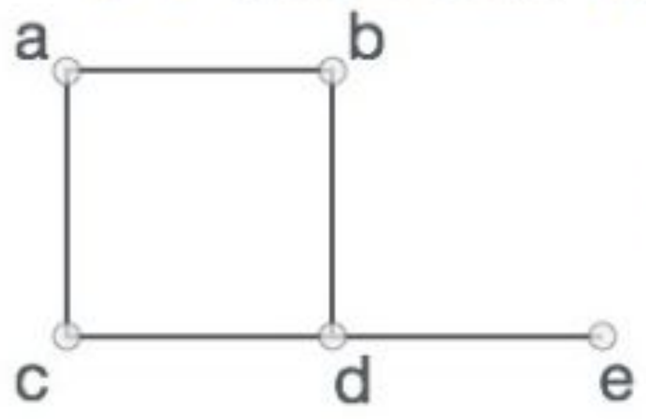
Chapter 5
Graph

Graphs

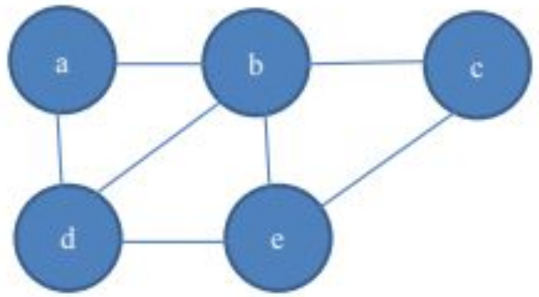
- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links.
- The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.
- Formally, a graph is a pair of sets **(V, E),** where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices.



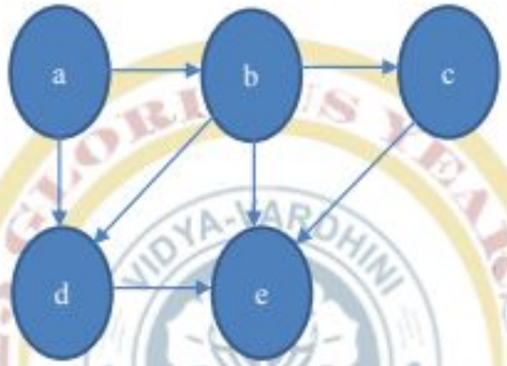
In the above graph,

V = {a, b, c, d, e}   E = {ab, ac, bd, cd, de}

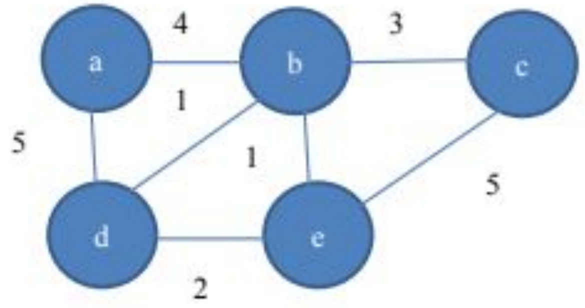
Types of Graphs



Undirected Graph



Directed Graph



Weighted Graph

Graph Data Structure

- Mathematical graphs can be represented in data-structure. We can represent a graph using an array of vertices and a two dimensional array of edges.
- Before we proceed further, let's familiarize with some important terms −
- **Vertex** − Each node of the graph is represented as a vertex.
- **Edge** − Edge represents a path between two vertices or a line between two vertices.
- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge.
- **Path** − Path represents a sequence of edges between two vertices.
- **Cycle**: A path that starts and ends on the same vertex

- **Directed Graph**: A graph that entail edges with ordered pair of vertices and has direction indicated with an arrow.

- **Undirected Graph**: A graph that entail edges with ordered pair of vertices, however it does not have direction define.

- **Weighted Graph**: A graph in which each edge carries a value.

- **Complete Graph**: A graph in which every vertex is directly connected to every other vertex.
- In a complete graph every vertex is adjacent to every other vertex.
- If there are N vertices then there will be N(N-1) edges in case of complete directed graph and N(N-1)/2 in case of complete undirected graph.

- **Degree**
- In an undirected graph degree of a vertex V is number of edges connected to vertex V.
- In a directed graph *out-degree* of a vertex V is number of edges leaving vertex V. It's in-degree is number of edges ending at vertex V.

- **Subgraph:** If graph G=(V, E) Then Graph G'=(V',E') is a **subgraph** of G if V' ⊆ V and E' ⊆ E

- **Tree**: undirected, connected graph with no cycles

- **Spanning tree**: A **spanning tree** of G is a connected subgraph of G that is a tree

Graph Representations

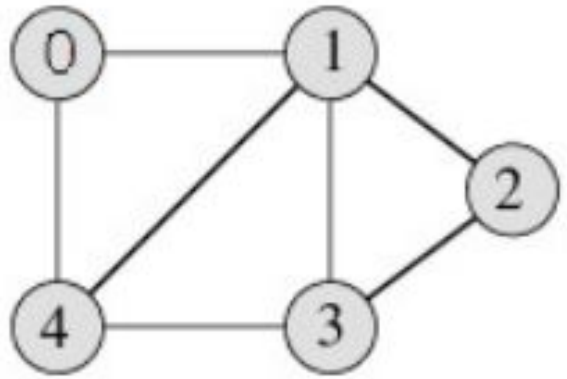
- Following two are the most commonly used representations of graph.
  1. Adjacency Matrix
  2. Adjacency List
- There are other representations also like, Incidence Matrix and Incidence List.

Adjacency Matrix

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.
- Let the 2D array be **adj[][]**,

  a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs.
- If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

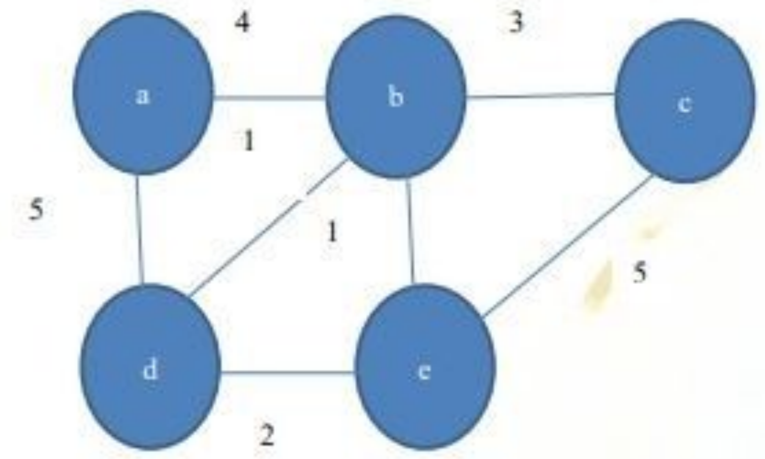
Adjacency Matrix



The adjacency matrix for the above example graph is:

