CHAPTER 5:
GRAPHS

Content

- Introduction, Graph Terminologies
- Representation of Graph
- Graph Traversals-Depth First Search (DFS)
- Breadth First Search (BFS)
- Graph Application- Topological Sorting

Somaiya TRUST

Introduction to Graph

✅ What is a Graph?

A **graph** is a **non-linear data structure** consisting of:

- **Vertices (Nodes)** — Represent entities
- **Edges (Links)** — Represent connections between vertices

Graphs are used to model **real-world relationships**, like social networks, road maps, or the internet.

📌 Key Terminology:

Term	Meaning
Vertex (V)	A node or point in the graph
Edge (E)	A connection between two vertices
Adjacency	Two vertices connected directly by an edge
Degree	Number of edges connected to a vertex
Path	Sequence of vertices connected by edges
Cycle	A path that starts and ends at the same vertex
Connected	A graph where a path exists between every pair of vertices

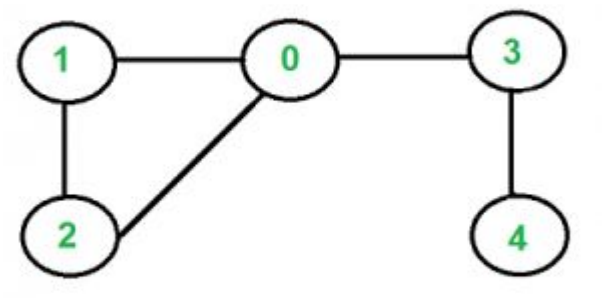
Types of Graph

🔁 Types of Graphs:

Graph Type	Description
Undirected Graph	Edges do **not** have direction (A—B is same as B—A)
Directed Graph (Digraph)	Edges have **direction** (A → B ≠ B → A)
Weighted Graph	Edges have **weights** (e.g., cost, distance)
Unweighted Graph	Edges have no weight
Cyclic Graph	Contains at least one cycle
Acyclic Graph	No cycles (e.g., DAG – Directed Acyclic Graph)
Connected Graph	There's a path between any two vertices
Disconnected Graph	Not all nodes are reachable

Somaiya TRUST

Undirected Graph

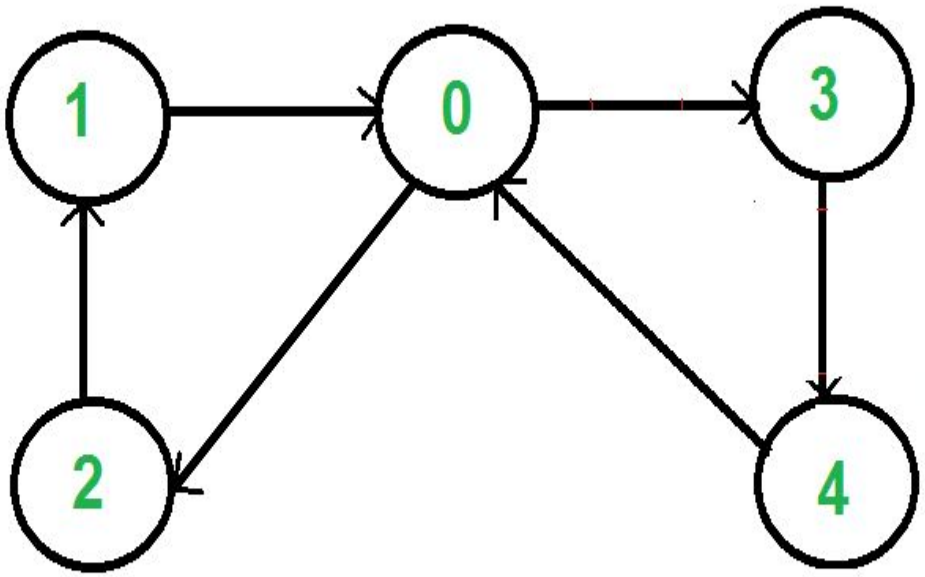


Types of Graphs:

Graph Type	Description
Undirected Graph	Edges do **not** have direction (A—B is same as B—A)
Directed Graph (Digraph)	Edges have **direction** (A → B ≠ B → A)
Weighted Graph	Edges have **weights** (e.g., cost, distance)
Unweighted Graph	Edges have no weight
Cyclic Graph	Contains at least one cycle
Acyclic Graph	No cycles (e.g., DAG – Directed Acyclic Graph)
Connected Graph	There's a path between any two vertices
Disconnected Graph	Not all nodes are reachable

Somaiya TRUST

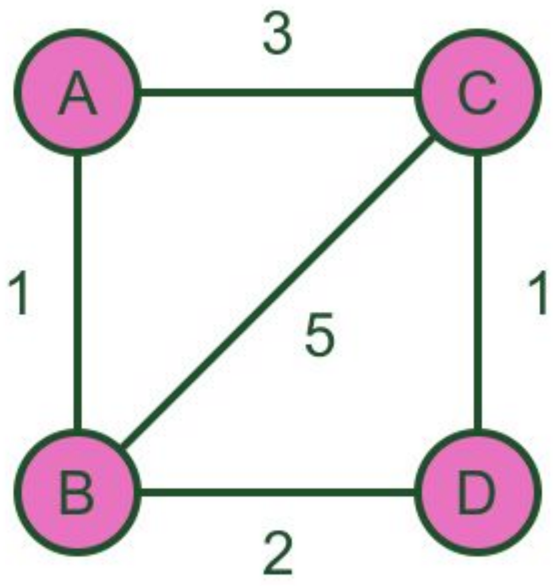
Directed Graph



Types of Graphs:

Graph Type	Description
Undirected Graph	Edges do **not** have direction (A—B is same as B—A)
Directed Graph (Digraph)	Edges have **direction** (A → B ≠ B → A)
Weighted Graph	Edges have **weights** (e.g., cost, distance)
Unweighted Graph	Edges have no weight
Cyclic Graph	Contains at least one cycle
Acyclic Graph	No cycles (e.g., DAG – Directed Acyclic Graph)
Connected Graph	There's a path between any two vertices
Disconnected Graph	Not all nodes are reachable

Weighted Graph



Weighted graph

Types of Graphs:

Graph Type	Description
Undirected Graph	Edges do **not** have direction (A—B is same as B—A)
Directed Graph (Digraph)	Edges have **direction** (A → B ≠ B → A)
Weighted Graph	Edges have **weights** (e.g., cost, distance)
Unweighted Graph	Edges have no weight
Cyclic Graph	Contains at least one cycle
Acyclic Graph	No cycles (e.g., DAG – Directed Acyclic Graph)
Connected Graph	There's a path between any two vertices
Disconnected Graph	Not all nodes are reachable

Unweighted Graph

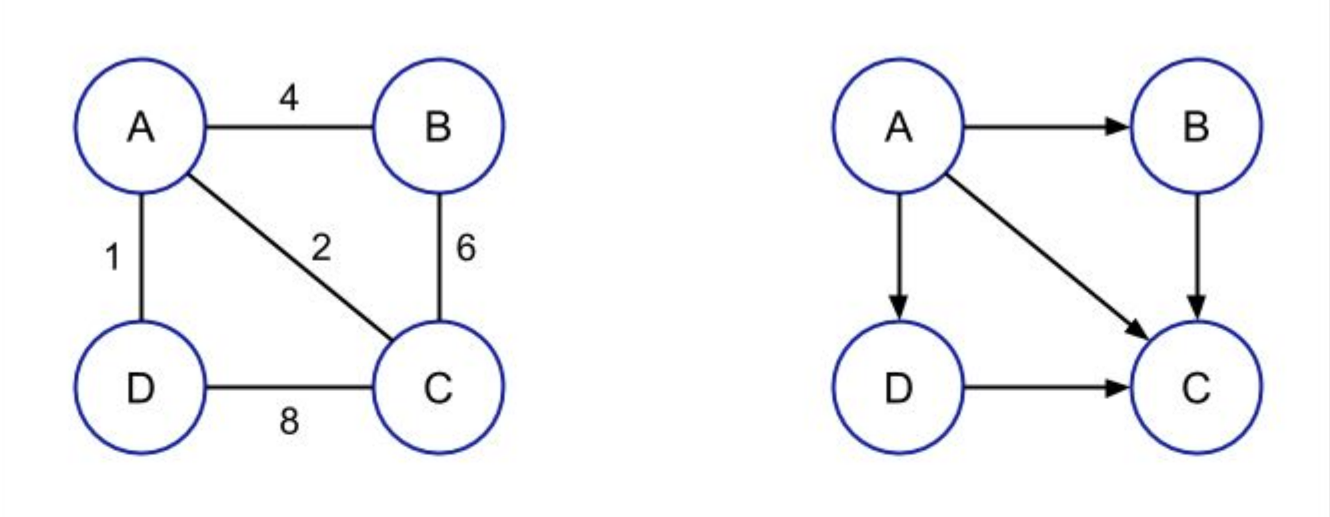


Figure: Weighted Graph (also weighted undirected graph)

Figure: Unweighted Graph (also unweighted directed graph)

Types of Graphs:

Graph Type	Description
Undirected Graph	Edges do **not** have direction (A—B is same as B—A)
Directed Graph (Digraph)	Edges have **direction** (A → B ≠ B → A)
Weighted Graph	Edges have **weights** (e.g., cost, distance)
Unweighted Graph	Edges have no weight
Cyclic Graph	Contains at least one cycle
Acyclic Graph	No cycles (e.g., DAG – Directed Acyclic Graph)
Connected Graph	There's a path between any two vertices
Disconnected Graph	Not all nodes are reachable

Cyclic Graph

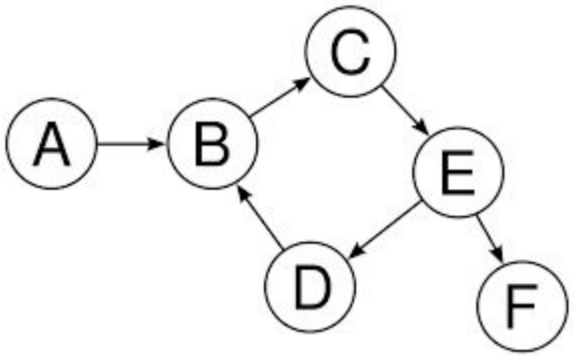


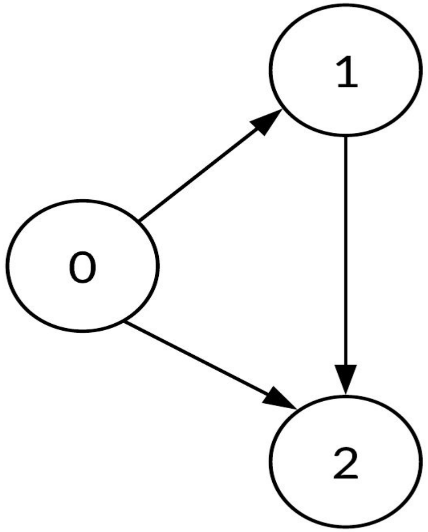
Figure 5 : Cyclic Graph

Types of Graphs:

