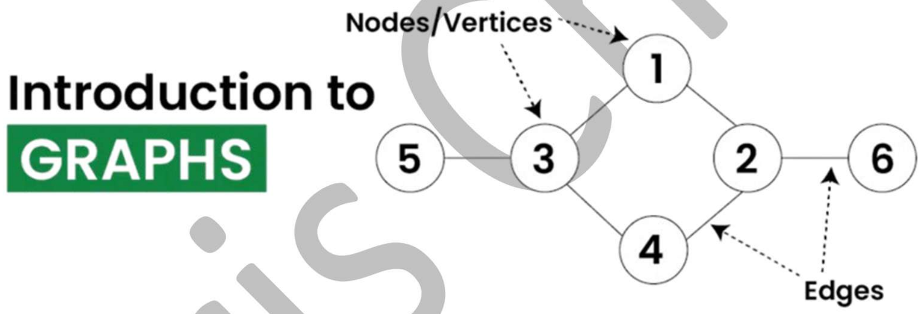
Unit – III
Graphs

1. Basic Concepts

A **graph** is a non-linear data structure consisting of **nodes (vertices)** and **edges (connections)** between them. It is used to represent relationships between objects, such as social networks, road maps, or dependency graphs. In the field of sports data science, graph data structure can be used to analyze and understand the dynamics of team performance and player interactions on the field.



Components of a Graph

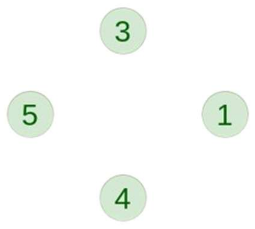
1. **Vertices (Nodes)**: The fundamental units of a graph, represented as **V**.
2. **Edges**: The connections between nodes, represented as **E**.

A graph is represented as **G(V, E)** where:

- **V** = Set of vertices
- **E** = Set of edges

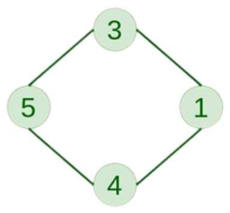
Types of Graphs

1. Null Graph: A graph is known as a null graph if there are no edges in the graph.

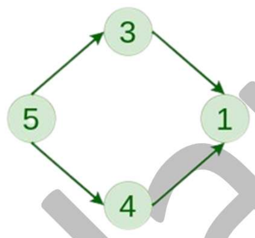


2. Trivial Graph: Graph having only a single vertex, it is also the smallest graph possible.

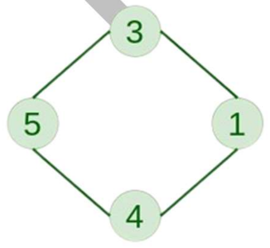
3. Undirected Graph: A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.



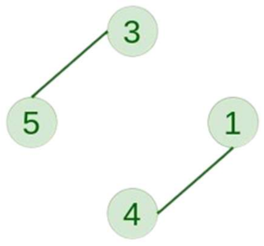
4. Directed Graph: A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.



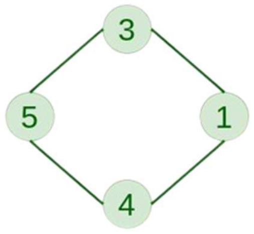
5. Connected Graph: The graph in which from one node we can visit any other node in the graph is known as a connected graph.



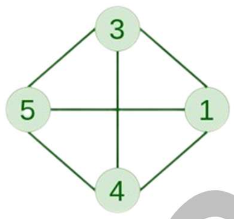
6. Disconnected Graph: The graph in which at least one node is not reachable from a node is known as a disconnected graph.



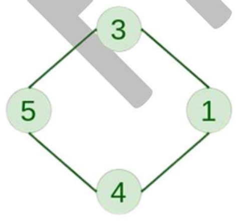
7. Regular Graph: The graph in which the degree of every vertex is equal to K is called K regular graph.



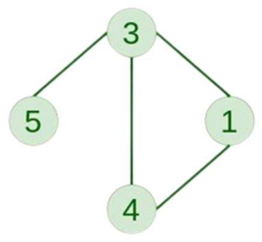
8. Complete Graph: The graph in which from each node there is an edge to each other node.



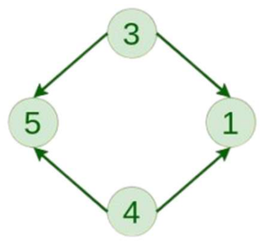
9. Cycle Graph: The graph in which the graph is a cycle in itself, the minimum value of degree of each vertex is 2.



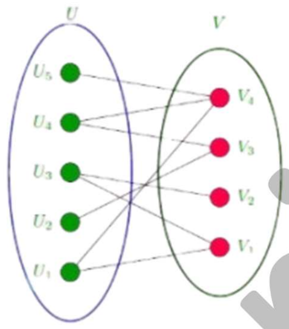
10. Cyclic Graph: A graph containing at least one cycle is known as a Cyclic graph.



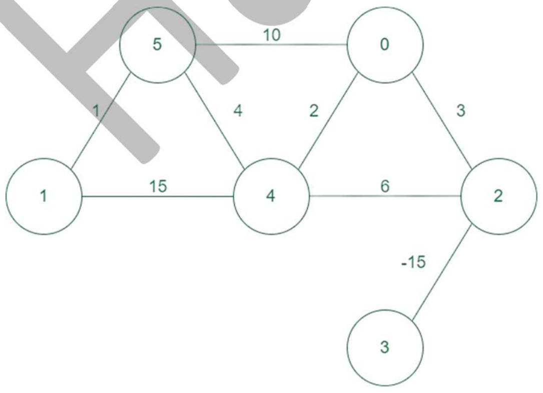
11. Directed Acyclic Graph: A Directed Graph that does not contain any cycle.



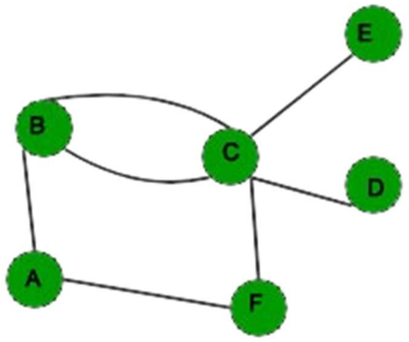
12. Bipartite Graph: A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.



13. Weighted Graph: A graph in which the edges are already specified with suitable weight is known as a weighted graph.

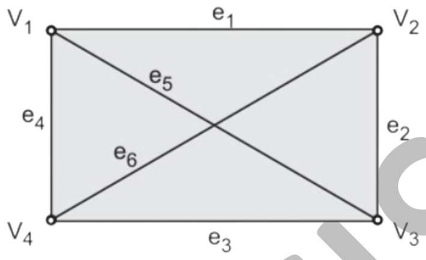


14. Multigraph: A graph in which multiple edges may connect the same pair of vertices is called a multigraph.

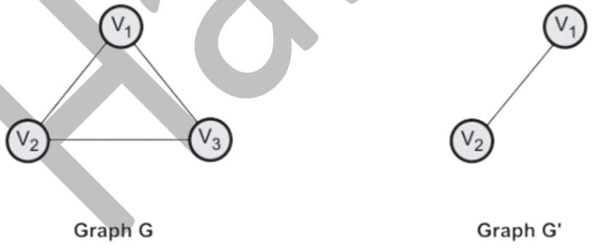


Properties of Graphs

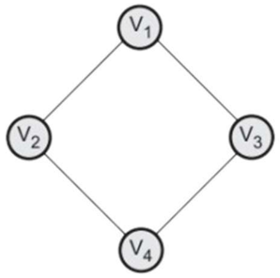
1. Complete Graph: If an undirected graph of n vertices consists of $\frac{n(n-1)}{2}$ number of edges, then that graph is called a complete graph.



2. Subgraph: A subgraph G' of graph G is a graph such that the set of vertices and set of edges of G' are proper subset of the set of edges of G.



3. Connected Graph: An undirected graph is said to be connected if for every pair of distinct vertices V_i and V_j in V(G) there is an edge V_i to V_j in G.

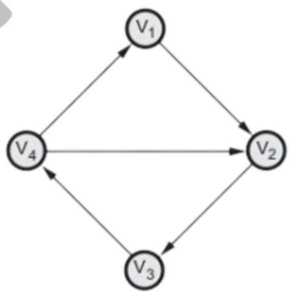


For example:

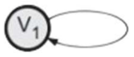
- V1 – V2
- V1 – V2 – V4
- V1 – V3

4. Degree, in-degree and out-degree: The degree of vertex is the number of edges associated with the vertex. In-degree of a vertex is the number of edges that are incident on that vertex. Out-degree of the vertex is total number of edges that are going away from the vertex.

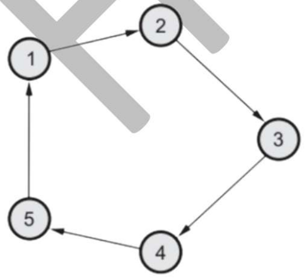
Vertices	Degree	In-degree	Out-degree
V_1	2	1	1
V_2	3	2	1
V_3	2	1	1
V_4	3	1	2



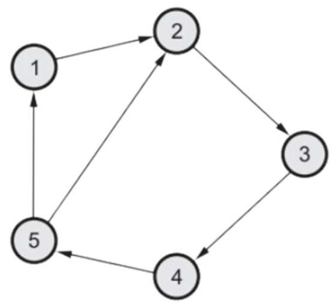
5. Self-loop: An edge that connects the same vertex, to itself.