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

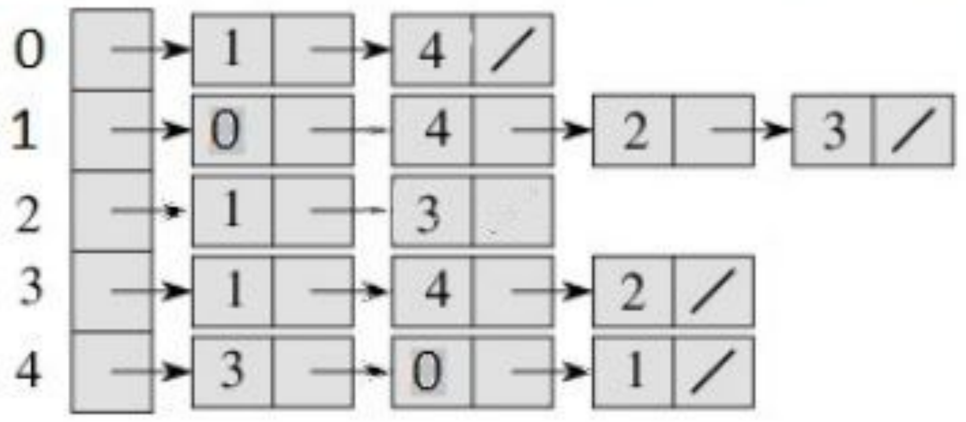
Weighted Graph with its adjacency matrix

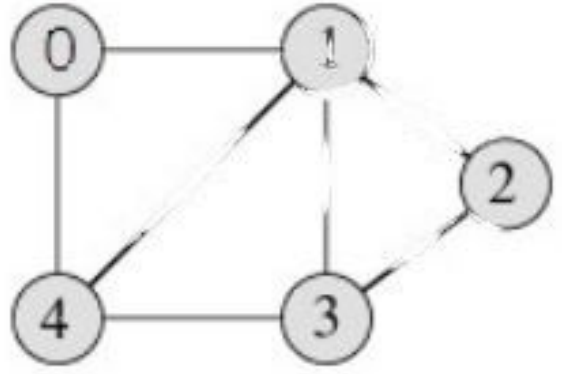


	a	b	c	d	e
a	0	4	0	5	0
b	4	0	3	1	1
c	0	3	0	0	5
d	5	1	0	0	2
e	0	1	5	2	0

Adjacency List:

- An array of linked lists is used.
- Size of the array is equal to number of vertices.
- Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ith vertex.
- This representation can also be used to represent a weighted graph.
- The weights of edges can be stored in nodes of linked lists.
- Following is adjacency list representation of the above graph.





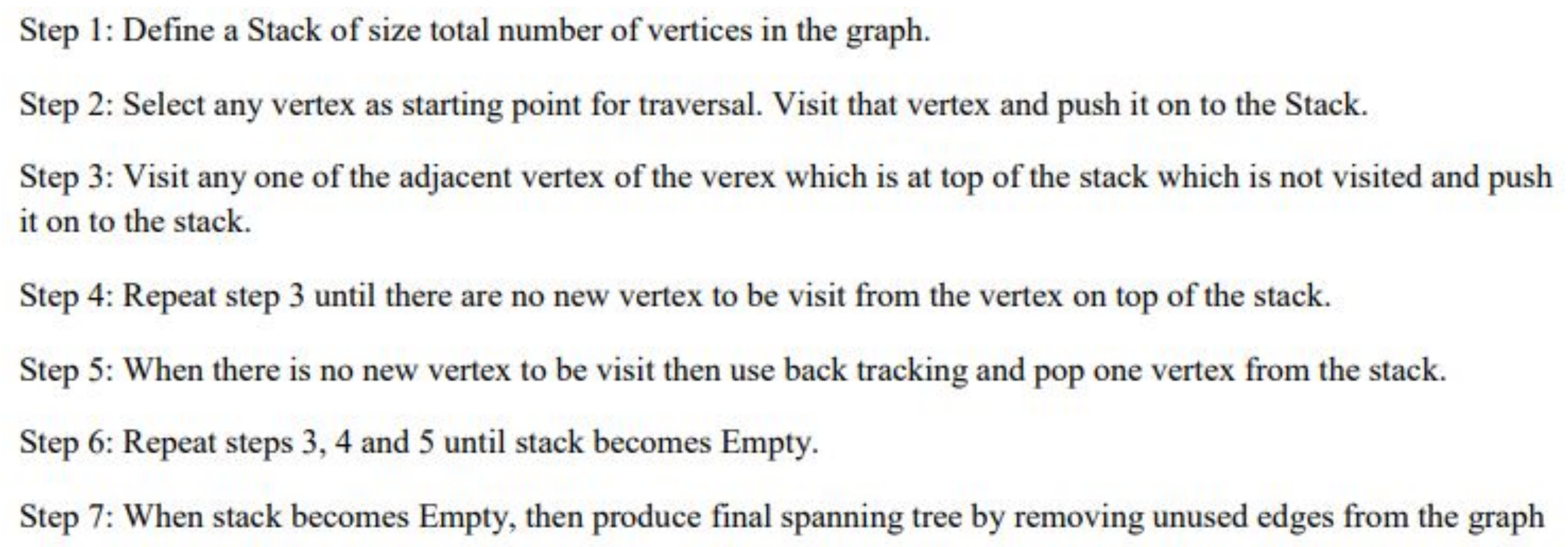
Graph Traversals

- There are two possible traversal methods for a graph:
  1. Depth-First Traversal
  2. Breadth-First Traversal

Depth First Traversal

- Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.
- The data structure which is being used in DFS is stack. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Algorithm



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 adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