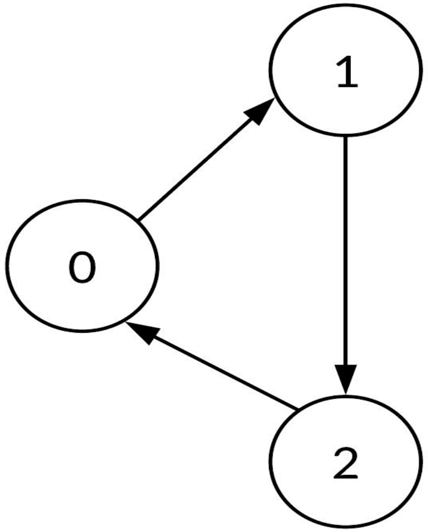
Graph Type	Description
Undirected Graph	Edges do **not** have direction (A—B is same as B—A)
Directed Graph (Digraph)	Edges have **direction** (A → B ≠ B → A)
Weighted Graph	Edges have **weights** (e.g., cost, distance)
Unweighted Graph	Edges have no weight
Cyclic Graph	Contains at least one cycle
Acyclic Graph	No cycles (e.g., DAG – Directed Acyclic Graph)
Connected Graph	There's a path between any two vertices
Disconnected Graph	Not all nodes are reachable

Acyclic Graph

Acyclic Graph



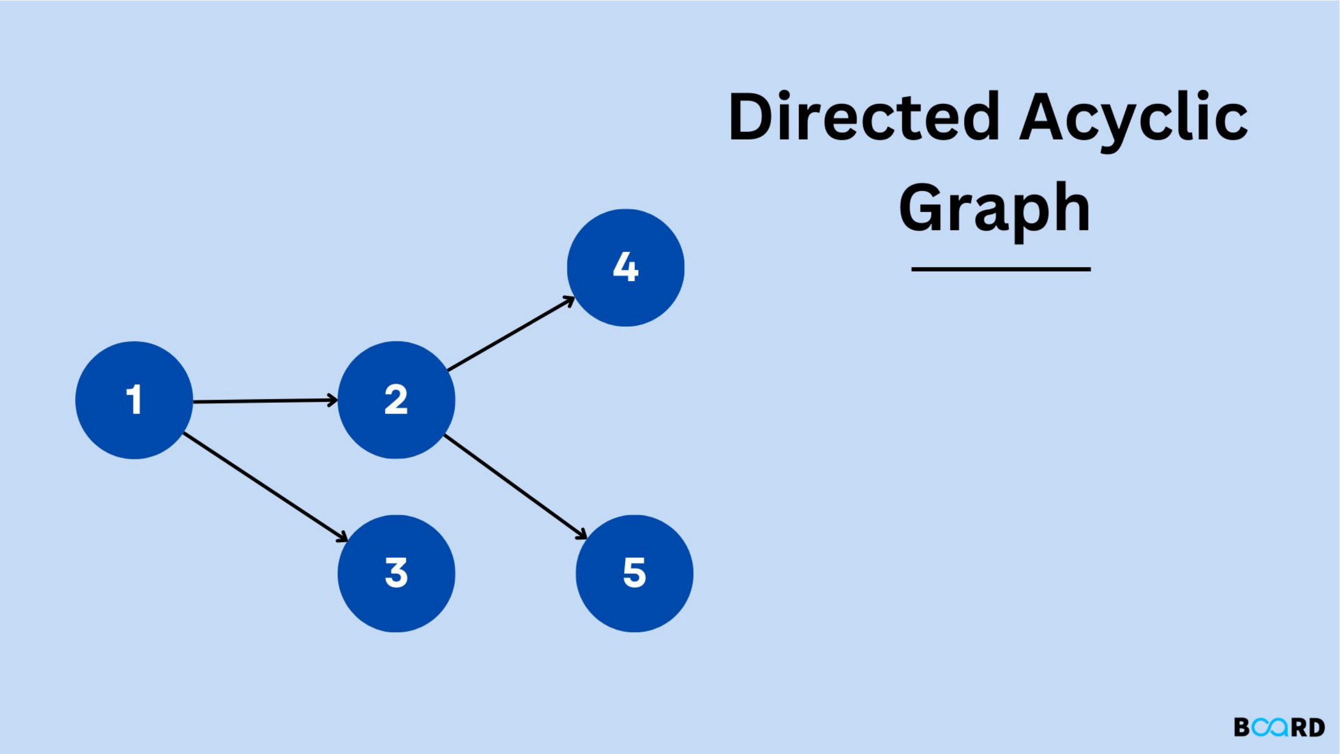
Cyclic Graph



Types of Graphs:

Graph Type	Description
Undirected Graph	Edges do **not** have direction (A—B is same as B—A)
Directed Graph (Digraph)	Edges have **direction** (A → B ≠ B → A)
Weighted Graph	Edges have **weights** (e.g., cost, distance)
Unweighted Graph	Edges have no weight
Cyclic Graph	Contains at least one cycle
Acyclic Graph	No cycles (e.g., DAG – Directed Acyclic Graph)
Connected Graph	There's a path between any two vertices
Disconnected Graph	Not all nodes are reachable

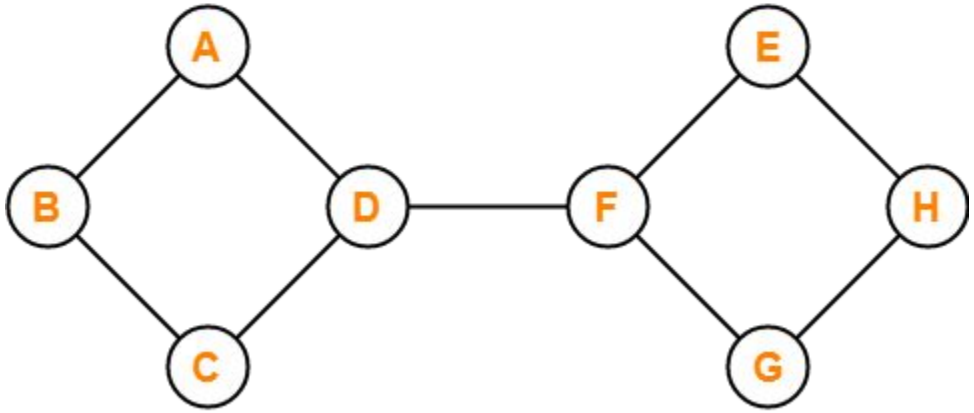
Directed Acyclic Graph



Types of Graphs:

Graph Type	Description
Undirected Graph	Edges do not have direction (A—B is same as B—A)
Directed Graph (Digraph)	Edges have **direction** (A → B ≠ B → A)
Weighted Graph	Edges have **weights** (e.g., cost, distance)
Unweighted Graph	Edges have no weight
Cyclic Graph	Contains at least one cycle
Acyclic Graph	No cycles (e.g., DAG – Directed Acyclic Graph)
Connected Graph	There's a path between any two vertices
Disconnected Graph	Not all nodes are reachable

Connected Graph

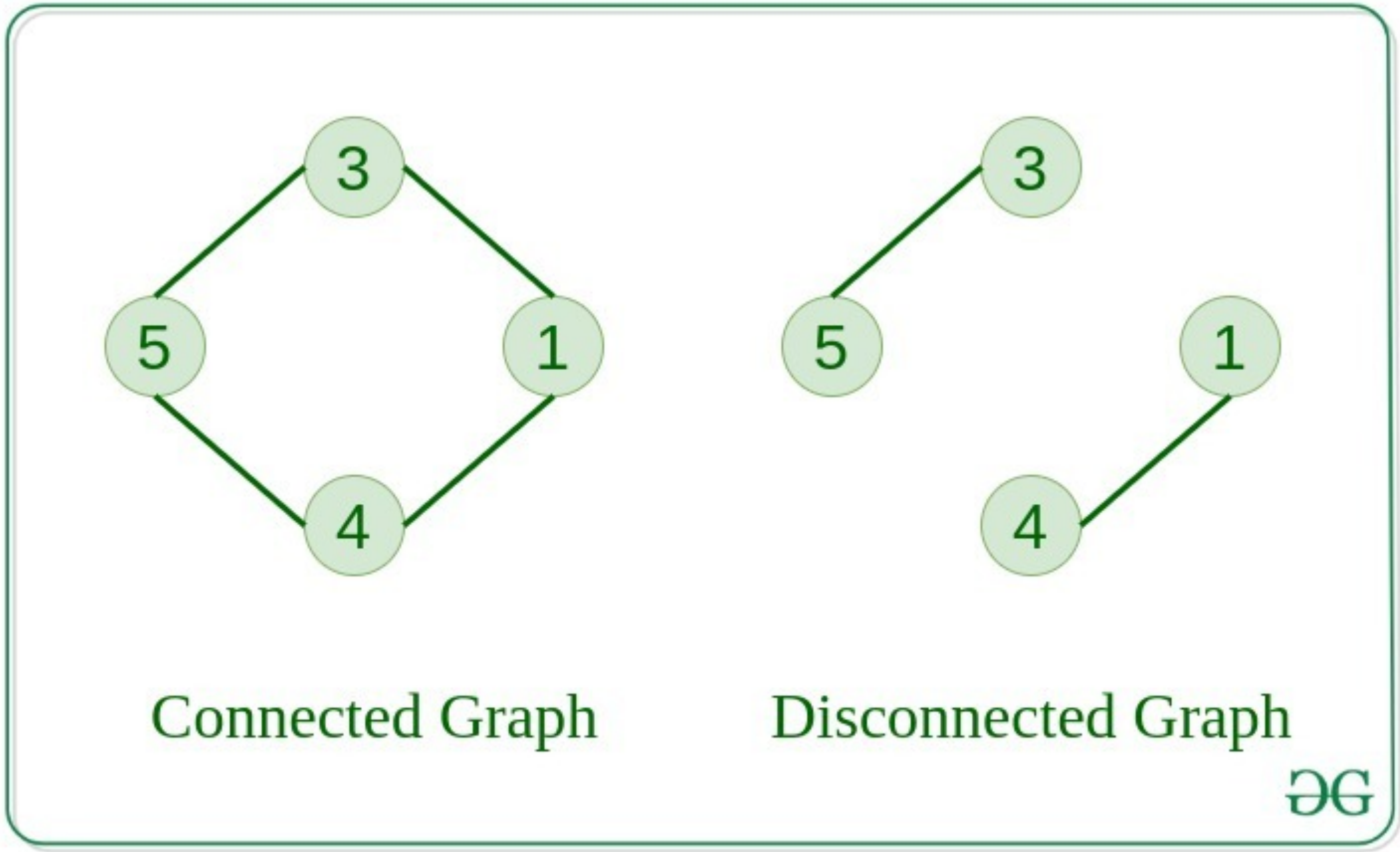


Example of Connected Graph

Types of Graphs:

Graph Type	Description
Undirected Graph	Edges do **not** have direction (A—B is same as B—A)
Directed Graph (Digraph)	Edges have **direction** (A → B ≠ B → A)
Weighted Graph	Edges have **weights** (e.g., cost, distance)
Unweighted Graph	Edges have no weight
Cyclic Graph	Contains at least one cycle
Acyclic Graph	No cycles (e.g., DAG – Directed Acyclic Graph)
Connected Graph	There's a path between any two vertices
Disconnected Graph	Not all nodes are reachable

Disconnected Graph



Types of Graphs:

Graph Type	Description
Undirected Graph	Edges do **not** have direction (A—B is same as B—A)
Directed Graph (Digraph)	Edges have **direction** (A → B ≠ B → A)
Weighted Graph	Edges have **weights** (e.g., cost, distance)
Unweighted Graph	Edges have no weight
Cyclic Graph	Contains at least one cycle
Acyclic Graph	No cycles (e.g., DAG – Directed Acyclic Graph)
Connected Graph	There's a path between any two vertices
Disconnected Graph	Not all nodes are reachable

Somaiya
TRUST

Representation of Graph

- Graphs in data structures are typically represented using **adjacency lists or adjacency matrices**

🧱 Graph Representations:

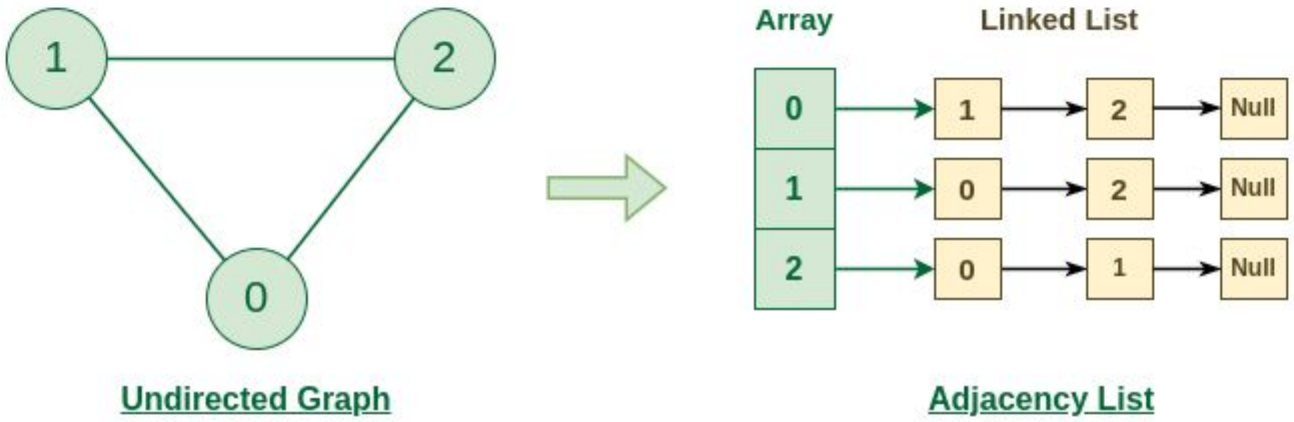
🔷 1. Adjacency Matrix

2D array of size `V x V` where `matrix[i][j] = 1` if there's an edge from `i` to `j`.

🔷 2. Adjacency List

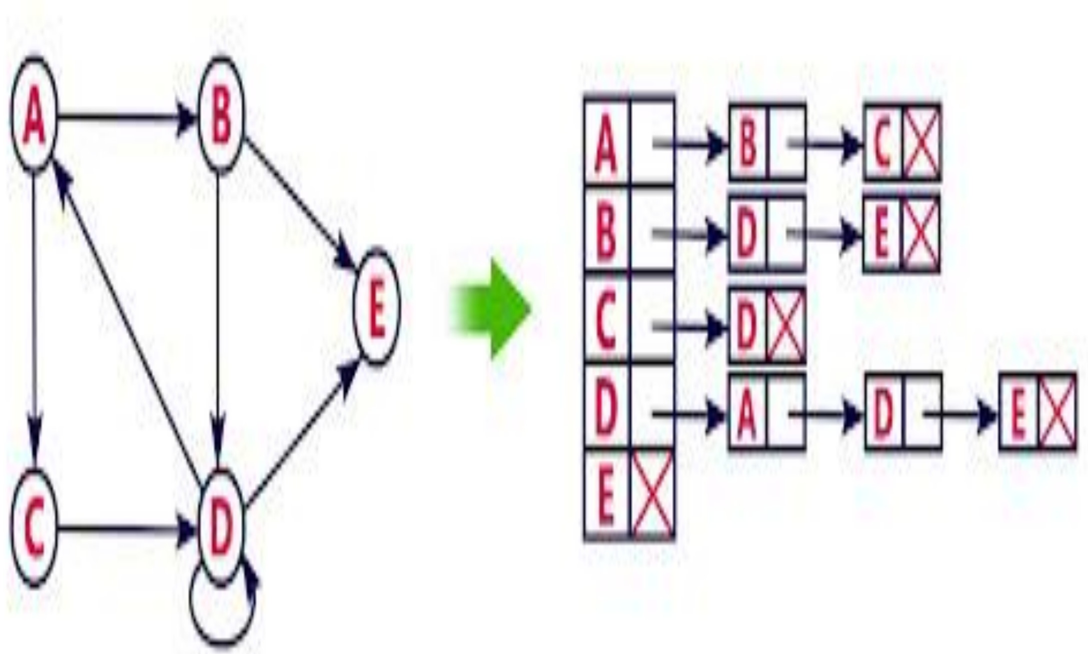
Array of linked lists or arrays where each index represents a vertex and lists its neighbors.

Adjacency List- Undirected Graph



Graph Representation of Undirected graph to Adjacency List

Adjacency List- Directed Graph



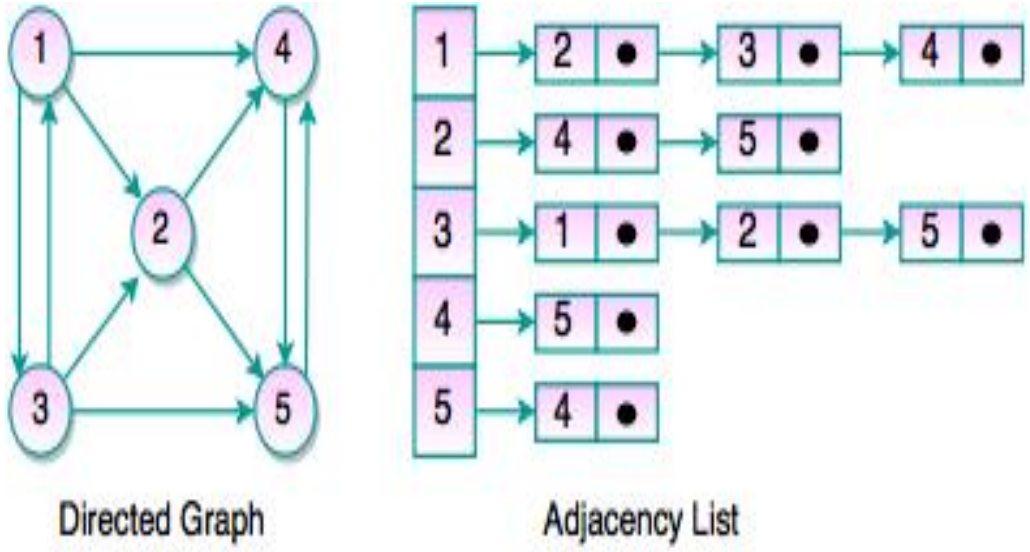
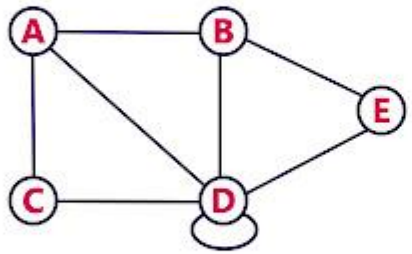


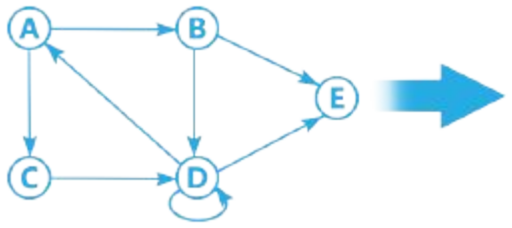
Fig. Adjacency List Representation of Directed Graph

Adjacency Matrix- Undirected Graph

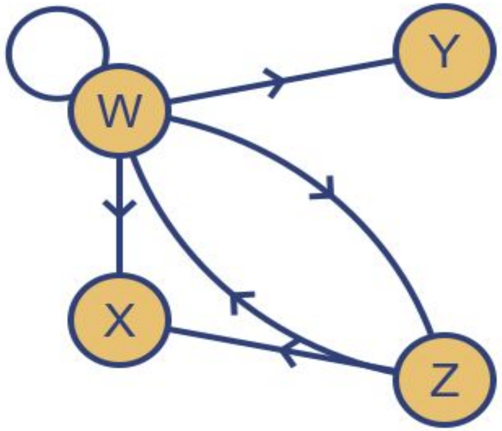


$$
\begin{array}{c c}
 & \begin{array}{ccccc} A & B & C & D & E \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} &
\begin{pmatrix}
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0
\end{pmatrix}
\end{array}
$$

Adjacency Matrix- Directed Graph



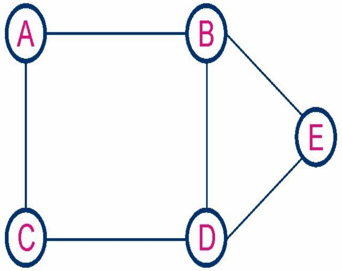
$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$



Directed graph with loop

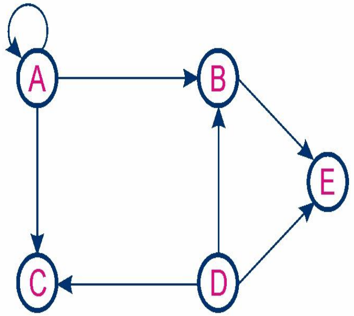
	W	X	Y	Z
W	1	1	1	1
X	0	0	0	0
Y	0	0	0	0
Z	1	1	0	0

Adjacency Matrix Example



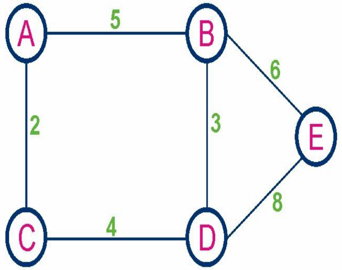
Undirected Graph

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	0	1	1
C	1	0	0	1	0
D	0	1	1	0	1
E	0	1	0	1	0



Directed Graph

	A	B	C	D	E
A	1	1	1	0	0
B	0	0	0	0	1
C	0	0	0	0	0
D	0	1	1	0	1
E	0	0	0	0	0



Weighted Graph

	A	B	C	D	E
A	0	5	2	0	0
B	5	0	0	3	6
C	2	0	0	4	0
D	0	3	4	0	8
E	0	6	0	8	0

Graph Traversal Techniques

✅ What is Graph Traversal?