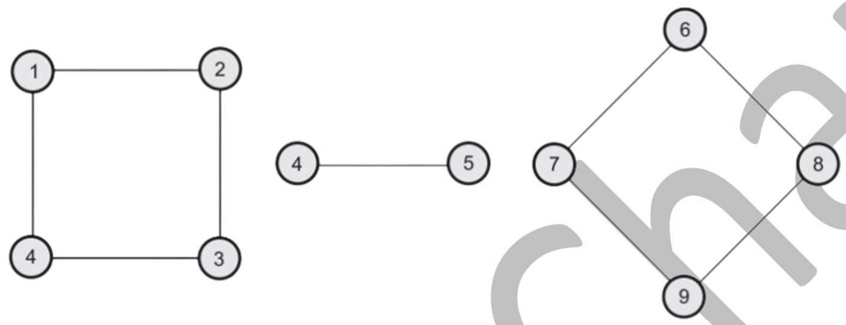
6. Path: A path is denoted using sequence of vertices and there exists an edge from one vertex to the next vertex. Path of the following graph is 1-2-3-4-5



7. Cycle: A closed walk through the graph with repeated vertices, mostly having the same starting and ending vertex is called a cycle.



8. Component: The maximal connected subgraph of a graph is called component of a graph. Following are 3 components of a graph:



Comparison between Graph and Tree

Feature	**Graph**	**Tree**
Definition	A collection of nodes (vertices) connected by edges.	A special type of graph with a hierarchical structure.
Structure	Can be cyclic or acyclic.	Always acyclic (no loops/cycles).
Hierarchy	No strict parent-child relationship.	Has a strict parent-child relationship.
Connectivity	Can be connected or disconnected.	Always connected (except for forest).
Edges	Can have multiple edges and loops.	Only one edge between any two nodes (no loops).
Root Node	No root node.	Has a root node at the top.
Traversal	DFS, BFS.	DFS (Preorder, Inorder, Postorder), BFS (Level Order).
Usage	Networks, social media, shortest path problems, etc.	Hierarchical data, file systems, databases, etc.

Applications of Graphs

1. **Google Maps & GPS Navigation** → Graphs help find the shortest route between locations using algorithms like Dijkstra's.
2. **Social Networks (Facebook, LinkedIn, Twitter)** → Users are nodes, and connections are edges, helping in friend suggestions and network analysis.
3. **Computer Networks & Internet Routing** → Graphs model network topology for **routing protocols** like OSPF & BGP.
4. **AI & Game Development (Pathfinding)** → Games use *A* and BFS/DFS* for character movement and decision-making.
5. **Task Scheduling & Dependency Management** → Graphs (DAGs) help schedule tasks in **project management & CPU scheduling**.

1.1 Storage Representation

There are multiple ways to store a graph: The following are the most common representations.

- Adjacency Matrix
- Adjacency List

1.2 Adjacency Matrix

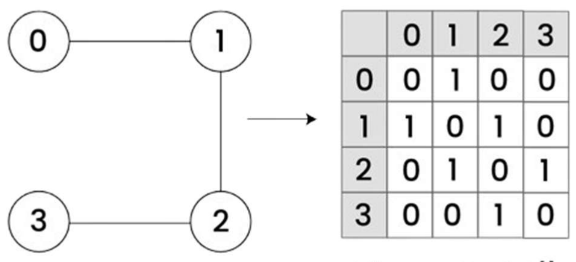
Adjacency Matrix is a square matrix used to represent a finite graph by storing the relationships between the nodes in their respective cells. For a graph with ***V*** vertices, the adjacency matrix *A* is a *V* × *V* matrix or 2D array.

Properties of Adjacency Matrix

- **Diagonal Entries**: The diagonal entries A[i][j] are usually set to 0 (in case of unweighted) and INF in case of weighted, assuming the graph has no self-loops.
- **Undirected Graphs**: For undirected graphs, the adjacency matrix is symmetric. This means A[i][j] = A[j][i] for all i and j.

1. Adjacency Matrix for Undirected and Unweighted graph:

- **A[i][j] = 1,** there is an edge between vertex i and vertex j.
- **A[i][j] = 0,** there is NO edge between vertex i and vertex j.

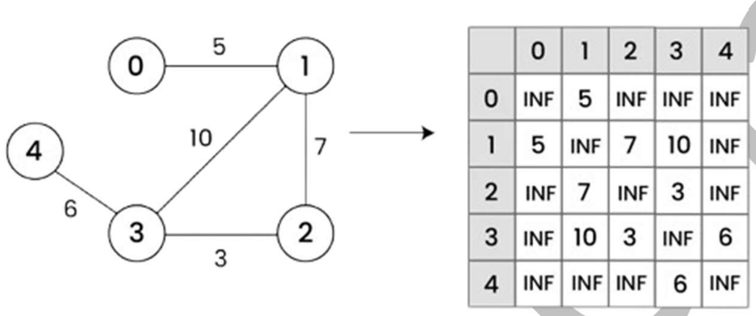


	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0

Adjacency Matrix A[ ]

2. Adjacency Matrix for Undirected and Weighted graph:

- **A[i][j] = INF**, when there is no edge between vertex i and j
- **A[i][j] = w,** when there is an edge between vertex i and j having weight = w.

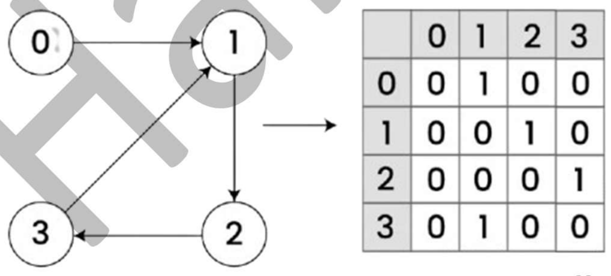


	0	1	2	3	4
0	INF	5	INF	INF	INF
1	5	INF	7	10	INF
2	INF	7	INF	3	INF
3	INF	10	3	INF	6
4	INF	INF	INF	6	INF

Adjacency Matrix A[ ]

3. Adjacency Matrix for Directed and Unweighted graph:

- **A[i][j] = 1,** there is an edge from vertex i to vertex j
- **A[i][j] = 0,** No edge from vertex i to j.

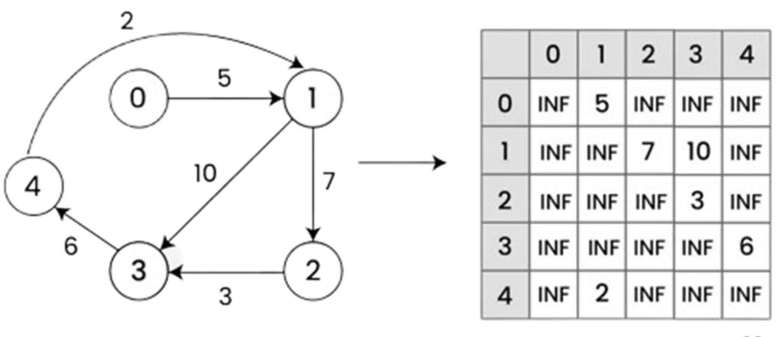


	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0

Adjacency Matrix A[ ]

4. Adjacency Matrix for Directed and Weighted graph:

- **A[i][j] = INF**, then there is no edge from vertex i to j
- **A[i][j] = w,** then there is an edge from vertex i having weight w



	0	1	2	3	4
0	INF	5	INF	INF	INF
1	INF	INF	7	10	INF
2	INF	INF	INF	3	INF
3	INF	INF	INF	INF	6
4	INF	2	INF	INF	INF

Adjacency Matrix A[ ]

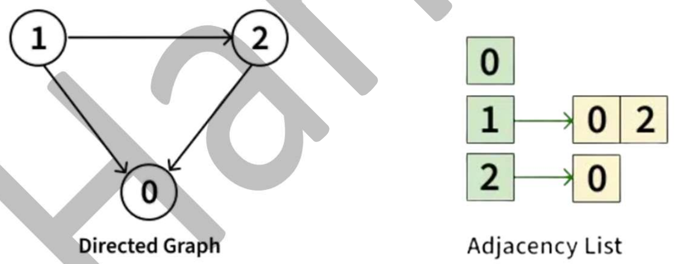
1.3 Adjacency List

An adjacency list is a data structure used to represent a graph where each node in the graph stores a list of its neighboring vertices.

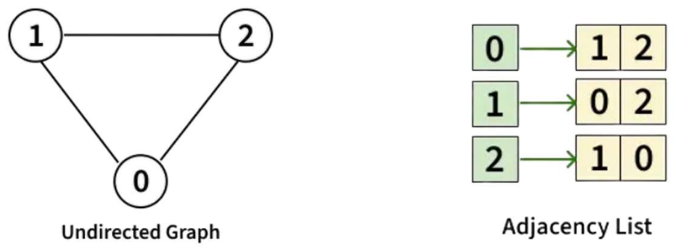
Characteristics of the Adjacency List:

- An adjacency list representation uses a list of lists. We store all adjacent of every node together.
- The size of the list is determined by the number of vertices in the graph.
- All adjacent of a vertex are easily available. To find all adjacent, we need only *O(n)* time where is the number of adjacent vertices.