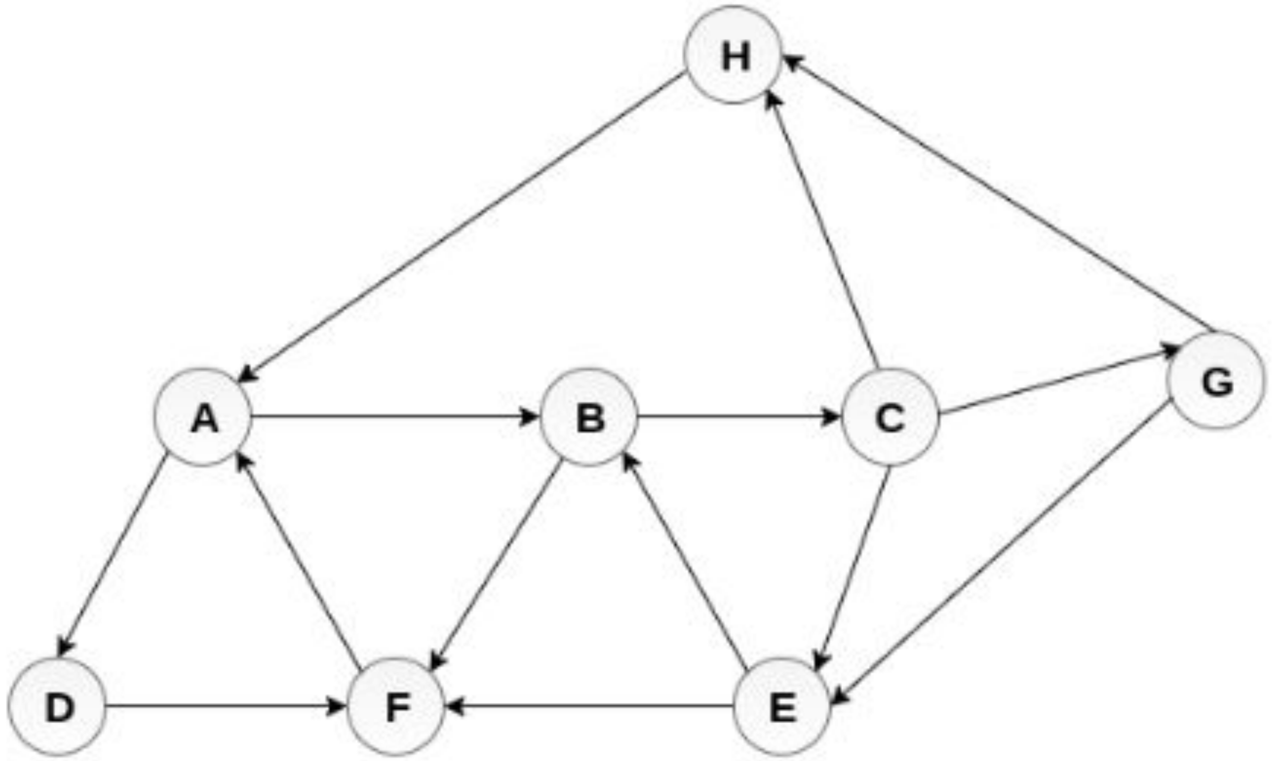
Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

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

Example :

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



Adjacency Lists

A : B, D

B : C, F

C : E, G, H

G : E, H

E : B, F

F : A

D : F

H : A

Solution:

Push H onto the stack

STACK : H

POP the top element of the stack i.e. H, print it and push all the neighbors of H onto the stack that are is ready state.

Print H

STACK : A

Pop the top element of the stack i.e. A, print it and push all the neighbors of A onto the stack that are in ready state.

Print A

Stack : B, D

Pop the top element of the stack i.e. D, print it and push all the neighbors of D onto the stack that are in ready state.

Print D

Stack : B, F

Pop the top element of the stack i.e. F, print it and push all the neighbors of F onto the stack that are in ready state.

Print F

Stack : B

Pop the top of the stack i.e. B and push all the neighbors

Print B

Stack : C

Pop the top of the stack i.e. C and push all the neighbours.

Print C

Stack : E, G

Pop the top of the stack i.e. G and push all its neighbours.

Print G

Stack : E

Pop the top of the stack i.e. E and push all its neighbours.

Print E

Stack :

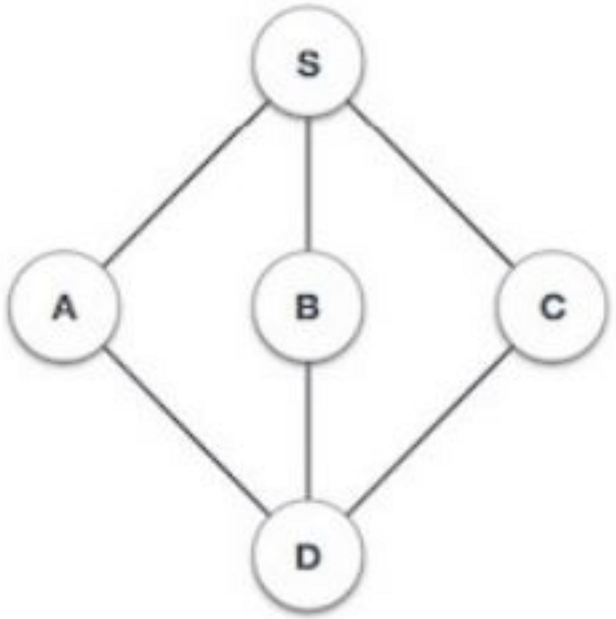
- Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

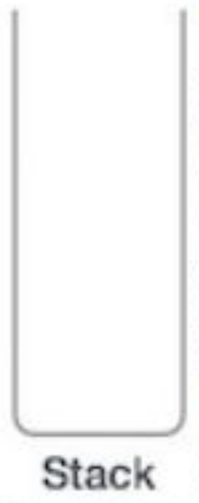
The printing sequence of the graph will be :

$H \rightarrow A \rightarrow D \rightarrow F \rightarrow B \rightarrow C \rightarrow G \rightarrow E$