Graph Traversal means **visiting all the vertices** (and possibly edges) of a graph **in a systematic way**.

It is used to explore the graph, process nodes, search for elements, or find paths.

🎯 Why Do We Traverse a Graph?

- To **search** for a specific node or value
- To **visit all nodes** (e.g., for printing or processing)
- To **find shortest paths**, connected components, or cycles
- Used in **AI**, **network routing**, **web crawling**, and **games**

Somaiya TRUST

Graph Traversal Techniques

🔁 Types of Graph Traversal:

There are two primary ways to traverse a graph:

🔷 1. **Breadth First Search (BFS)**

📌 Strategy:

- Explore all **neighbors (adjacent vertices)** first, before going deeper
- Think of it like **waves** expanding from the starting node

🧠 How it works:

- Uses a **queue**
- Starts from a **source node**
- Visits all its neighbors
- Then visits their neighbors, and so on...

🔷 2. **Depth First Search (DFS)**

📌 Strategy:

- Go as **deep** as possible down a path before backtracking
- Think of it like **exploring a maze**

🧠 How it works:

- Uses a **stack** (or recursion)
- Starts from a source
- Explores one path as deep as possible
- If dead end, it backtracks and explores another path

Somaiya TRUST

BFS-Graph Traversal Techniques

BFS (Breadth First Search)

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

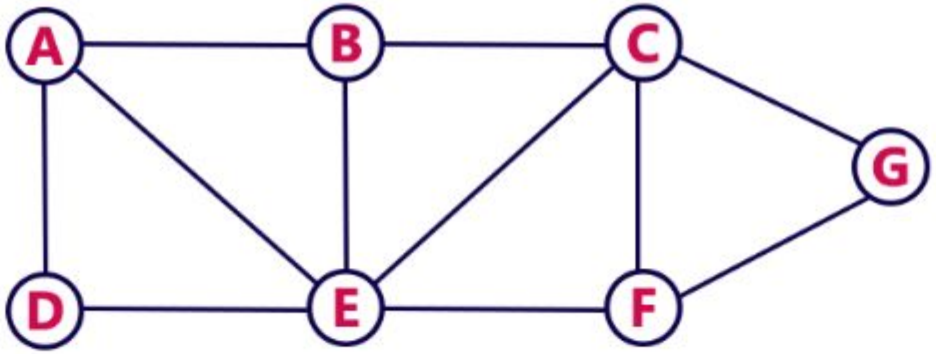
Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Somaiya TRUST

BFS-Graph Traversal Techniques

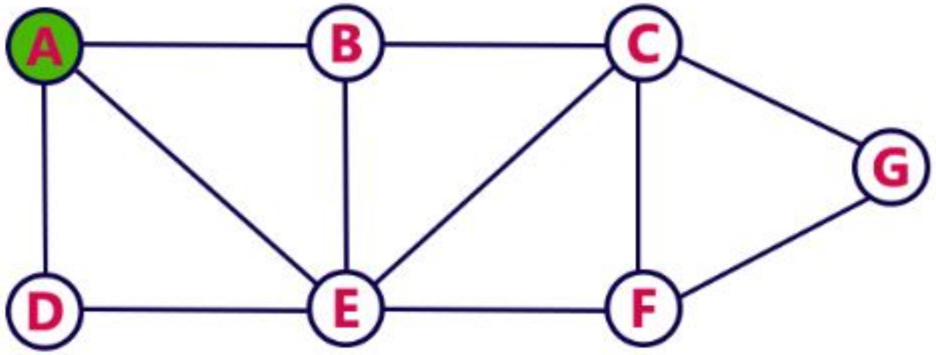
Example

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



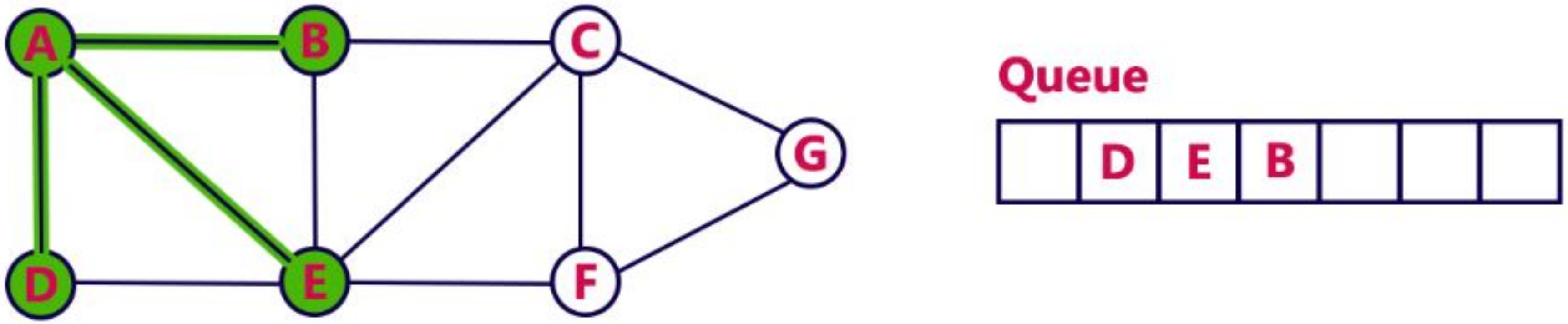
Queue

A						

BFS-Graph Traversal Techniques

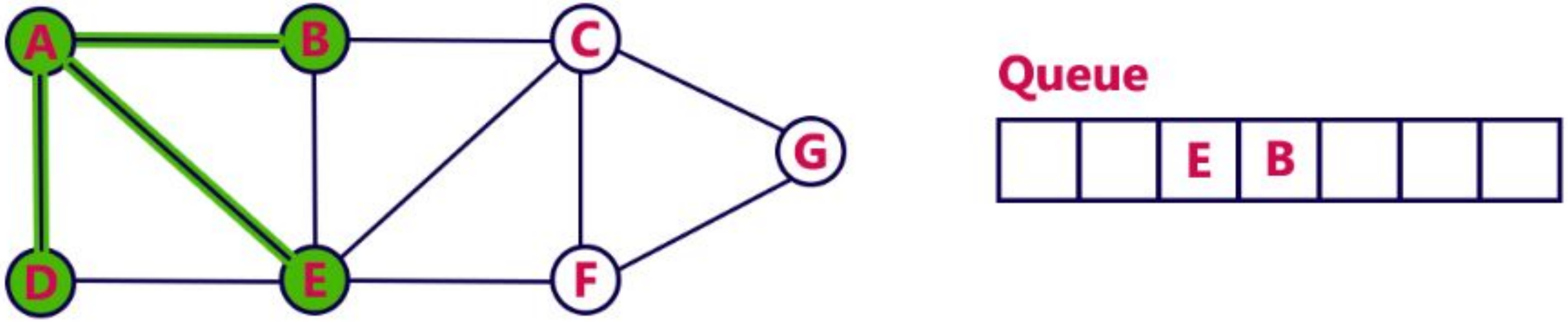
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



Step 3:

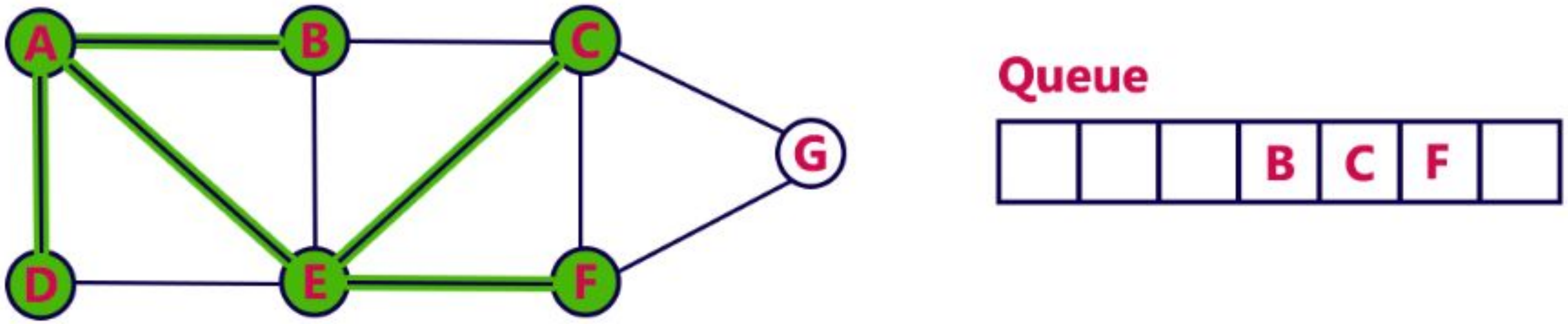
- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



