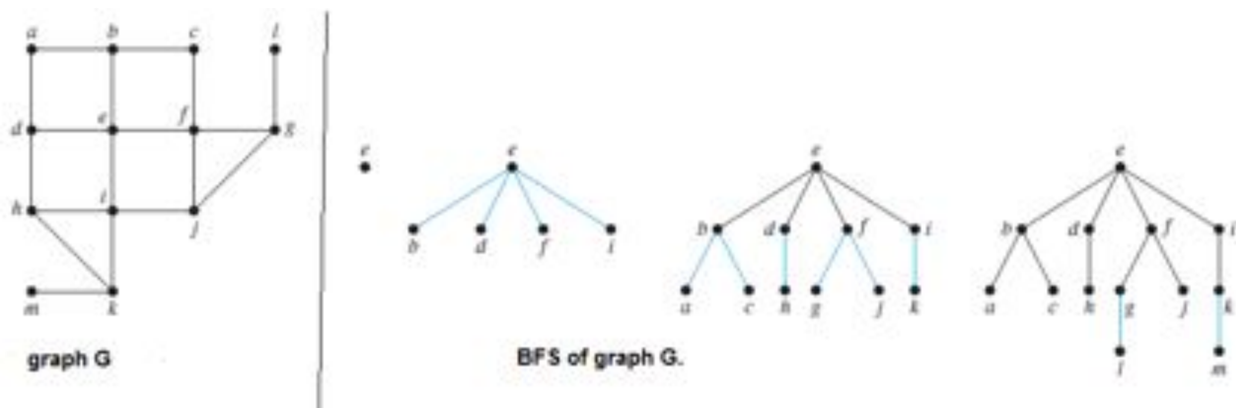
Repeat this procedure, beginning at the last vertex visited, moving back up the path one vertex at a time, forming new paths that are as long as possible until no more edges can be added. Because the graph has a finite number of edges and is connected, this process ends with the production of a spanning tree. Each vertex that ends a path at a stage of the algorithm will be a leaf in the rooted tree, and each vertex where a path is constructed starting at this vertex will be an internal vertex. Depth-first search is also called backtracking, because the algorithm returns to vertices previously visited to add paths. An example of the DFS follows.



The Graph G. Depth-First Search of G.

2. BFS (breadth-first search)

Procedure: Arbitrarily choose a root from the vertices of the graph. Then add all edges incident to this vertex. The new vertices added at this stage become the vertices at level 1 in the spanning tree. Arbitrarily order them. Next, for each vertex at level 1, visited in order, add each edge incident to this vertex to the tree as long as it does not produce a simple circuit. Arbitrarily order the children of each vertex at level 1. This produces the vertices at level 2 in the tree. Follow the same procedure until all the vertices in the tree have been added. The procedure ends because there are only a finite number of edges in the graph. A spanning tree is produced because we have produced a tree containing every vertex of the graph. An example of breadth-first search follows.



DFS AND BFS for digrected graphs:

Both depth-first search and breadth-first search can be modified so that they run on a directed graph. However, the output will not necessarily be a spanning tree, but rather a **spanning forest**.

Q. How many edges must be removed from a connected graph with n vertices and m edges to produce a spanning tree?

Solution: note that the number of vertices in resulting tree $= n$. And number of edges in resulting tree $= n-1$. Thus number of edges that must be removed $= m - (n-1) = m-n+1$.

Q. How many edges must be removed to produce a spanning forest of a graph with n vertices , m edges and c connected components?

Solution: $m-n+c$.

MINIMUM SPANNING TREE

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

There may be more than one minimum spanning tree for a given connected weighted simple graph.

ALGORITHMS FOR CONSTRUCTING MINIMUM SPANNING TREES

1. Prim’s algorithm: (Given by Robert Prim, 1957).
    a. Choose any edge with smallest **weight** and put it into the spanning tree T.
    b. Successively add to the tree T edges of minimum weight that are **incident** to a vertex already in the tree, never forming a simple circuit with those edges already in the tree.
    c. Stop when $n - 1$ edges have been added.

2. Kruskal’s algorithm : (Given by Joseph Kruskal, 956).
    a. Choose an edge in the graph with minimum weight.
    b. Successively add **any edge (**not necessarily incident to a vertex already in the tree, unlike Kruskal’s algorithm**)** with minimum weight that do not form a simple circuit with those edges already chosen.
    c. Stop after n – 1 edges have been selected.

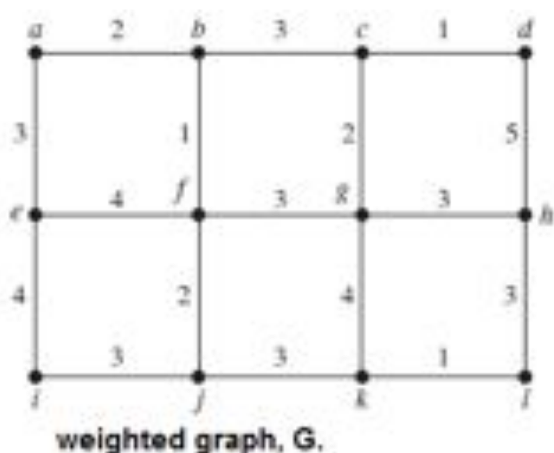
Note: When there is more than one edge with the same weight, arbitrarily choose one of them,(except when ordering of edges is given).