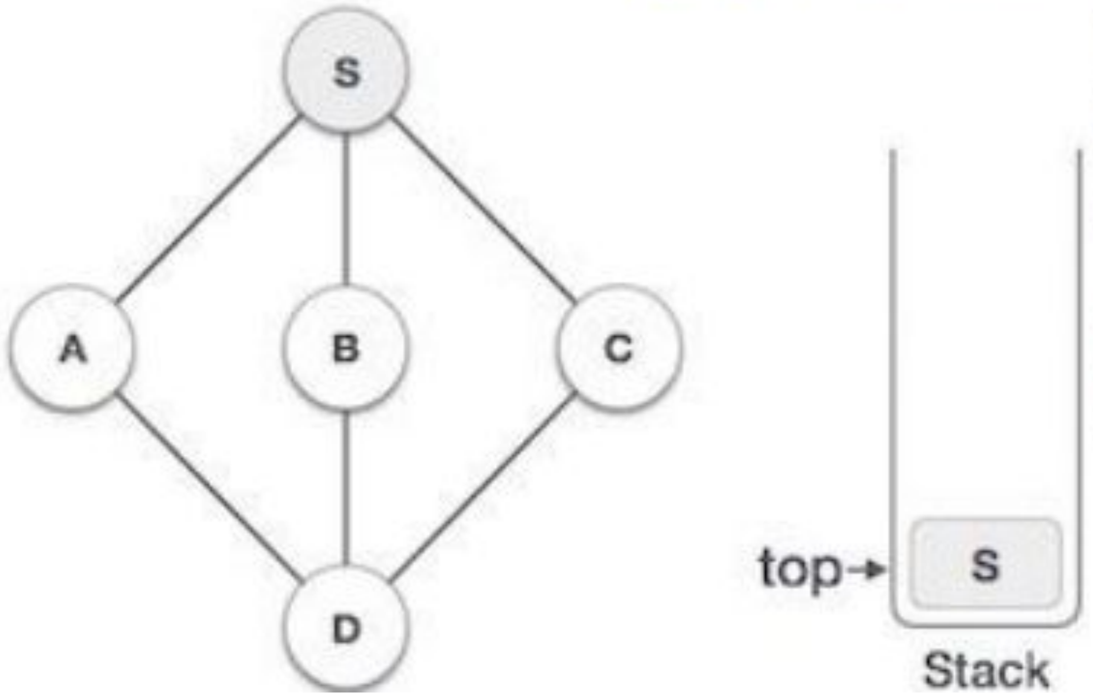
DFS Example

1. Initialize the stack



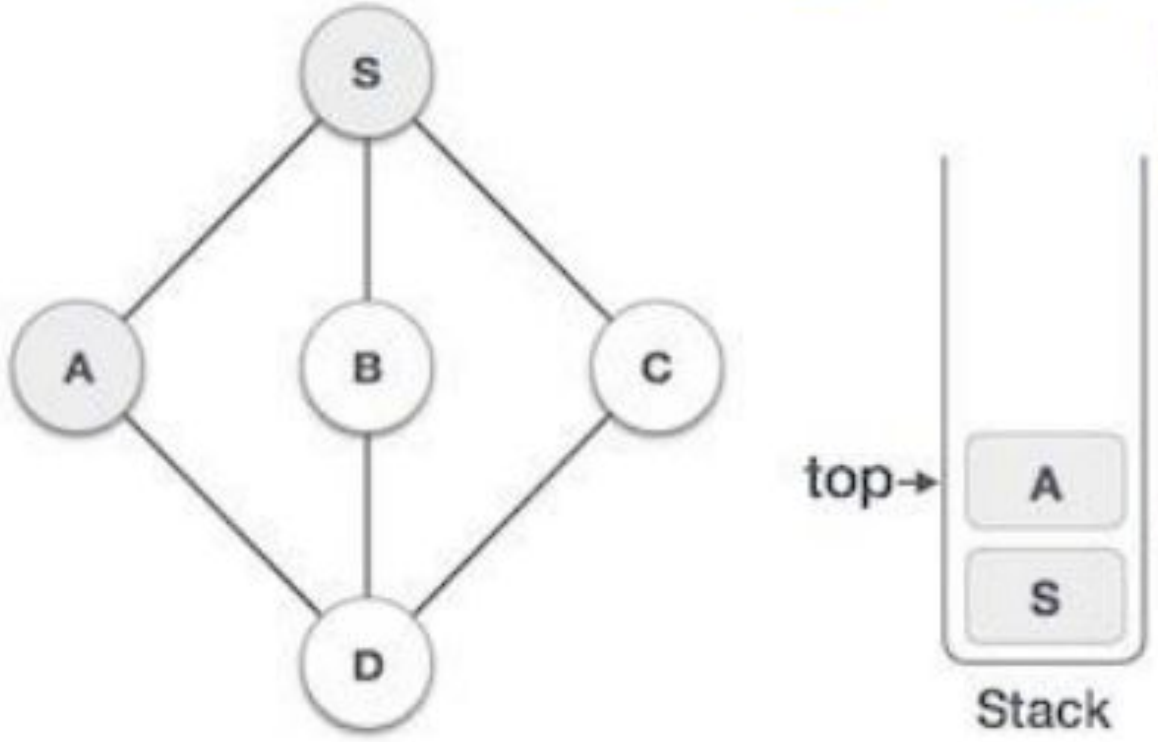


2. Mark S as visited and put it onto the stack.
   - Explore any unvisited adjacent node from S.
   - We have three nodes and we can pick any of them.
     - *For this example, we shall take the node in alphabetical order.*



Output - S

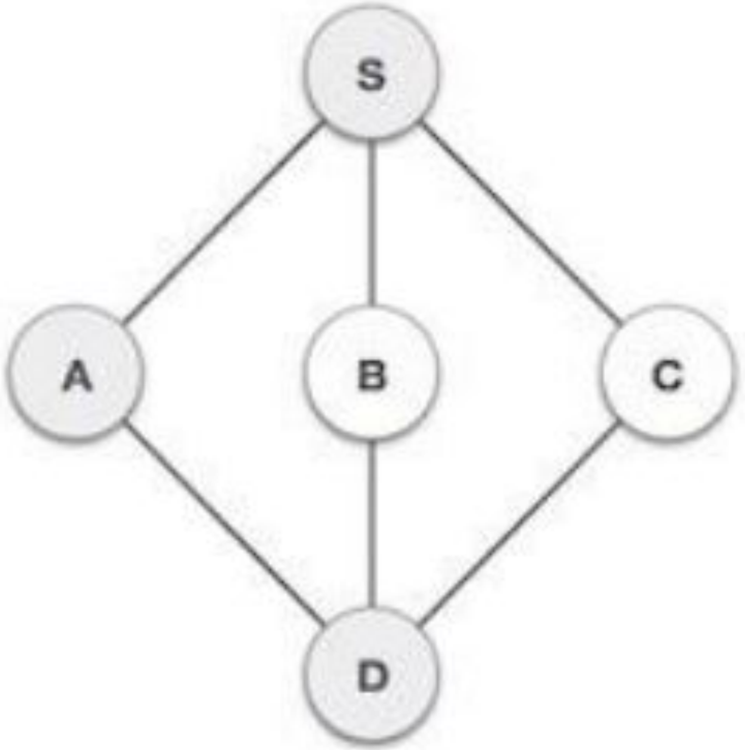
3. Mark A as visited and put it onto the stack.
- Explore any unvisited adjacent node from A.
- Both S and D are adjacent to A but we are concerned for unvisited nodes only.