BFS-Graph Traversal Techniques

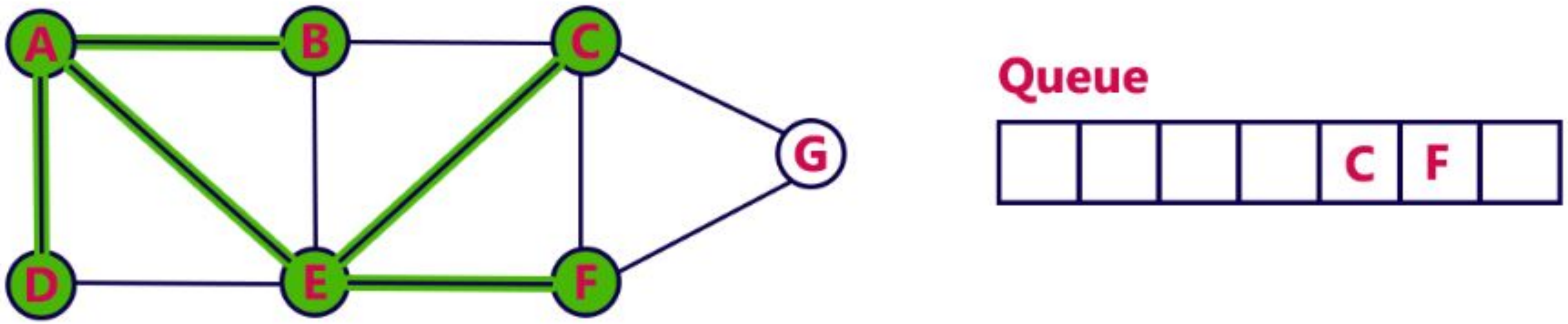
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



Step 5:

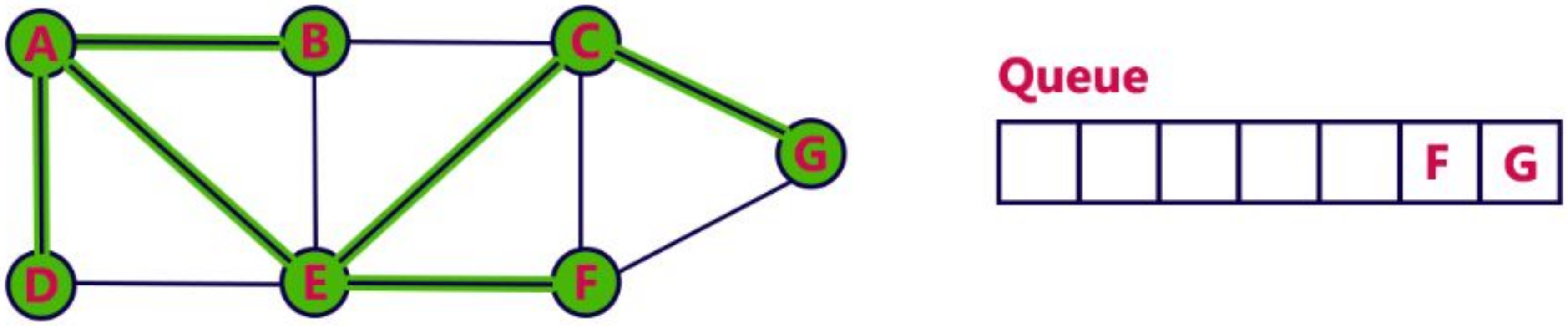
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



BFS-Graph Traversal Techniques

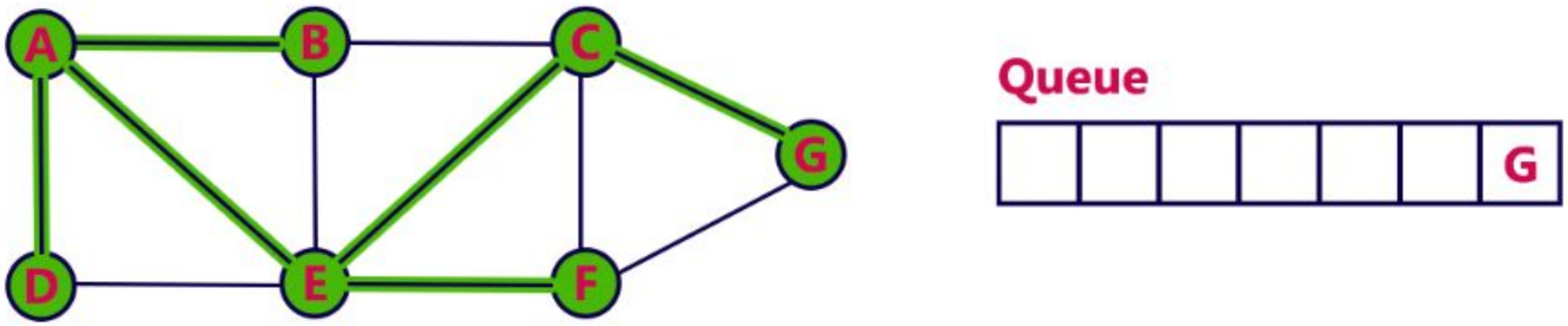
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

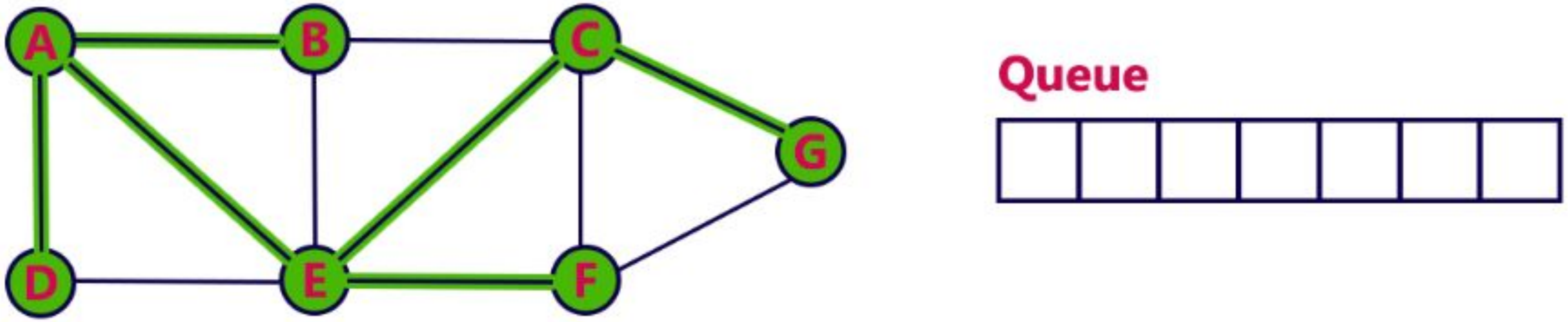


Somaiya TRUST

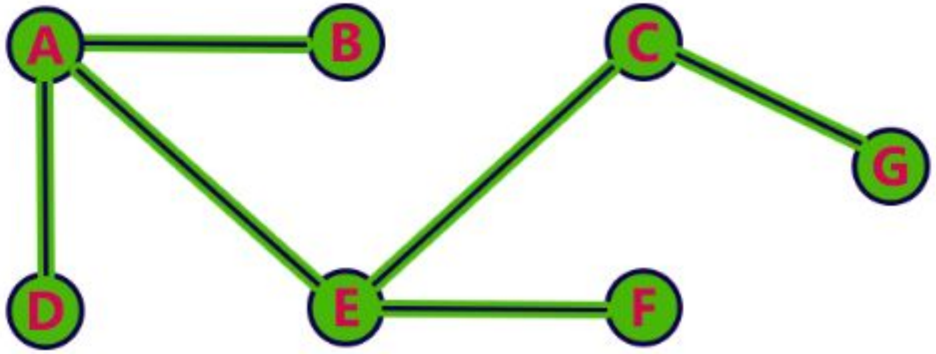
BFS-Graph Traversal Techniques

Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



DFS-Graph Traversal Techniques

DFS (Depth First Search)

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

Step 3 - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.

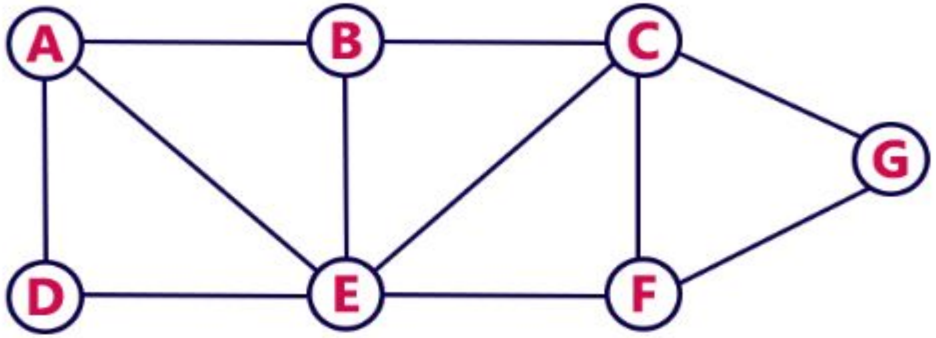
Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

DFS-Graph Traversal Techniques

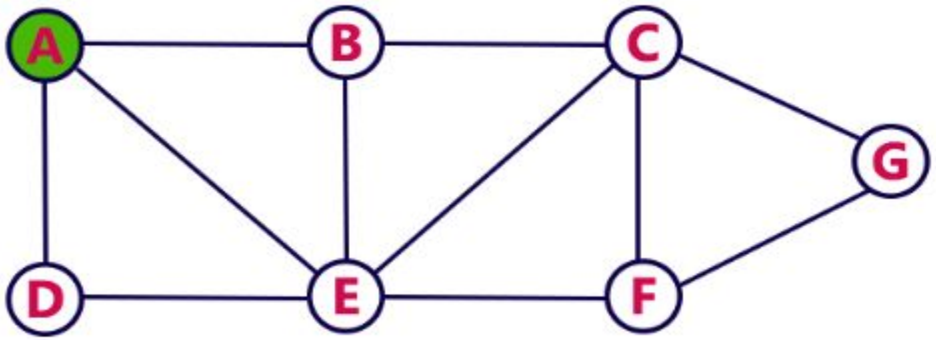
Example

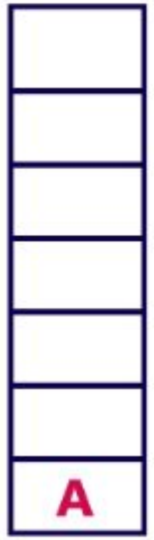
Consider the following example graph to perform DFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