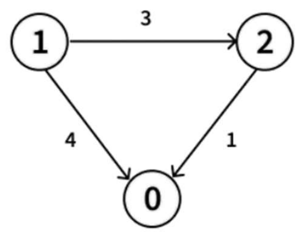
1. Adjacency List for Directed and Unweighted graph:



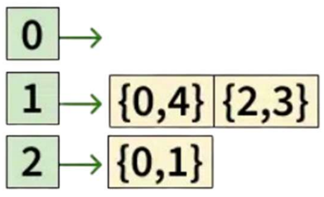
2. Adjacency List for Undirected and Unweighted graph:



3. Adjacency List for Directed and Weighted graph:

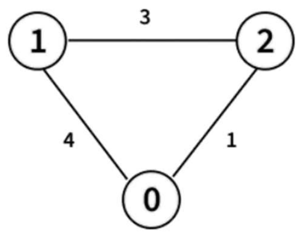


Directed Weighted Graph

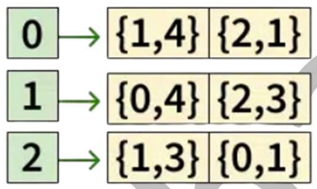


Adjacency List

4. Adjacency List for Undirected and Weighted graph:



Undirected Weighted Graph

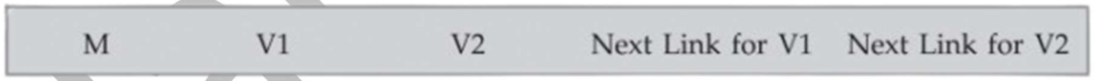


Adjacency List

1.4 Adjacency Multi List

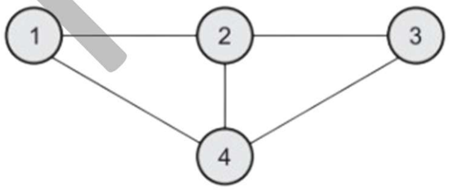
An **Adjacency Multi List** is a graph representation that is a mix of **adjacency list** and **edge list**, mainly used for **undirected graphs**. It efficiently stores edges while allowing easy traversal of both **vertices and edges**.

The node structure is as follows:

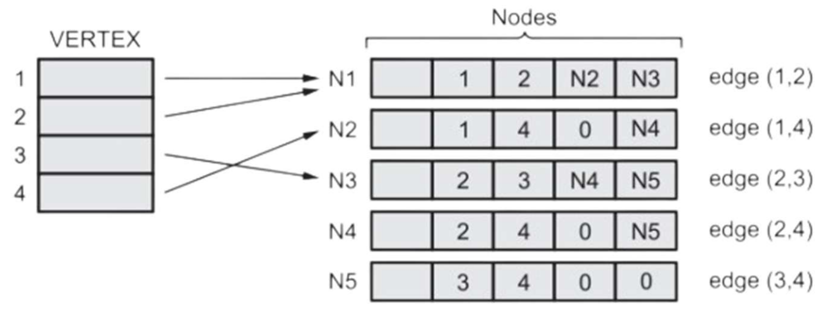


M	V1	V2	Next Link for V1	Next Link for V2

Here M is a one-bit field that is used to represent whether the edge is visited or not.



Here the edges are (1, 2), (1, 4), (2, 3), (2, 4) and (3, 4). These edges are represented as nodes. The vertices are 1, 2, 3, 4.



In node N1, the edge (1, 2) is specified. Third field N2 is for edge (1, 4), this link is the adjacent edge of edge (1, 2) considering vertex V1, i.e., 1 forth-field-N3 is for edge (2, 3), this link is the adjacent edge of edge (1, 2) considering vertex V2 i.e. 2.

In this way remaining nodes N2, N3, N4 and N5 are created.

Now vertex 1 is connected to (1, 2) and (1, 4) i.e. N1, N2.

Similarly, vertex 2 is connected to (1, 2), (2, 3) and (2, 4) i.e. N2, N3 and N4

Vertex 1: N1 → N2

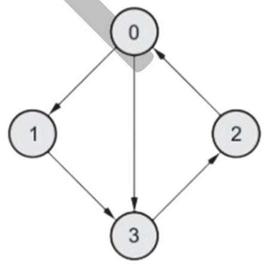
Vertex 2: N2 → N3 → N4

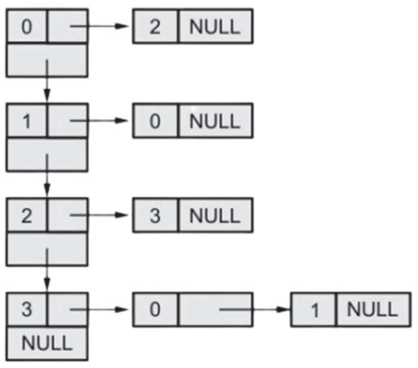
Vertex 3: N3 → N5

Vertex 4: N2 → N4 → N5

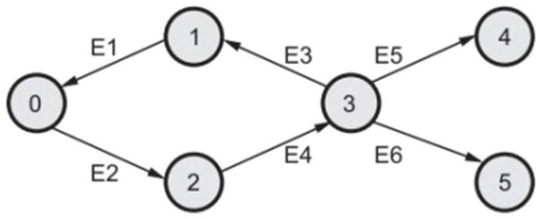
1.5 Inverse Adjacency List

An **Inverse Adjacency List** is a variation of the adjacency list used for **directed graphs**. Instead of storing outgoing edges (as in a standard adjacency list), it stores **incoming edges** for each vertex.





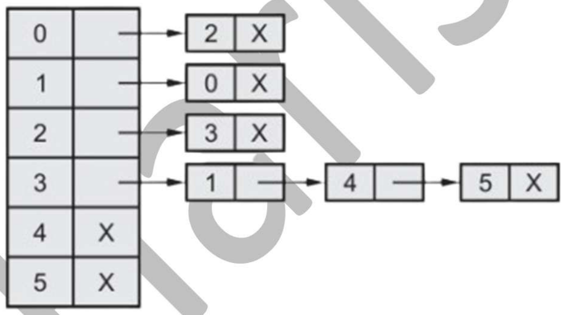
Q. Draw any directed graph with minimum 6 nodes and represent graph using adjacency matrix, adjacency list, adjacency multilist and inverse adjacency list.



i) Adjacency Matrix

$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 1 & 1 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

ii) Adjacency List



iii) Adjacency Multi List

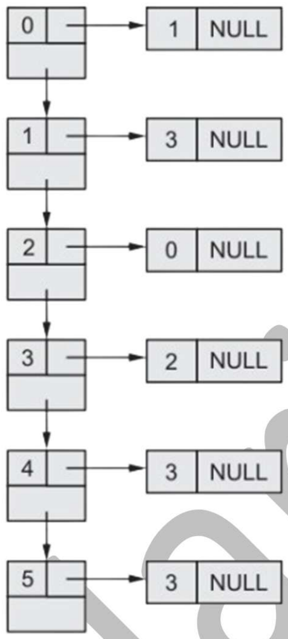
Edge 1		1	0	Edge 3	NULL	edge (1, 0)
Edge 2		0	2	NULL	NULL	edge (0, 2)
Edge 3		3	1	Edge 4	NULL	edge (3, 1)
Edge 4		2	3	NULL	NULL	edge (2, 3)

Edge 5		3	4	NULL	NULL	edge (3, 4)
Edge 6		3	5	NULL	NULL	edge (3, 5)

In above multilist various edges are represented. In the node of edge 1, the edge (1, 0) is represented. For vertex 1, the incident edge is E3. Hence, we enter 'Edge 3' in the fourth field. For vertex 0, the incident edge is E1 itself. So, we need not have to mention this self-defining edge. So, we enter NULL in fifth field.

Again, for Edge 2, for vertex 0 the incident edge is 1 – 0 (Edge 1) but this edge is already defined. Hence, we mention NULL in 4th field of Edge 2.

iv) Inverse Adjacency List



2. Traversals

2.1 Depth First

DFS (Depth-First Search) is a **graph traversal algorithm** that explores as far as possible along each branch before backtracking. It uses **recursion** (or a stack in an iterative approach) to visit nodes.

DFS for a Graph (Recursive):

Recursive DFS Algorithm

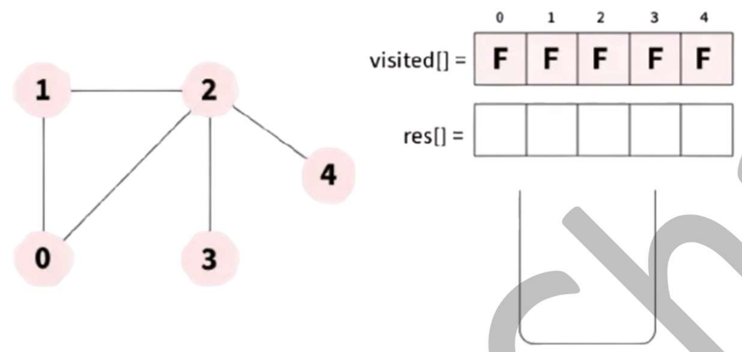
1. **Maintain a boolean visited array** to track visited nodes.
2. **Start DFS from a source node** and call `dfsRec()`.
3. **In `dfsRec()` function**:
   - Mark the node as visited.
   - Process the node (e.g., print/store).
   - Recursively call `dfsRec()` for all unvisited neighbors.
4. **Backtrack** when all neighbors are visited.