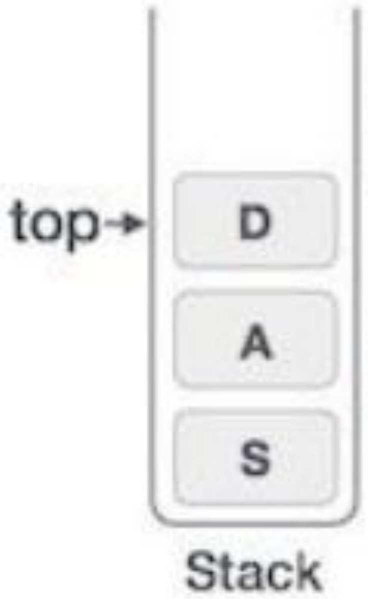


Output – S-A

4. Visit D and mark it visited and put onto the stack.

- Here we have B and C nodes which are adjacent to D and both are unvisited.
- *But we shall again choose in alphabetical order.*

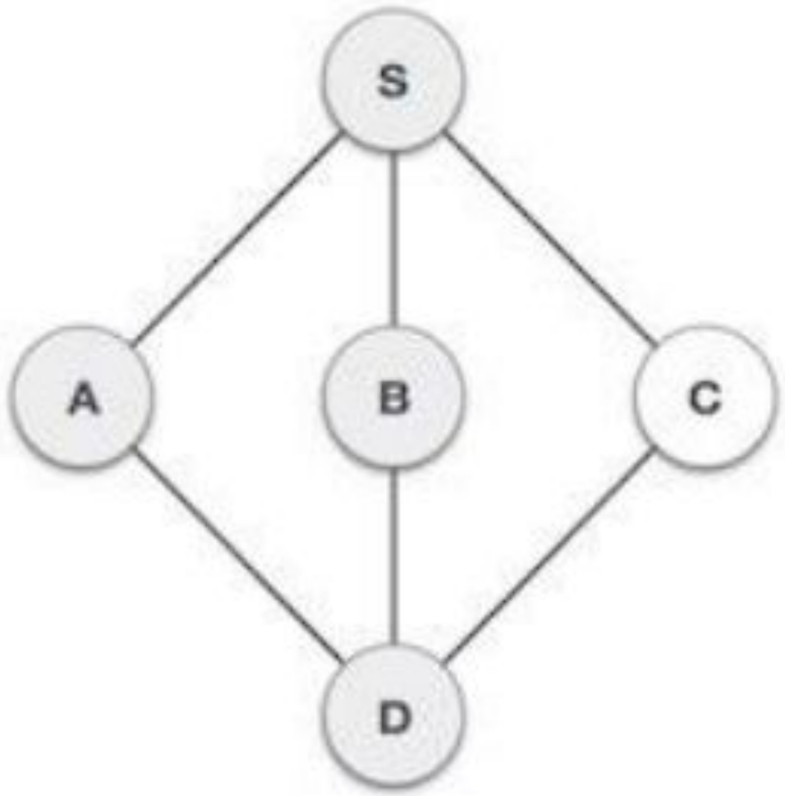


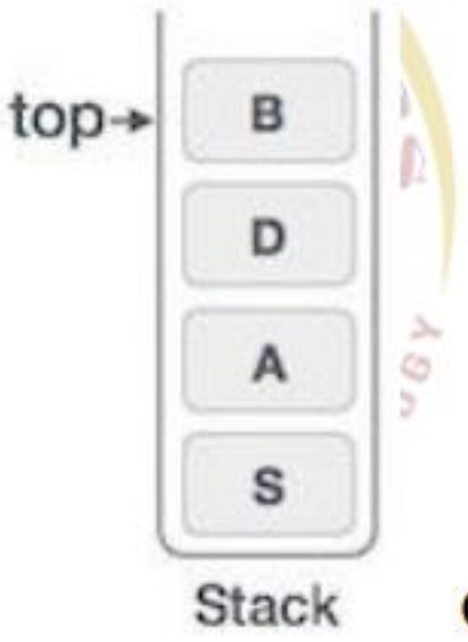


Output – S-A-D

5. We choose B, mark it visited and put onto stack.

- Here B does not have any unvisited adjacent node. So we pop B from the stack.