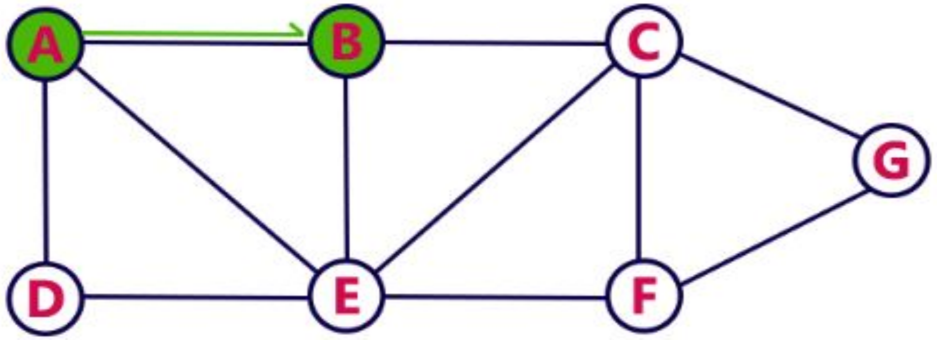


Stack

DFS-Graph Traversal Techniques

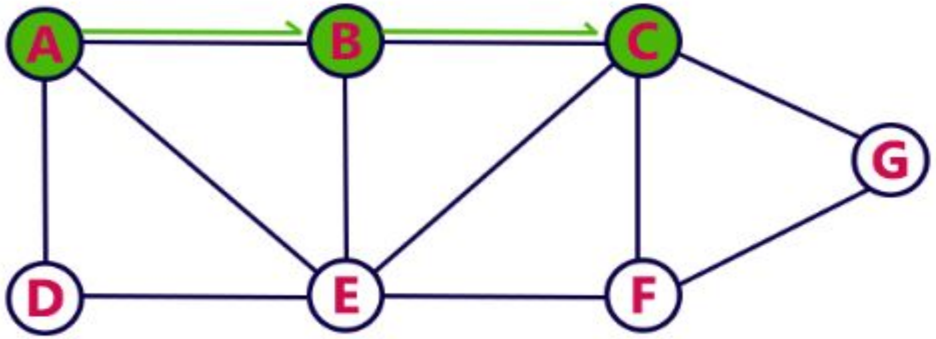
Step 2:

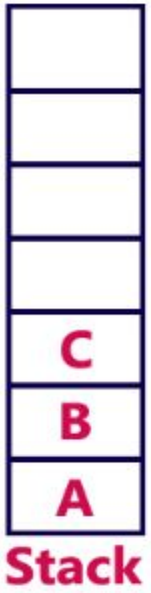
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

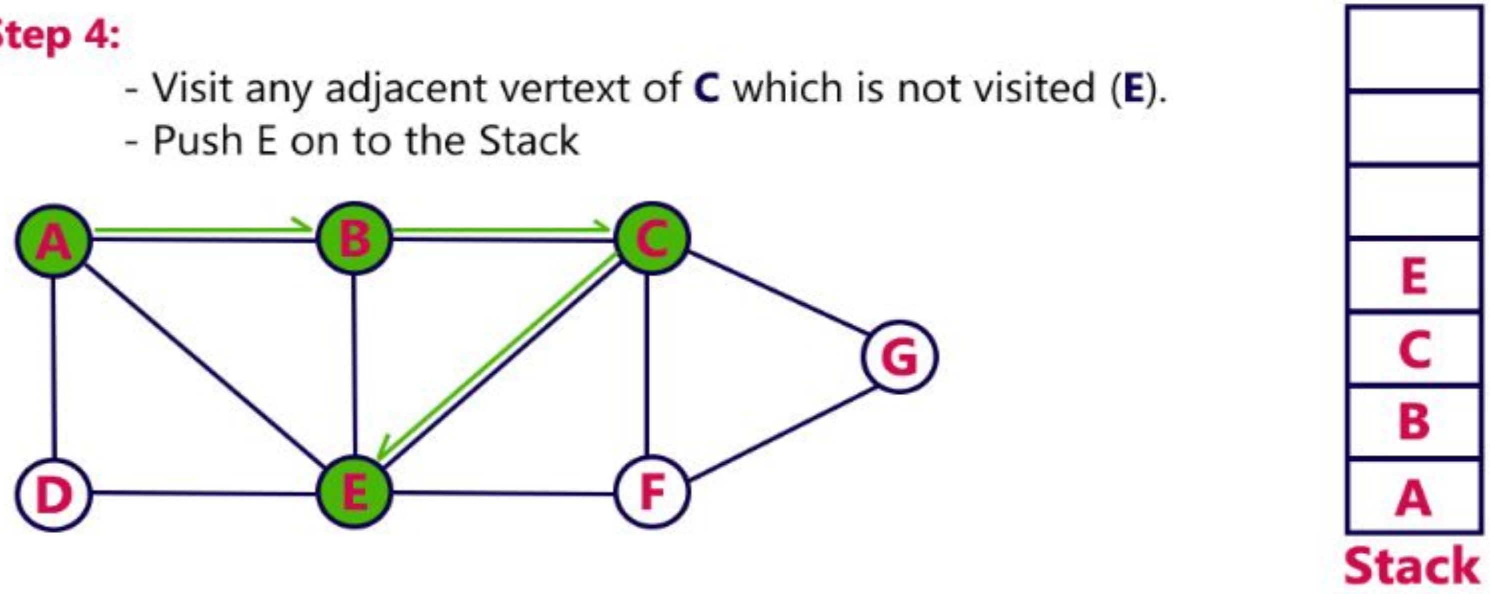




DFS-Graph Traversal Techniques

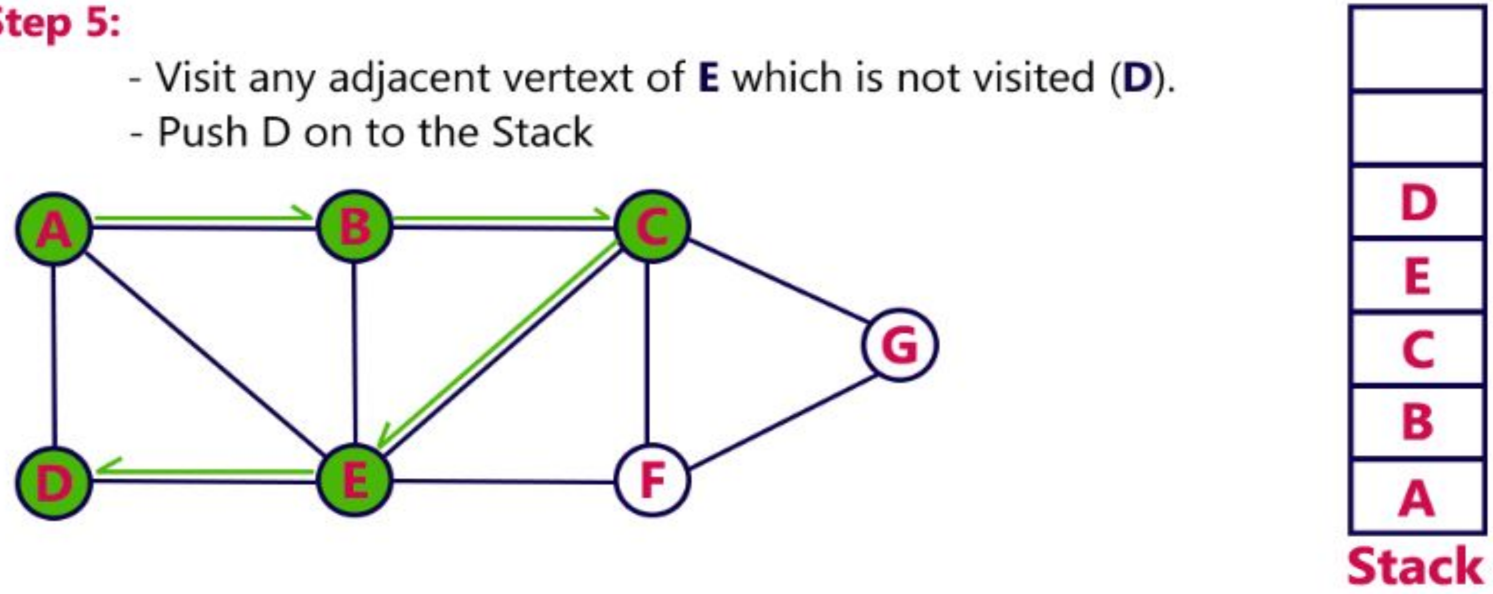
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



Step 5:

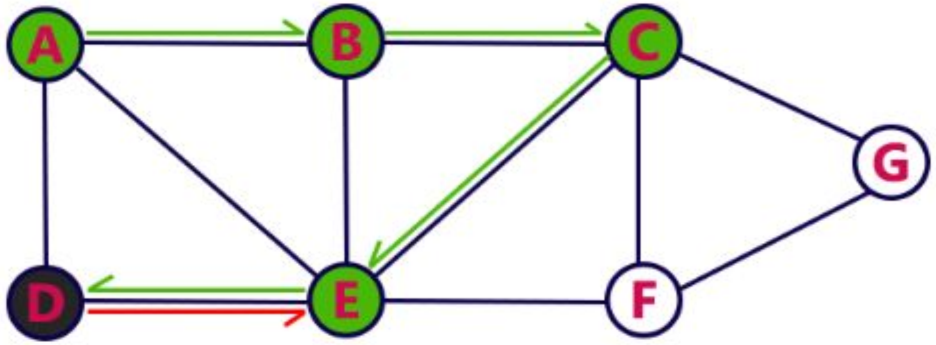
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



DFS-Graph Traversal Techniques

Step 6:

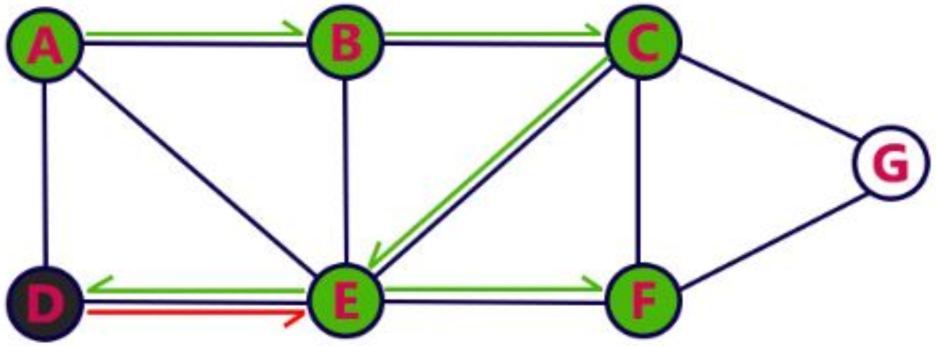
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