Recursive DFS C++ Function

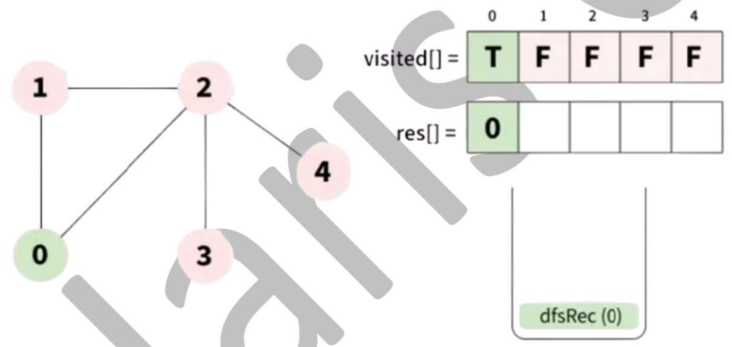
```
void DFSRec(vector<int> adj[], bool visited[], int node) {
    // Step 1: Mark node as visited
    visited[node] = true;

    // Step 2: Process node (print/store)
    cout << node << " ";

    // Step 3: Recursively visit all unvisited neighbors
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            DFSRec(adj, visited, neighbor);
        }
    }
}

// Function to handle multiple components
void DFS(vector<int> adj[], int V) {
    bool visited[V] = {false};  // Initialize visited array

    for (int i = 0; i < V; i++) { // Loop for disconnected graphs
        if (!visited[i]) {
            DFSRec(adj, visited, i);
        }
    }
}
```

Let us understand the working of **Depth First Search** with the help of the following **Illustration:** for the **source as 0**.

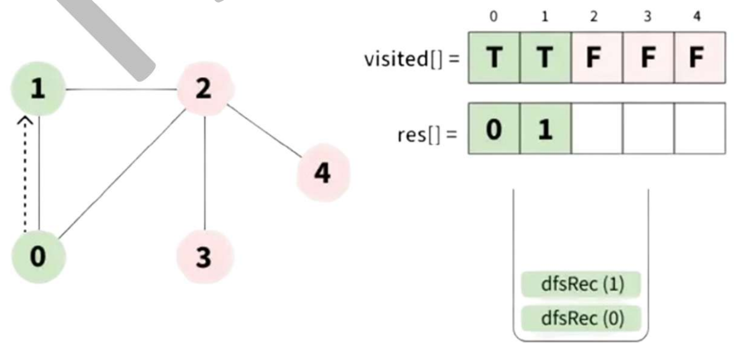
Step 1: Start with a `visited[]` array to mark which spots (nodes) we've seen.



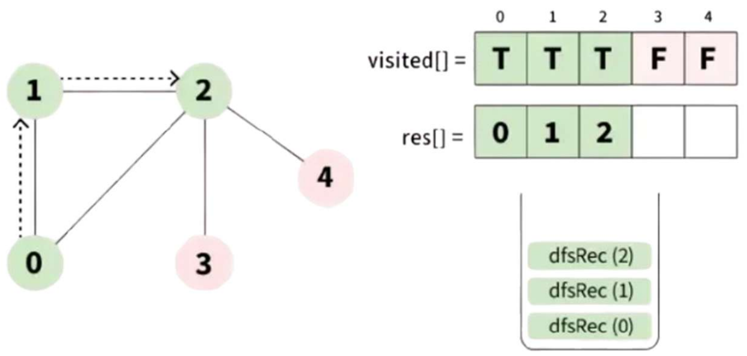
Step 2: Iteration 1: `DFSRec(adj, visited, 0)`, Marks `visited[0] = true`.



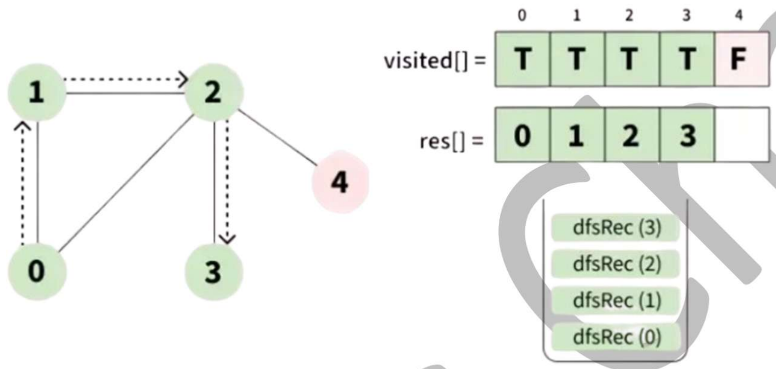
Step 3: Iteration 2: `DFSRec(adj, visited, 1)`, Marks `visited[1] = true`.



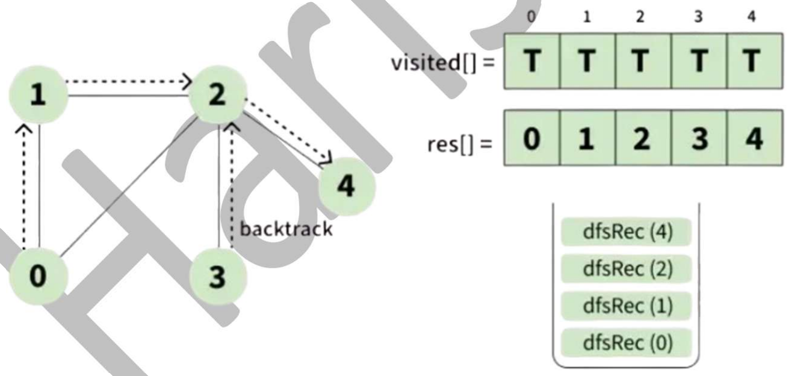
Step 4: Iteration 3: `DFSRec(adj, visited, 2)`, Marks `visited[2] = true`



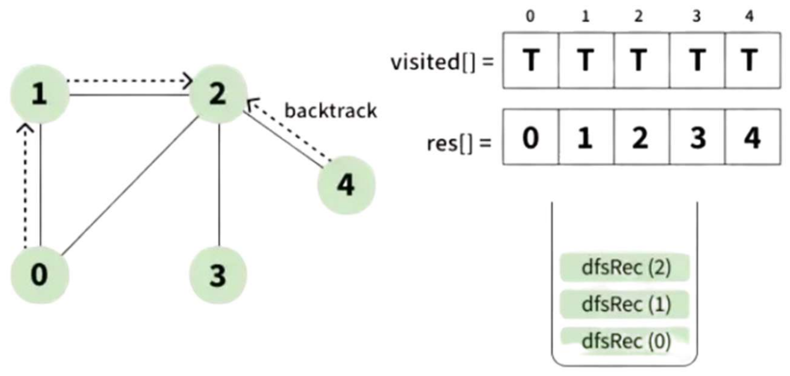
Step 5: Iteration 4: `DFSRec(adj, visited, 3)`, Marks `visited[3] = true`



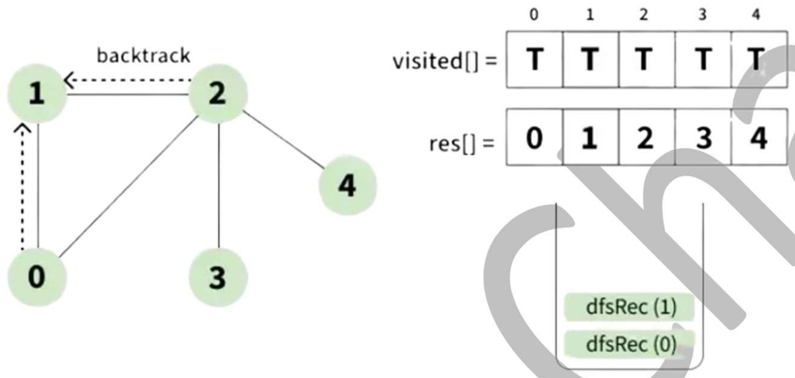
Step 6: Iteration 4: `DFSRec(adj, visited, 3)`, Marks `visited[3] = true`



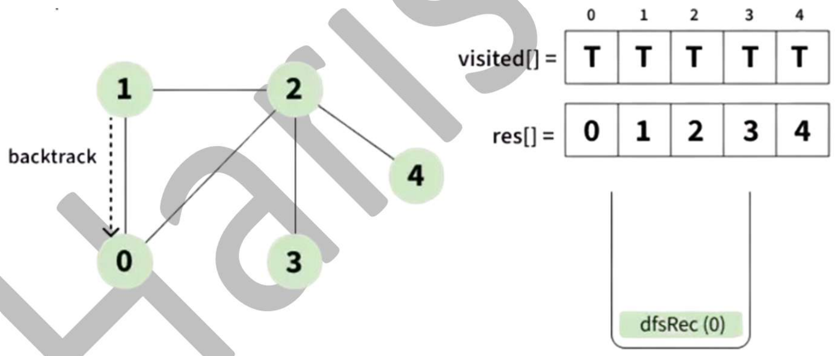
Step 7: Backtracking from 4



Step 8: Backtracking from 2

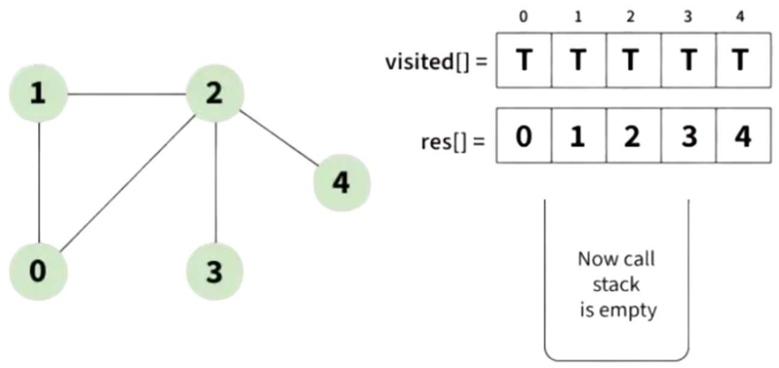


Step 9: Backtracking from 1



Step 10: DFS from source: 0

0 → 1 → 2 → 3 → 4



DFS for a Graph (Non-Recursive/Iterative):