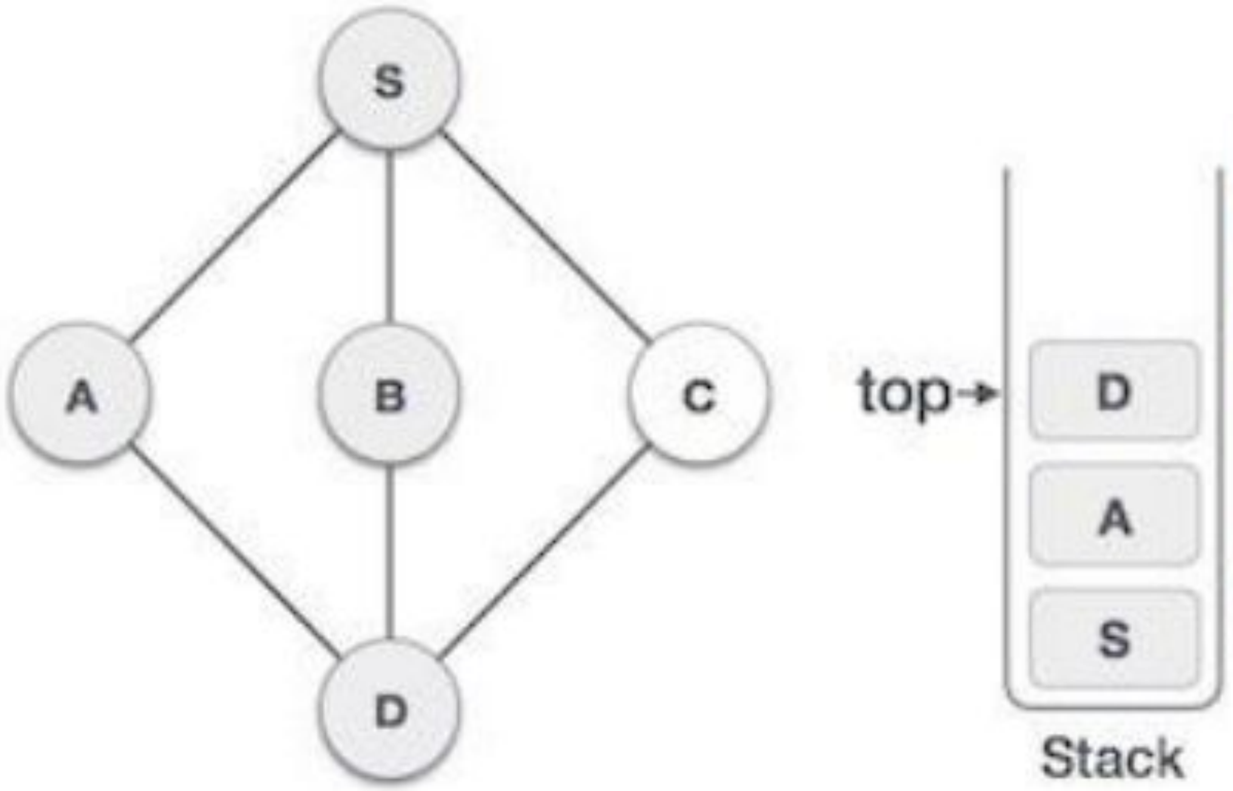




Output – S-A-D-B

6. We check stack top for return to previous node and check if it has any unvisited nodes.

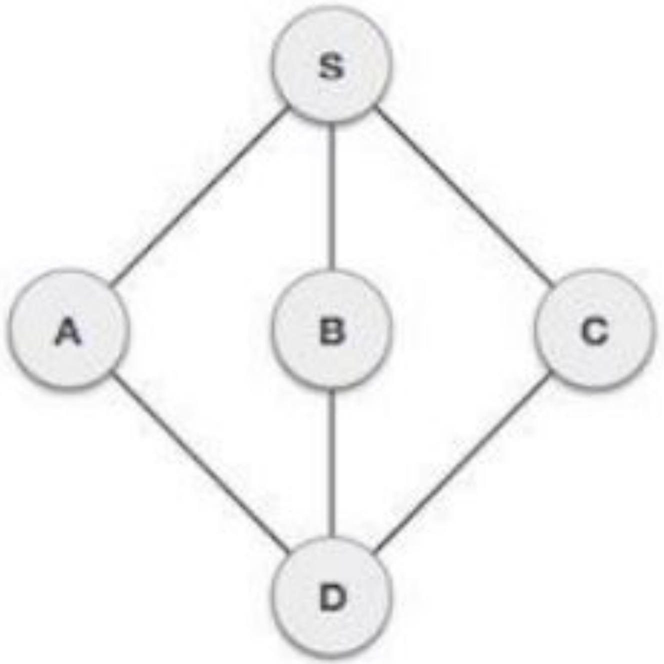
- *Here, we find D to be on the top of stack.*

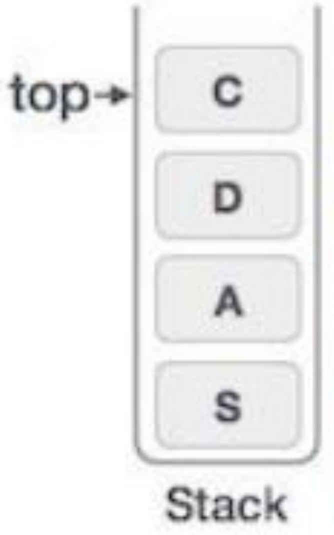


Output – S-A-D-B

7. Only unvisited adjacent node is from D is C now.

So we visit C, mark it visited and put it onto the stack.





Output – S-A-D-B-C

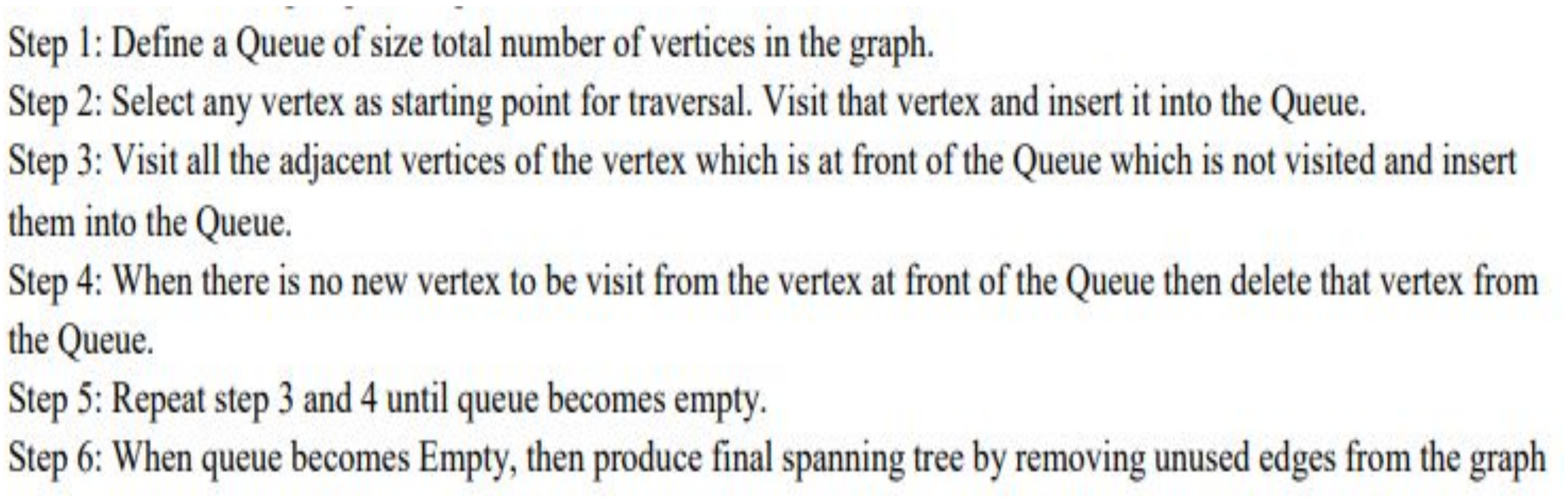
8. As C does not have any unvisited adjacent node so we keep popping the stack until we find a node which has unvisited adjacent node.

- *In this case, there's none and we keep popping until stack is empty.*

Breadth First Traversal

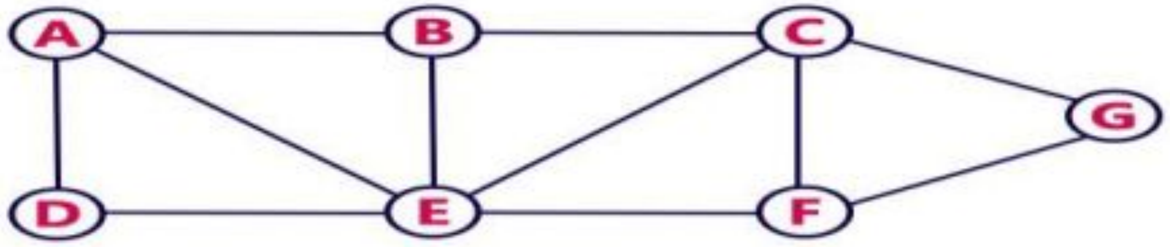
- Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighboring nodes.
- Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

Algorithm



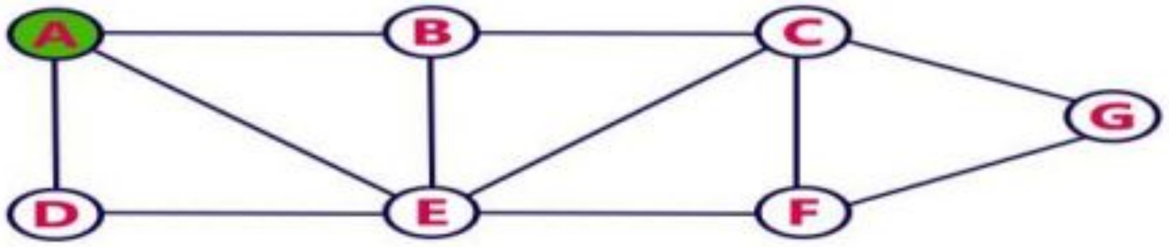
Step 1: Define a Queue of size total number of vertices in the graph.
Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
Step 3: Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.
Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
Step 5: Repeat step 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