Stack

Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

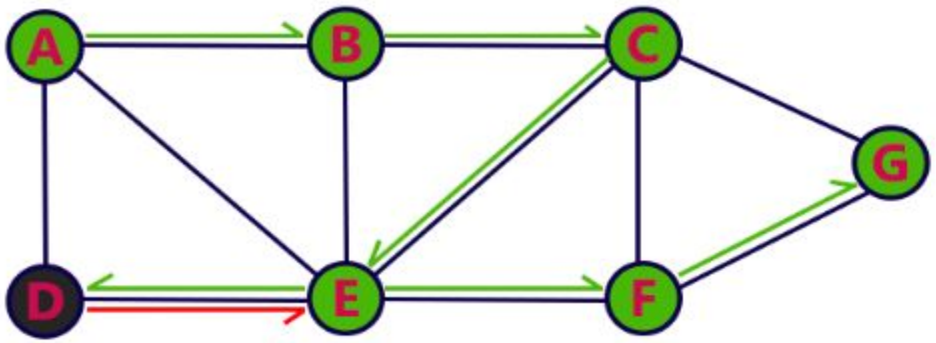


Stack

DFS-Graph Traversal Techniques

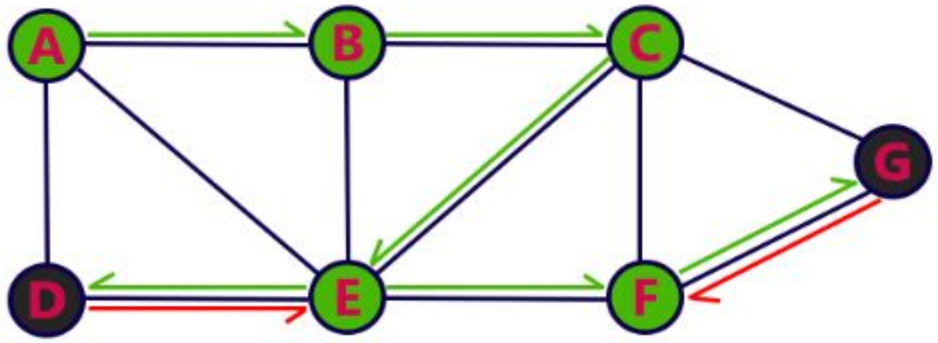
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

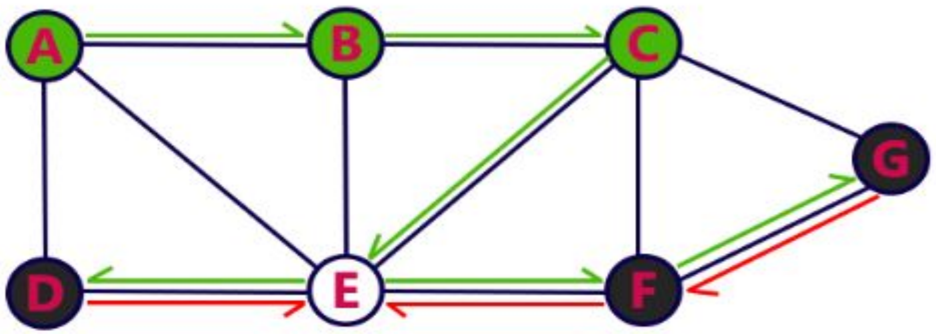


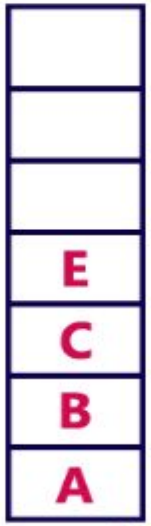
DFS-Graph Traversal Techniques

Stack

Step 10:

- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

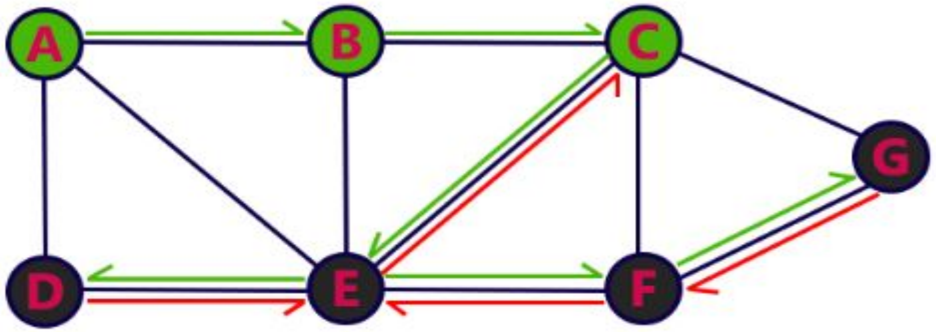


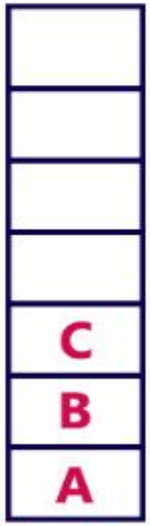


Stack

Step 11:

- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.



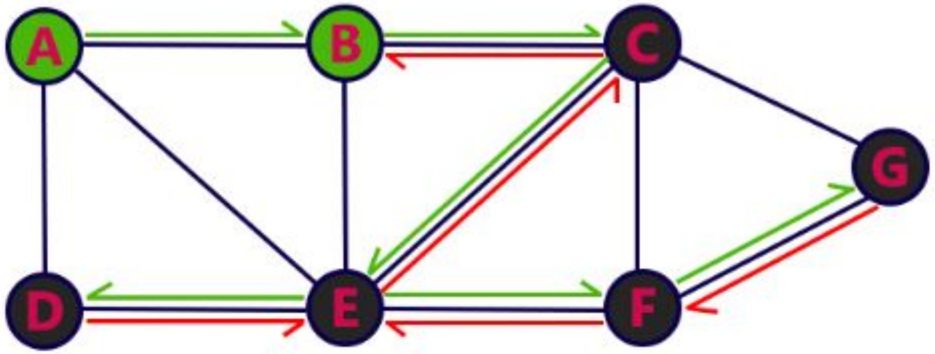


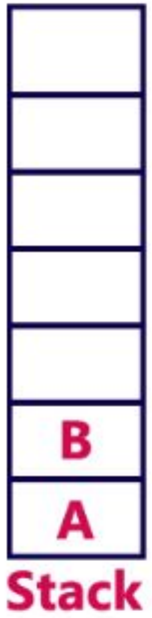
Stack

DFS-Graph Traversal Techniques

Step 12:

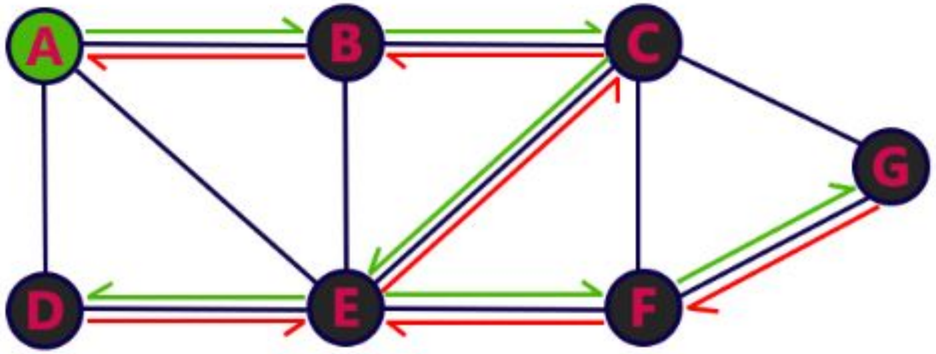
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.





Step 13:

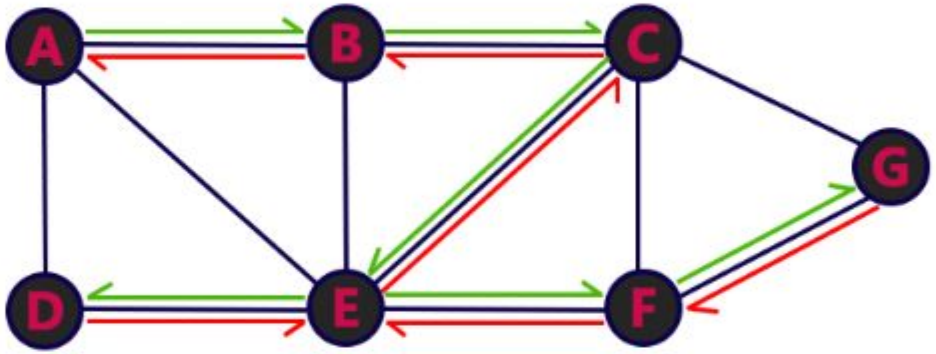
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



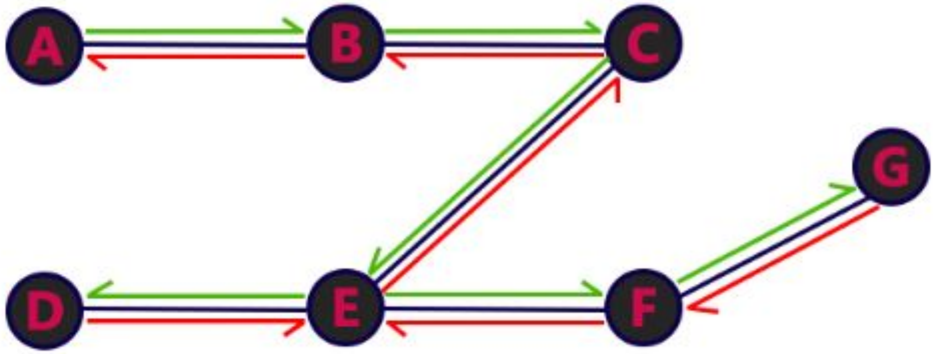
DFS-Graph Traversal Techniques

Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Application of Graph- Topological Sorting

✅ What is Topological Sorting?

Topological Sorting is a **linear ordering of vertices** in a **Directed Acyclic Graph (DAG)** such that:

> **For every directed edge** `u → v`, **vertex** `u` **comes before** `v` in the ordering.

🔁 Applicable Only To:

- **Directed Graph**
- **Acyclic** (no cycles allowed)

⚠️ If the graph has a cycle:

Topological sort is **not possible**. You must check for **cycles in DAG** using DFS (back edge detection).

Application of Graph- Topological Sorting

How to find Topological Sort

Topological order can be one of the subsets of all the permutations of all the vertices following the condition that for every directed edge $x \rightarrow y$, x will come before y in the ordering.