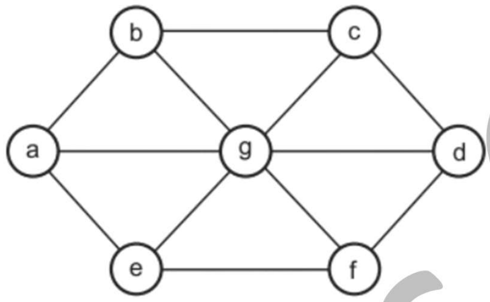
Non-Recursive DFS Algorithm

1. **Initialize a boolean visited array** to track visited nodes.
2. **Use a stack** to manage the traversal order.
3. **Push the starting node onto the stack** and mark it as visited.
4. **While the stack is not empty:**
    - Pop a node and process it.
    - Push all **unvisited** neighbors onto the stack and mark them as visited.
5. **Repeat** until all nodes are visited.

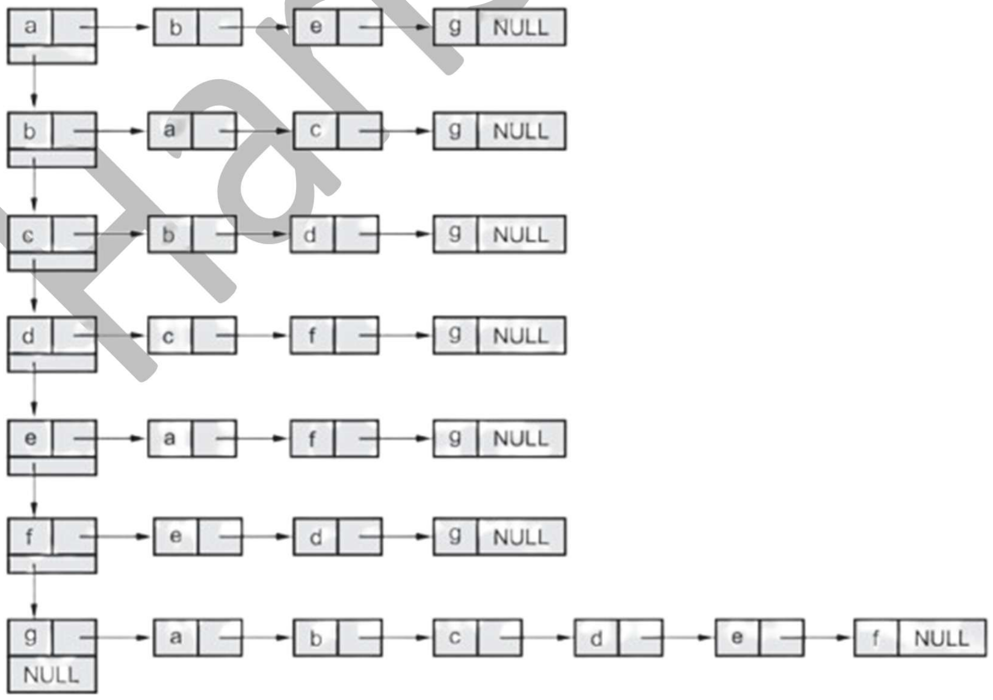
Non-Recursive DFS C++ Function

```
void DFSIterative(vector<int> adj[], int V, int start) {
    stack<int> s;                  // Stack for DFS
    bool visited[V] = {false};      // Visited array

    s.push(start);                  // Push starting node
    visited[start] = true;          // Mark it as visited

    while (!s.empty()) {
        int node = s.top();         // Get the top node
        s.pop();                    // Remove it from stack
        cout << node << " ";        // Process the node
        // Push unvisited neighbors onto the stack
        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                s.push(neighbor);
                visited[neighbor] = true;  // Mark as visited
            }
        }
    }
}
```

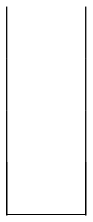
Example: For the graph given below, find DFS stepwise.



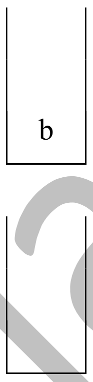
We first create adjacency list for given graph:



Step 1: Start with vertex **a**, mark it visited by writing 1 to index 'a' of visited array. Then print **a** as an output.

Stack	Visited Array	Output
	a 1 b c d e f g	a

Step 2: Find adjacent of a, and mark it visited. Push it onto the stack. Later b will be popped and printed.

Stack	Visited Array	Output
	a 1 b 1 c d e f g	a, b

Step 3: Find adjacent of b. The a is already visited, hence ignore. Next node will be c. Insert c onto the stack, mark it as visited. Push it. Later c will be popped and printed.

Stack	Visited Array		Output
c	a	1	a, b, c
	b	1	
	c	1	
	d		
	e		
	f		
	g		

Step 4: Find adjacent of c. Push d, mark it visited. Later d will be popped and printed.

Stack	Visited Array		Output
d	a	1	a, b, c, d
	b	1	
	c	1	
	d	1	
	e		
	f		
	g		

Step 5: Find adjacent of d. Push f, mark it as visited. Later f will be popped and printed

Stack	Visited Array		Output
f	a	1	a, b, c, d, f
	b	1	
	c	1	
	d	1	
	e		
	f	1	
	g		

Step 6: Find adjacent of f. Push e, mark it as visited. Later e will be popped and printed

Stack	Visited Array		Output
e	a	1	a, b, c, d, f, e
	b	1	
	c	1	
	d	1	
	e	1	
	f	1	
	g		

Step 7: Find adjacent of e. Push g, mark it as visited. Later g will be popped and printed

Stack	Visited Array		Output
g	a	1	a, b, c, d, f, e, g
	b	1	
	c	1	
	d	1	
	e	1	
	f	1	
	g	1	

As all nodes are visited, and stack is empty, we stop traversing.

The DFS sequence is **a, b, c, d, f, e, g**

Applications of DFS:

1. **Cycle Detection** → Detects cycles in both **directed** and **undirected** graphs.
2. **Topological Sorting** → Used in **task scheduling** for dependency resolution.
3. **Strongly Connected Components (SCCs)** → Finds SCCs in **directed graphs** (Kosaraju's Algorithm).

4. **Maze & Pathfinding** → Helps navigate mazes and find paths in AI/game development.
5. **Articulation Points & Bridges** → Identifies critical points in a network (Tarjan's Algorithm).

2.2 Breadth First

Breadth-First Search (BFS) is a graph traversal algorithm that explores **all neighbors** of a node before moving to the next level. It follows a **queue-based approach** and ensures nodes are visited in increasing order of their distance from the starting node.

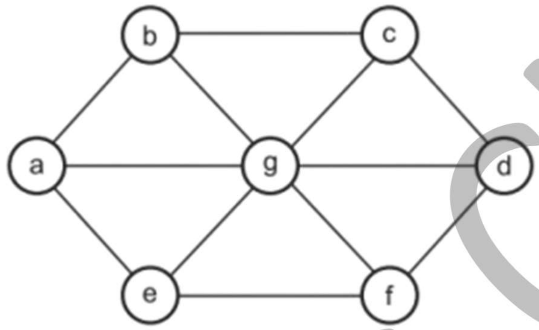
Algorithm for BFS

1. **Initialize a queue** and enqueue the starting node.
2. **Mark the starting node as visited**.
3. **While the queue is not empty**:
   - Dequeue a node from the queue.
   - Process the node (print/store it).
   - Enqueue all **unvisited adjacent nodes** and mark them as visited.
4. **Repeat until all reachable nodes are visited**.