Comparison:

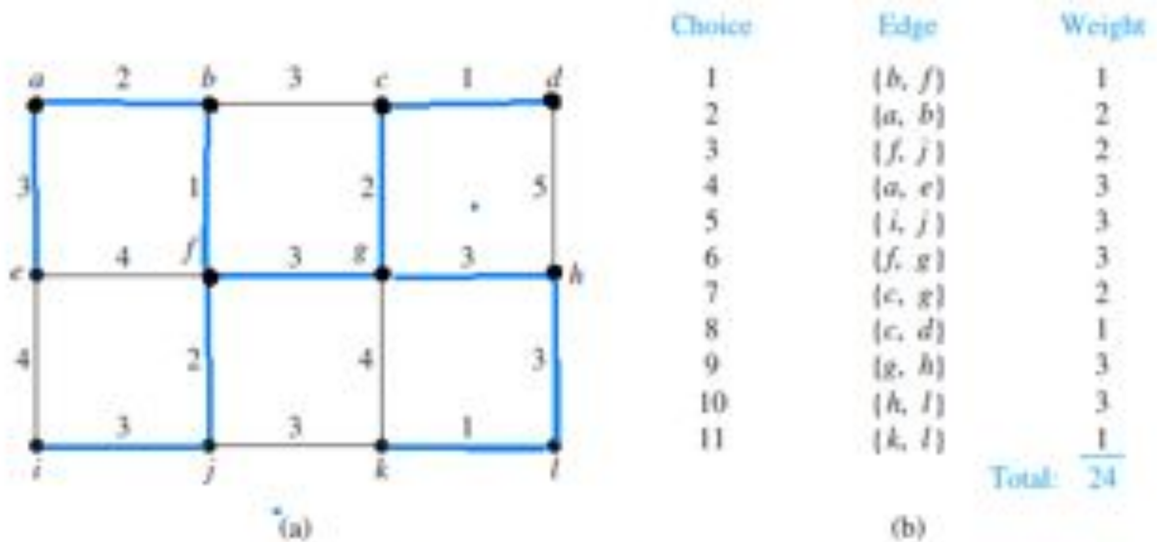
Both Prim’s and Kruskal’s algorithms are greedy algorithms (greedy algorithm is a procedure that makes an optimal choice at each of its steps i.e local optimal choice). The overall solution produced by greedy algorithm may not be optimal. However, both Prim’s and Kruskal’s algorithms produce optimal solution.

The time complexity of Prim’s algorithm is $O(e \log v)$ and that of Kruskal’s algorithm is $O(e \log e)$. Thus, Kruskal’s algorithm is preferred when the graph is sparse (contains relatively few edges).

Example: The workings of Prim’s and Kruskal’s algorithm for the weighted graph G are shown as:



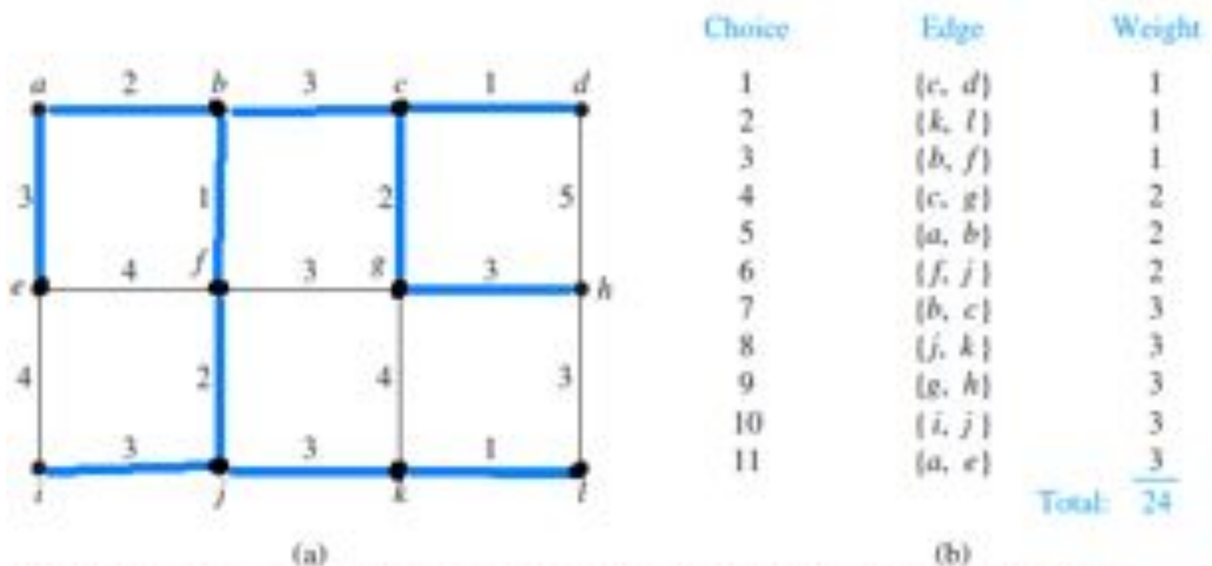
weighted graph, G.



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

(a) (b)

Fig X. (a) A minimum spanning tree produced using Prim's algo.The edges in color are the one's in required tree.(b) the choice of edge made at each step.



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

(a) (b)

Fig X. (a) a minimum spanning tree produced using Kruskal's algo. (b) the choice of edge made at each step.