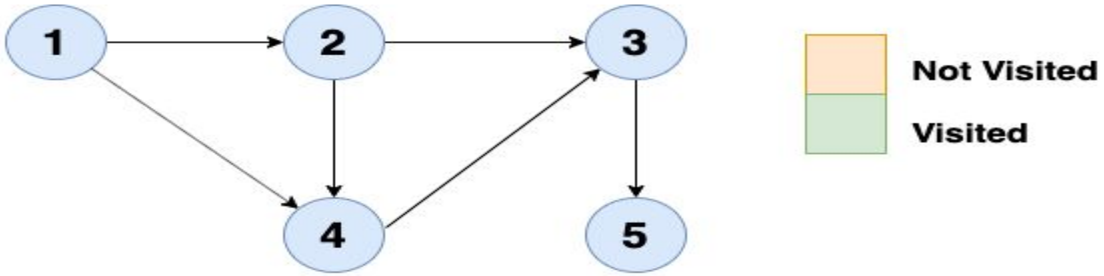
For that, we will maintain an array **T[ ],** which will store the ordering of the vertices in topological order. We will store the number of edges that are coming into a vertex in an array **in_degree[N],** where the ***i-th*** element will store the number of edges coming into the vertex ***i***. We will also store whether a certain vertex has been visited or not in **visited[N].** We will follow the belo steps:

Somaiya
TRUST

Application of Graph- Topological Sorting

- First, take out the vertex whose `in_degree` is 0. That means there is no edge that is coming into that vertex.
- We will append the vertices in the Queue and mark these vertices as visited.
- Now we will traverse through the queue and in each step we will `dequeue()` the front element in the Queue and push it into the **T.**
- Now, we will put out all the edges that are originated from the front vertex which means we will decrease the `in_degree` of the vertices which has an edge with the front vertex.
- Similarly, for those vertices whose `in_degree` is 0, we will push it in Queue and also mark that vertex as visited. (**Hope you must be thinking its BFS but with in_degree)**

Application of Graph- Topological Sorting



Step 1

Queue = [1]

in_degree[ ]

1	2	3	4	5
0	1	2	2	1

T = [ ]

Step 2

Queue = [2]

in_degree[ ]

1	2	3	4	5
0	0	2	1	1

T = [1]

Step 3

Queue = [4]

in_degree[ ]

1	2	3	4	5
0	0	1	0	1

T = [1, 2]

Step 4

Queue = [3]

in_degree[ ]

1	2	3	4	5
0	0	0	0	1

T = [1, 2, 4]

Step 5

Queue = [5]

in_degree[ ]

1	2	3	4	5
0	0	0	0	0

T = [1, 2, 4, 3]

Step 6

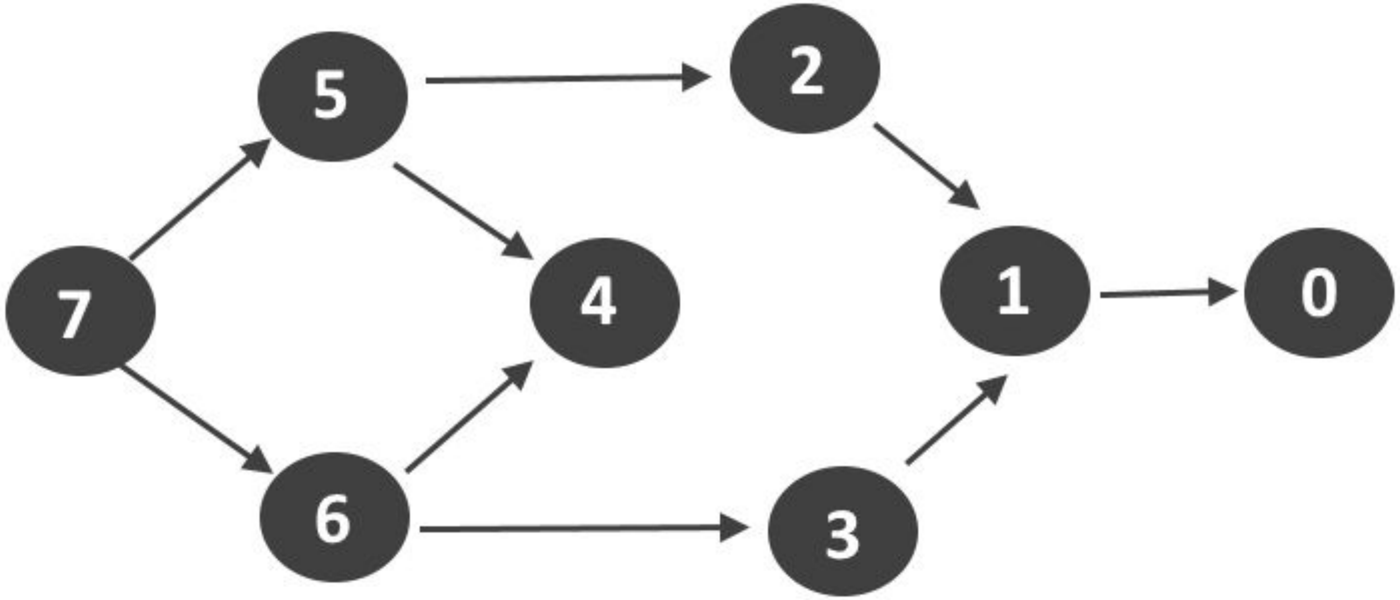
Queue = [ ]

in_degree[ ]

1	2	3	4	5
0	0	0	0	0

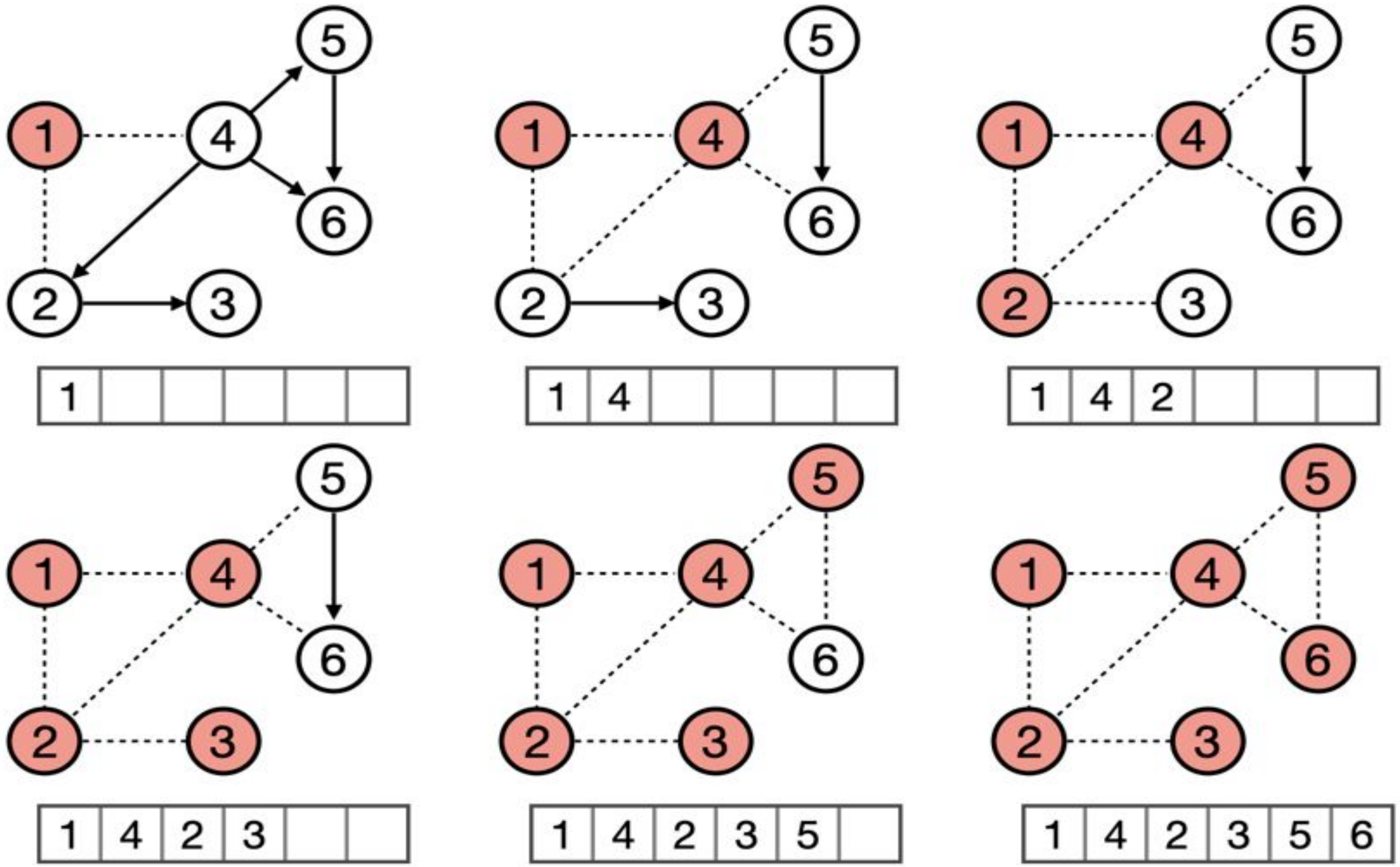
T = [1, 2, 4, 3, 5]

Application of Graph- Topological Sorting EXAMPLE

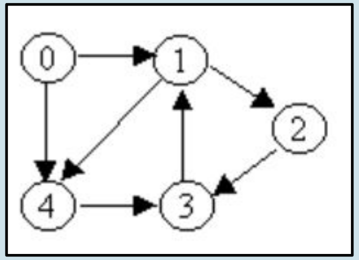
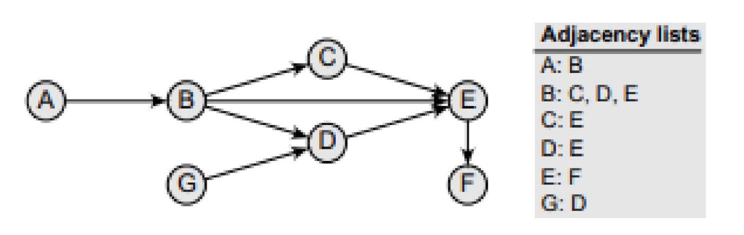


Topological Sort : 7 6 5 4 3 2 1 0

Application of Graph- Topological Sorting EXAMPLE



SAMPLE QUESTIONS

QUESTION NO.	SAMPLE QUESTIONS MODULE 5	
1	Give the DFS and Breadth-first traversal of the graph for the following graph, starting from vertex 0. Show all the steps. 	
2	Consider a directed acyclic graph G given below Find a topological sort T of G. 	
3	Explain the graph traversal techniques in detail with examples	
5	Explain How is graph represented in memory with diagram and example.	

Somaiya TRUST