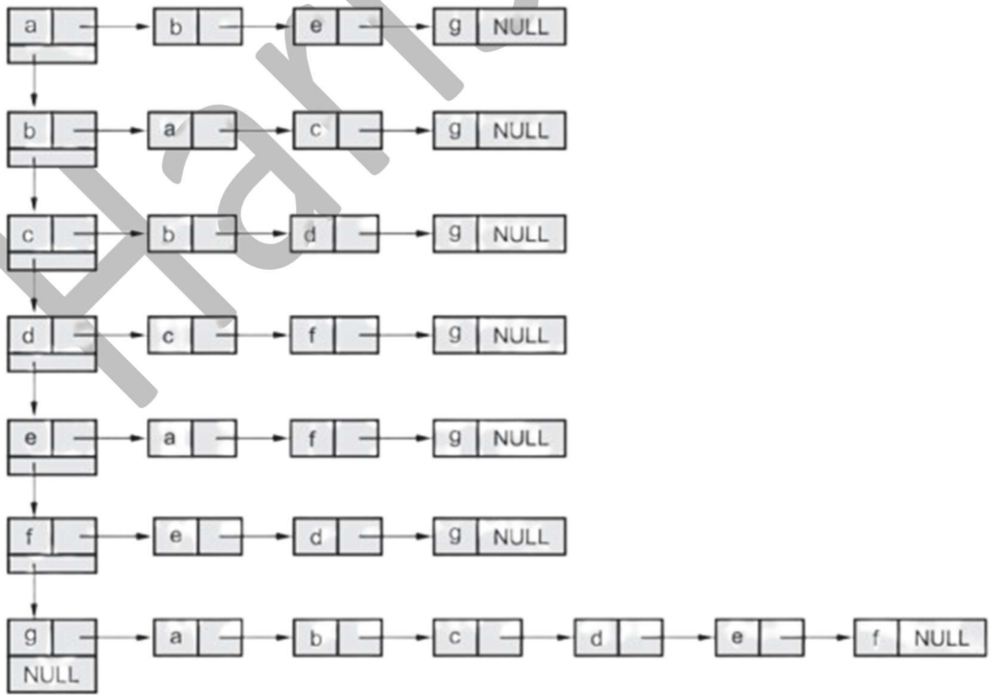
C++ Pseudocode for BFS

```
void BFS(vector<int> adj[], int start, int n) {
    vector<bool> visited(n, false); // Mark all nodes as unvisited
    queue<int> q;

    visited[start] = true; // Mark start node as visited
    q.push(start);

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";  // Process the node
        // Traverse all adjacent nodes
        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true; // Mark as visited
                q.push(neighbor);
            }
        }
    }
}
```

Example: For the graph given below, find BFS:



We will first create an adjacency list for given graph:



Step 1: Insert **a** in queue. Mark visited (a) = 1.

Queue	Visited Array		Output
a	a	1	
	b		
	c		
	d		
	e		
	f		
	g		

Step 2: Delete a and print.

Queue	Visited Array		Output
	a	1	a
	b		
	c		
	d		
	e		
	f		
	g		

Step 3: Find adjacent of a and insert them in a queue. Mark proper adjacent nodes as visited.

Queue	Visited Array		Output
b e g	a	1	a
	b	1	
	c		
	d		
	e	1	
	f		
	g	1	

Step 4: Delete b and print. Find adjacent of b and if those nodes are not visited then insert them in the queue. Mark corresponding nodes as visited.

Queue	Visited Array	Output
e g c	a 1 b 1 c 1 d e 1 f g 1	a, b

Step 5: Delete e and print. Find adjacent of e and if those nodes are not visited then insert them in a queue. Mark corresponding nodes as visited.

Queue	Visited Array	Output
g c f	a 1 b 1 c 1 d e 1 f 1 g 1	a, b, e

Step 6: Delete g and print. Find adjacent of g and if those nodes are not visited then insert them in a queue. Mark corresponding nodes as visited.

Queue	Visited Array	Output
| | | | c | f | d |	a 1 b 1 c 1 d 1 e 1 f 1 g 1	a, b, e, g

Step 7: Delete c and print. Find adjacent of c and if those nodes are not visited then insert them in a queue. Mark corresponding nodes as visited.

Queue	Visited Array	Output
| | | | | f | d |	a 1 b 1 c 1 d 1 e 1 f 1 g 1	a, b, e, g, c

Step 8: Delete f and print. Find adjacent of f and if those nodes are not visited then insert them in a queue. Mark corresponding nodes as visited.

Queue	Visited Array	Output
| | | | | | d |	a 1 b 1 c 1 d 1 e 1 f 1 g 1	a, b, e, g, c, f

Step 9: Delete d and print. Find adjacent of d and if those nodes are not visited then insert them in a queue. Mark corresponding nodes as visited.

Queue	Visited Array	Output
	a 1 b 1 c 1 d 1 e 1 f 1 g 1	a, b, e, g, c, f, d

Step 10: As queue is empty, and all entries in the visited array are visited, then stop. At display we get the BFS sequence.

BFS Sequence is **a, b, e, g, c, f, d.**

Applications of BFS

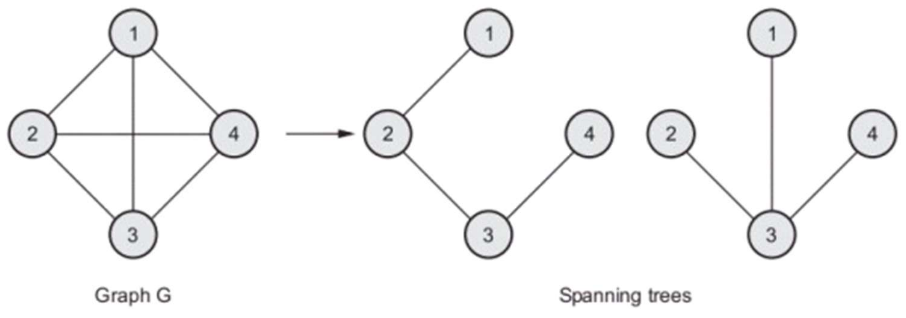
1. **Shortest Path in an Unweighted Graph** → Finds the shortest path between two nodes.
2. **Connected Components in an Undirected Graph** → Identifies distinct connected components.
3. **Bipartite Graph Checking** → Determines if a graph can be colored with two colors.
4. **Network Broadcasting** → Used in computer networks for spreading information.
5. **Web Crawling** → Used by search engines to index web pages.

2.3 Minimum Spanning Tree

A **Minimum Spanning Tree (MST)** of a graph is a **subset of edges** that:

1. **Connects all vertices** in the graph.
2. **Has no cycles** (i.e., it forms a tree).
3. **Has the minimum possible total edge weight** among all spanning trees.

It applies to **weighted, connected, and undirected graphs**.



2.4 Greedy Algorithms for Computing Minimum Spanning Tree (MST)

The Greedy technique is an algorithmic method for obtaining optimized solution for a given problem. For example, for obtaining the shortest path from source vertex to a destination vertex within a graph – we use greedy technique.

The Greedy technique is applied to find out minimum spanning tree in a graph.

Two well-known greedy algorithms are used to compute an MST:

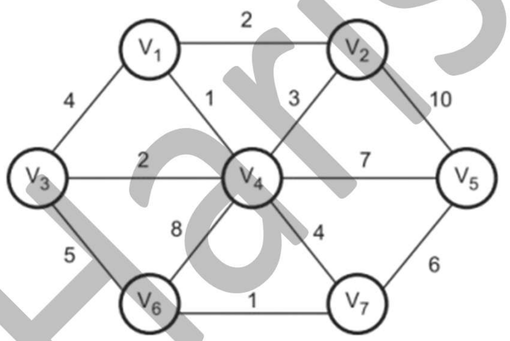
1. Prim's and Kruskal's Algorithms

Prim's Algorithm: In Prim's Algorithm, the pair with the minimum weight is to be chosen. Then adjacent to these vertices whichever is the edge having minimum weight is selected. This process will be continued till all the vertices are not covered. The necessary condition in this case is that the circuit should not be formed.

Prim's Algorithm/Pseudocode for Minimum Spanning Tree

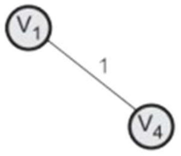
```
Prim(Graph, V):
    // Step 1: Initialize arrays
    key[V] = INF   // key[i] = minimum weight to connect i to the MST
    parent[V] = -1        // parent[i] = node that connects i to MST
    inMST[V] = false      // inMST[i] = true if i is already in MST

    key[0] = 0                               // Start from vertex 0

    // Step 2: Loop V times to include all vertices in MST
    for count = 0 to V - 1:
        // Step 3: Pick the vertex u with the minimum key[] value not in MST
        u = vertex with min key[u] such that inMST[u] == false
        inMST[u] = true                     // Include u in MST

        // Step 4: Update key and parent for adjacent vertices of u
        for each neighbor v of u:
            weight = weight of edge (u, v)
            if inMST[v] == false and weight < key[v]:
                key[v] = weight
                parent[v] = u

    // Step 5: parent[] contains the MST
    print "Edges in MST:"
    for i = 1 to V - 1:
        print parent[i], "->", i, "with weight", key[i]
```

Q. For the graph given below, construct a minimum spanning tree using Prim's algorithm. Show the table created during each pass of the algorithm.



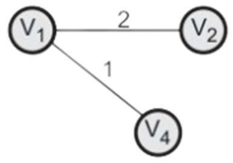
In Prim's algorithm the minimum path length is selected. We only need to make sure that the circuit should not be formed.

Step 1:



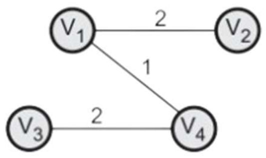
Path length = 1

Step 2:



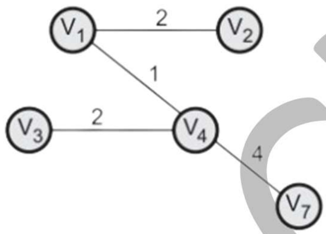
Path length = 3

Step 3:



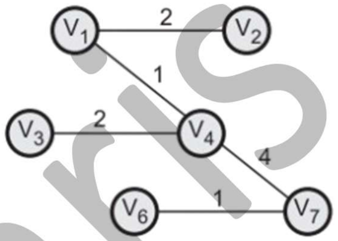
Path length = 5

Step 4:



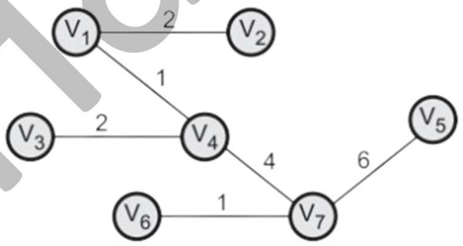
Path length = 9

Step 5:



Path length = 10

Step 6:



Path length = 16

Thus, the minimum spanning tree with path length 16 is obtained.

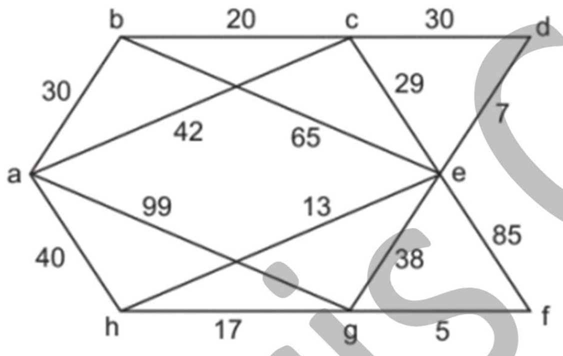
2. Kruskal's Algorithm: In Kruskal's algorithm, the minimum weight is obtained. In this algorithm, just like Prim's, circuit should not be formed. Each time the edge of minimum weight has to selected from the graph. It's not necessary in this algorithm to have edges of minimum weights to be adjacent.

Kruskal's Algorithm/Pseudocode for Minimum Spanning Tree

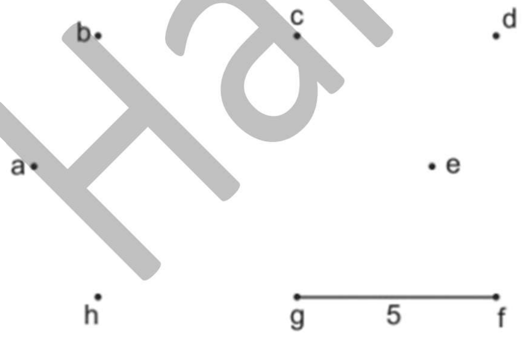
```
Kruskal(Graph, V):
    // Step 1: Create a list of all edges and sort them by weight
    edges = list of all edges in the form (u, v, weight)
    sort(edges by increasing weight)

    // Step 2: Initialize Disjoint Set (Union-Find)
    parent[V] = {0, 1, 2, ..., V-1}  // Each node is its own parent
    rank[V] = array of 0           // Used to optimize union operation

    MST = empty list       //  Final  list  of  edges  in  the  Minimum
Spanning Tree

    // Step 3: Process each edge in sorted order
    for each edge (u, v, weight) in edges:
        if findParent(u) != findParent(v):          // If u and v are
in different sets
            MST.add(edge)                     // Include edge in MST
            union(u, v)                          // Merge the sets

    // Step 4: MST is complete when it has (V - 1) edges
    return MST

// Helper function: Find the representative parent of a node
findParent(x):
    if parent[x] != x:
        parent[x] = findParent(parent[x])    // Path compression
    return parent[x]

// Helper function: Union of two sets by rank
```

```
union(x, y):
    rootX = findParent(x)
    rootY = findParent(y)

    if rank[rootX] < rank[rootY]:
        parent[rootX] = rootY
    else if rank[rootX] > rank[rootY]:
        parent[rootY] = rootX
    else:
        parent[rootY] = rootX
        rank[rootX]++
```

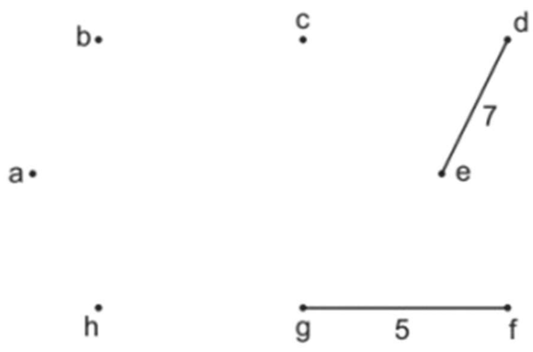
Q. Obtain Minimum Spanning Tree by Kruskal's algorithm for the following graph:



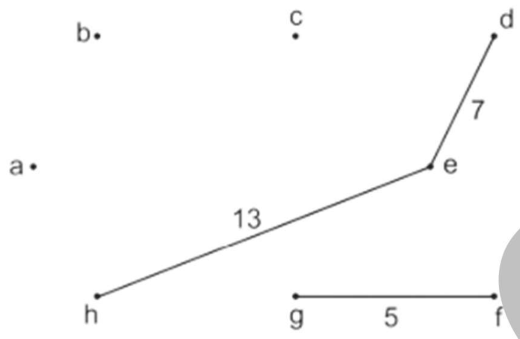
Step 1: Select an edge gf with weight 5.



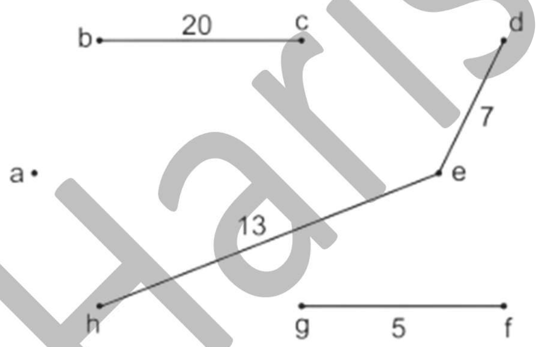
Step 2: Select an edge ed with weight 7.



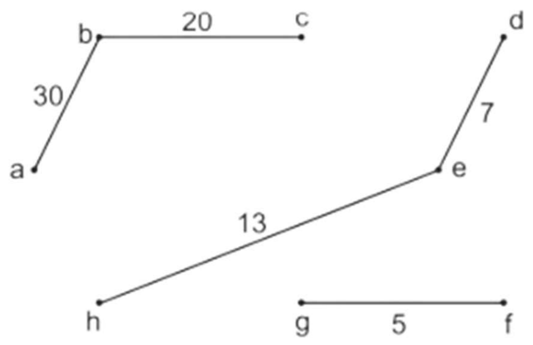
Step 3: Select an edge eh with weight 18.



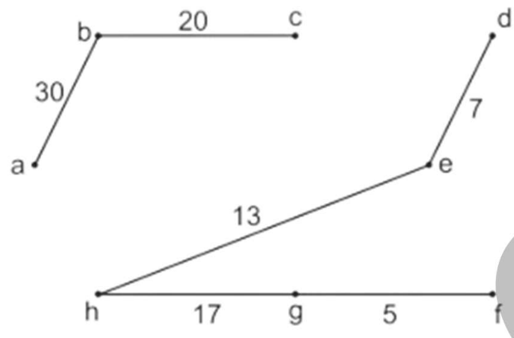
Step 4: Select an edge bc with weight 20.



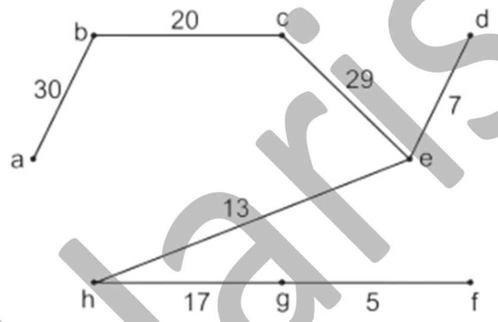
Step 5: Select an edge ab with weight 30.



Step 6: Select an edge hg with weight 17.



Step 7: Select an edge ce with weight 29.



The minimum spanning tree has the weight 121.

Comparison between Prim's and Kruskal's Algorithm

Feature	**Prim's Algorithm**	**Kruskal's Algorithm**
Approach	Expands MST from a node	Builds MST using sorted edges
Works best for	Dense graphs	Sparse graphs
Data Structure	Priority queue (min heap)	Disjoint Set (Union-Find)
Complexity	O(v^2) (Adj. Matrix) / O(E log V) (heap)	O (E log E)

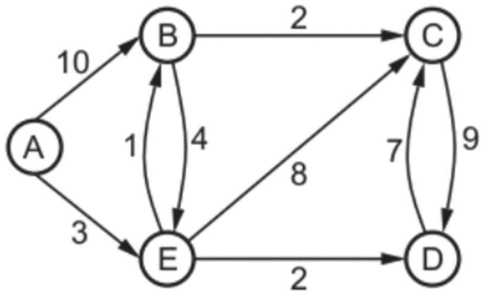
2. Dikjstra's Single Source Shortest Path

Dijkstra's Algorithm is used to find the shortest path from a **single source vertex** to all other vertices in a **weighted graph** (with non-negative weights). It works by iteratively selecting the vertex with the **minimum distance** and updating the distances of its neighboring vertices.

Algorithm/Pseudocode for Dikjstra's Single Source Shortest Path

```
Dijkstra(Graph, source, V):
    // Step 1: Initialize distance array and visited array
    dist[V] = array filled with INF         // dist[i] holds the shortest distance from source to i
    visited[V] = array filled with false    // visited[i] tracks whether node i is finalized
    dist[source] = 0  // Distance from source to itself is 0

    // Step 2: Repeat V-1 times (for all vertices)
    for count = 0 to V - 1:
        // Step 3: Pick the unvisited node with the smallest distance
        u = node with minimum dist[u] where visited[u] == false
        visited[u] = true  // Mark this node as visited

        // Step 4: Update distances of all adjacent unvisited neighbors
        for each neighbor v of u:
            if visited[v] == false and dist[u] + weight(u, v) < dist[v]:
                dist[v] = dist[u] + weight(u, v)    // Relax the edge

    // Step 5: After all vertices are processed, dist[] contains shortest distances
    for i = 0 to V - 1:
        print "Shortest distance from", source, "to", i, "=", dist[i]
```

Q. Find the shortest path in the following graph from node A, using Dijkstra's Algorithm.



`Formula: dist(v) = min(dist(v), dist(u) + weight(u,v))`

Consider source vertex A.

S = {A}, T = {B, C, D, E}

L(B) = min{∞, 0 + 10} = 10

L(C) = min{∞, 0 + ∞} = ∞

L(D) = min{∞, 0 + ∞} = ∞

L(E) = min{∞, 0 + 3} = 3 → shortest distance

S = {A, E}, T = {B, C, D}

L(B) = min{∞, 3 + 1} = 4 → shortest distance

L(C) = min{∞, 3 + 8} = 11

L(D) = min{∞, 3 + 2} = 5

S = {A, E, B}, T = {C, D}

L(E) = min{∞, 4 + 2} = 6

L(D) = min{∞, 3 + 2} = 5 → shortest distance

S = {A, E, B, D}, T = {C}

L(C) = min{∞, 4 + 2} = 6 → shortest distance

Thus we obtain shortest paths from node A as follows:

- d(A, B) = 4 A - E - B
- d(A, C) = 6 A - E - B - C
- d(A, D) = 5 A - E - D
- d(A, E) = 3 A - E

2.5 All Pairs Shortest Paths

1. Floyd-Warshall Algorithm

The **Floyd-Warshall algorithm** is a **dynamic programming** algorithm used to find the **shortest paths** between all pairs of vertices in a **weighted graph** (both directed and undirected). It works efficiently even with negative weights (but no negative cycles).

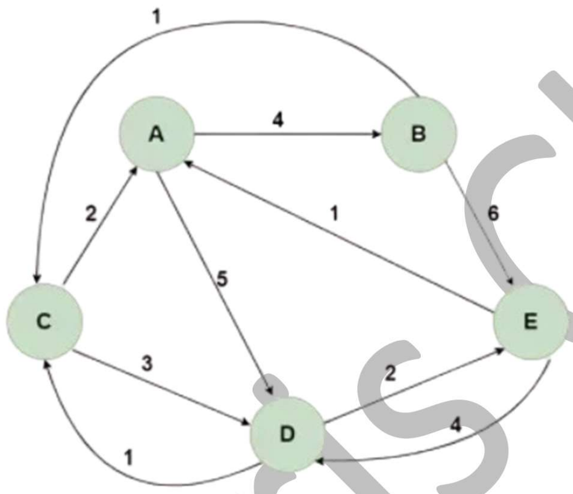
Key Features:

- **All-Pairs Shortest Path Algorithm**: Finds the shortest paths between every pair of vertices.
- **Works with Negative Weights**: Unlike Dijkstra's algorithm, it handles graphs with negative weights but **not negative cycles**.
- **Dynamic Programming Approach**: It gradually improves path estimates by considering intermediate vertices.
- **Time Complexity: O(V^3)**, where V is the number of vertices.
- **Space Complexity: O(V^2)** (stores the shortest path matrix).

Algorithm/Pseudocode:

```
Algorithm FloydWarshall(dist[][], n)
{
    // Step 1: Iterate over all possible intermediate vertices (k)
    for k from 0 to n-1 do
    {
        // Step 2: Iterate over all pairs of vertices (i, j)
        for i from 0 to n-1 do
        {
            for j from 0 to n-1 do
            {
                // Step 3: Update dist[i][j] if a shorter path exists
through vertex k
                if dist[i][j] > dist[i][k] + dist[k][j] then
                    dist[i][j] = dist[i][k] + dist[k][j]
            }
        }
    }

    // Step 4: Return the updated distance matrix containing shortest paths
    return dist
}
```

Q. Find the shortest path between all the pairs of vertices from the given graph using Floyd-Warshall Algorithm.



1. Create a matrix A^0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell A[i][j] is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

A^0 =

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

Formula: A[i][j] = min(A[i][j], A[i][k]+A[k][j])

Create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. Let k be the intermediate vertex in the shortest path from source to destination. Here, k = A. The remaining cells are filled in the following way. Examples:

$A^1[C][B] = \min(A^0[C][B],\ A^0[C][A] + A^0[A][B]) = \min(\infty,\ 2+4) = 6$

$A^1[E][B] = \min(A^0[E][B],\ A^0[E][A] + A^0[A][B]) = \min(\infty,\ 1+4) = 5$

A^1 =

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	∞
D	∞	∞	1	0	2
E	1	5	∞	4	0

To find A^2:

Here, k = B

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	∞
D	∞	∞	1	0	2
E	1	5	∞	4	0

A^2 =

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

To find A^3:

Here, k = C

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

A^3 =

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

To find A^4:

Here k = D

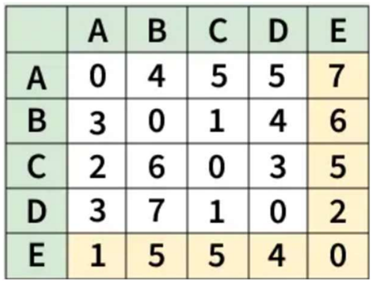
	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

A^4 =

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

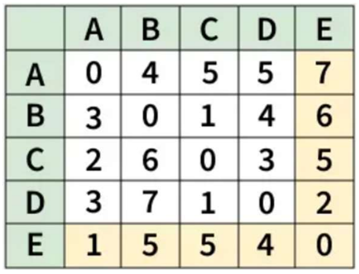
To find A^5:

Here k = E



	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

A^5 =



	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

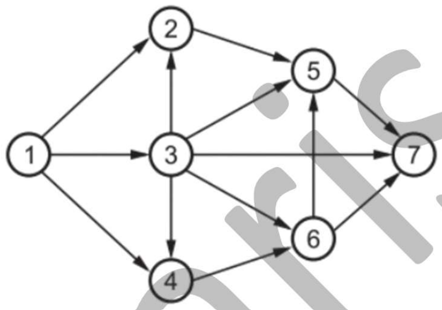
2. Topological Ordering

This algorithm is a direct implementation of the **Decrease and Conquer** method. The steps involved are as follows:

1. Identify a vertex in the given graph that has no incoming edges. Remove this vertex along with all its outgoing edges. If multiple such vertices exist, choose one randomly.
2. Record the deleted vertices in the order they are removed.
3. The sequence of recorded vertices forms the **topologically sorted order** of the graph.

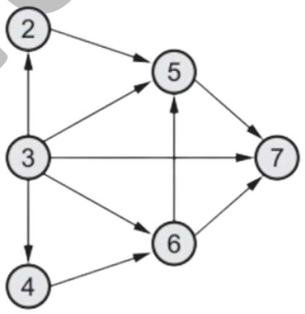
```
TopologicalSort_Kahn(Graph):
    // Step 1: Calculate in-degree of each node
    in_degree = dict with 0 for all nodes
    for each node in Graph:
        for each neighbor in node's adjacency list:
            in_degree[neighbor] += 1

    // Step 2: Enqueue all nodes with in-degree 0
    queue = empty queue
    for each node in Graph:
        if in_degree[node] == 0:
            queue.enqueue(node)

    // Step 3: Initialize list to store topological order
    topo_order = []

    // Step 4: Process nodes in queue
    while queue is not empty:
        current = queue.dequeue()
        topo_order.append(current)

        // Step 5: Decrease in-degree of neighbors
        for each neighbor of current:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.enqueue(neighbor)

    // Step 6: Check for cycles (if all nodes were not processed)
    if length of topo_order != total number of nodes:
        print("Cycle detected - topological sort not possible")
    else:
        print(topo_order)
```

Q. Find the topological sorting of the given graph:

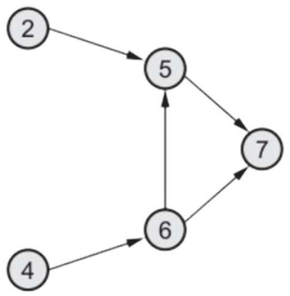


Step 1: Choose vertex ‘1’ as it has no incoming edge. Delete it along with its adjacent-edges.

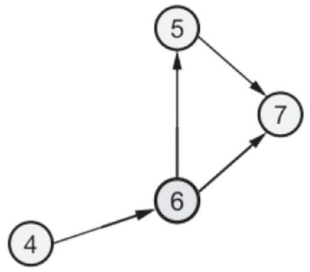
Delete 1:

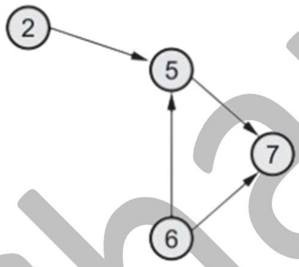


Delete 3:

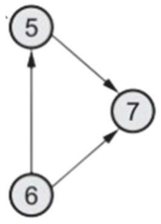


Delete 2 or 4:

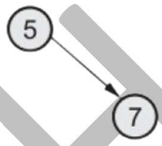




Delete 4 or 2:



Delete 6:



Delete 5:

Delete 7.

The various topological sorting will be:

- **1, 3, 2, 4, 6, 5, 7**
- **1, 3, 4, 2, 6, 5, 7**