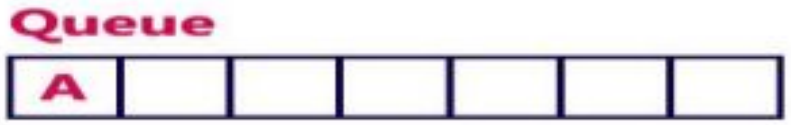
Consider the following example graph to perform BFS traversal



Step 1:

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

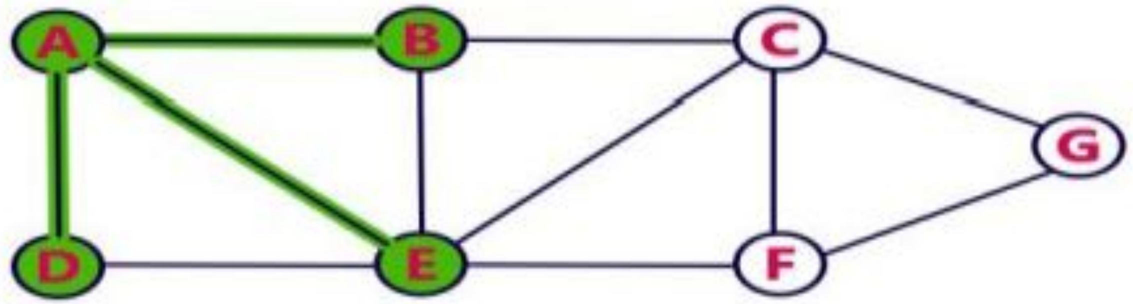


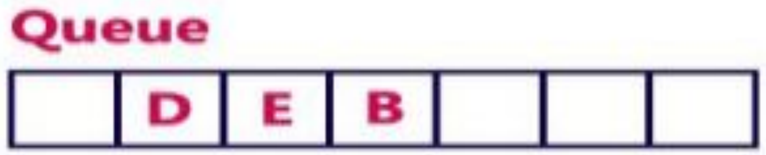


Visiting Array: A

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..

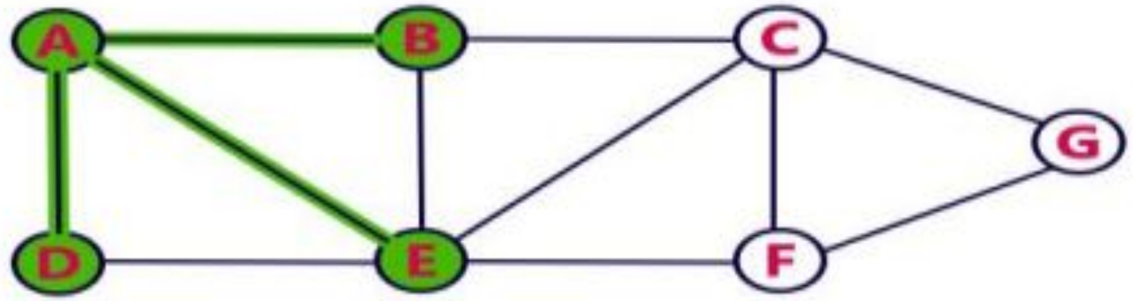


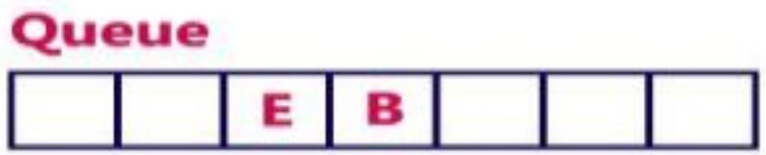


Visiting Array: A , D

Step 3:

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

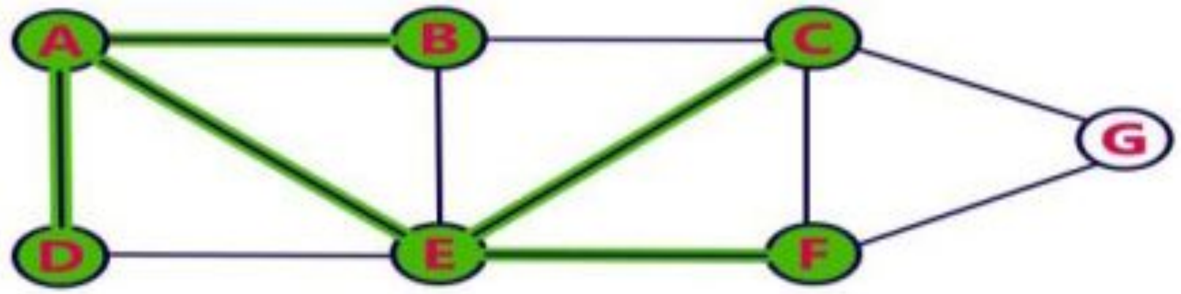




Visiting Array: A , D , E

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.



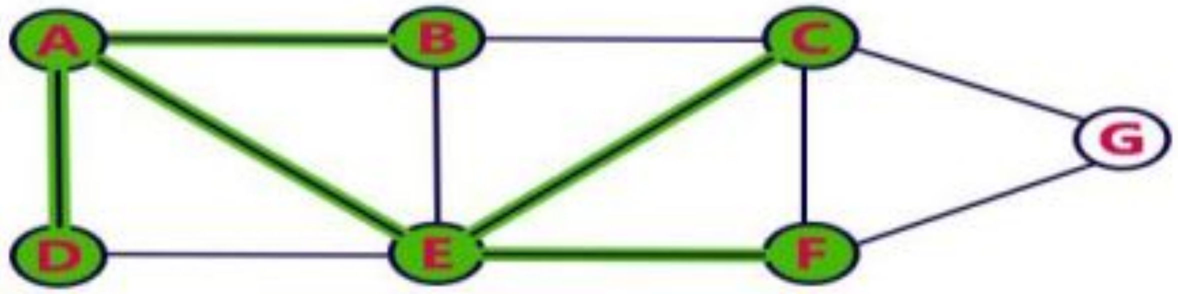
Queue

			B	C	F	

Visiting Array: A , D , E , B

Step 5:

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



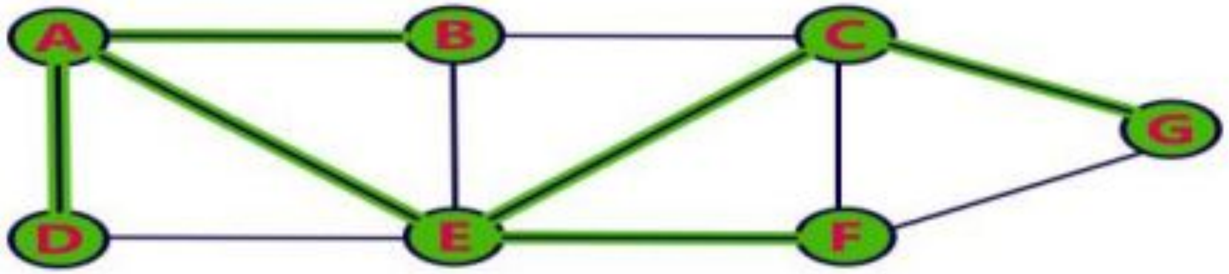
Queue

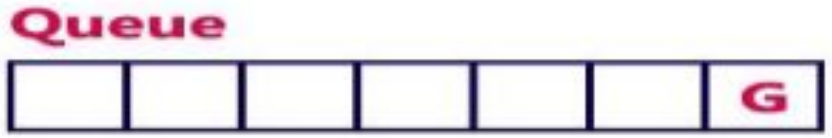
				C	F	

Visiting Array: A , D , E , B, C

Step 7:

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

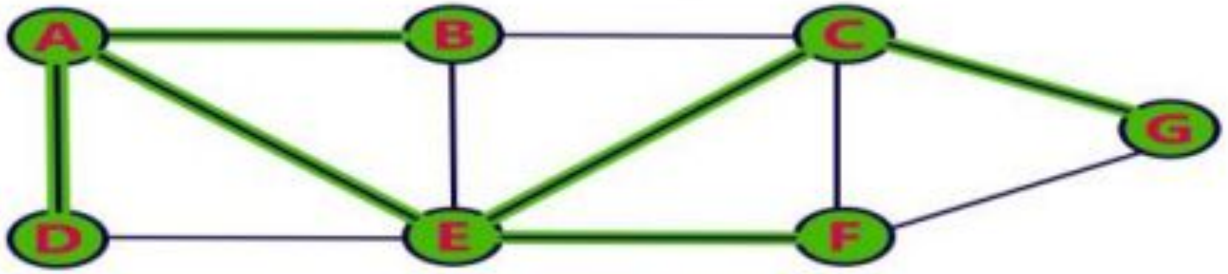


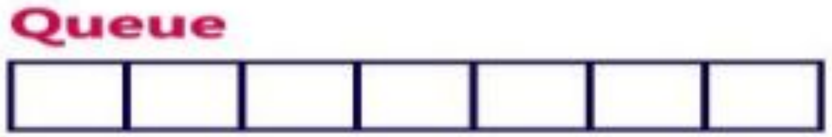


Visiting Array: A , D , E , B, C ,F

Step 8:

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





Visiting Array: A , D , E , B, C ,F , G

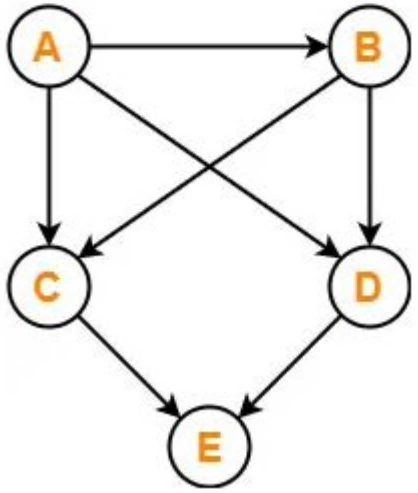
Topological Sorting

- Topological Sort is a linear ordering of the vertices in such a way that

if there is an edge in the DAG going from vertex ‘u’ to vertex ‘v’, then ‘u’ comes before ‘v’ in the ordering.

- It is important to note that-
  - Topological Sorting is possible if and only if the graph is a **Directed Acyclic Graph**.
  - There may exist multiple different topological orderings for a given directed acyclic graph.

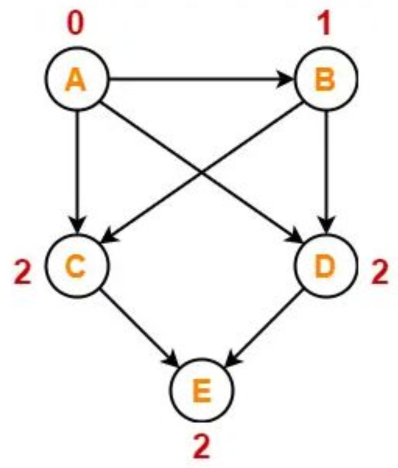
Find the number of different topological orderings possible for the given graph-



The topological orderings of the above graph are found in the following steps-

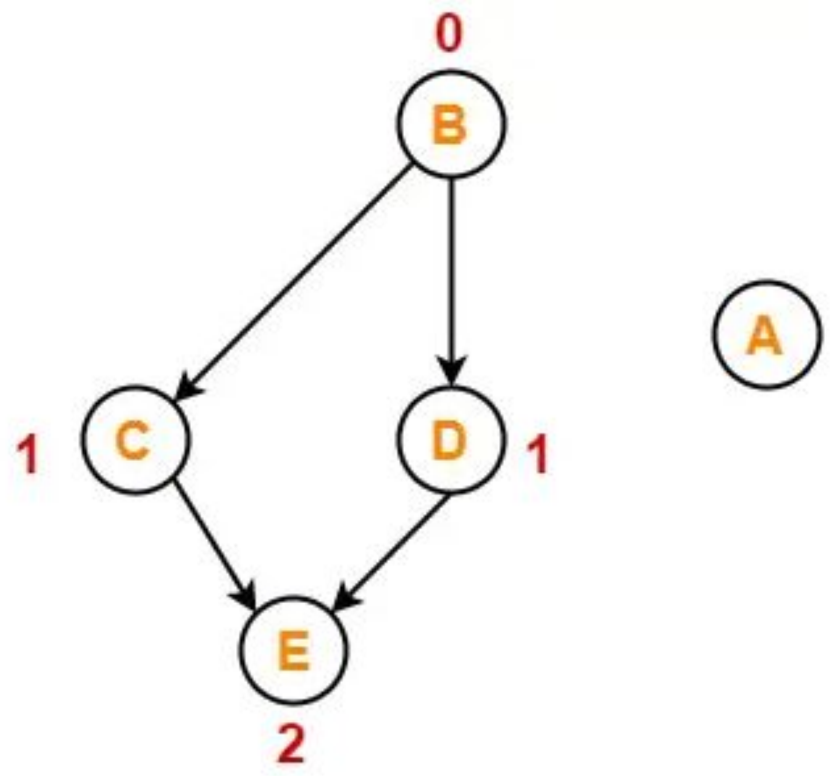
Step-01:

Write in-degree of each vertex-



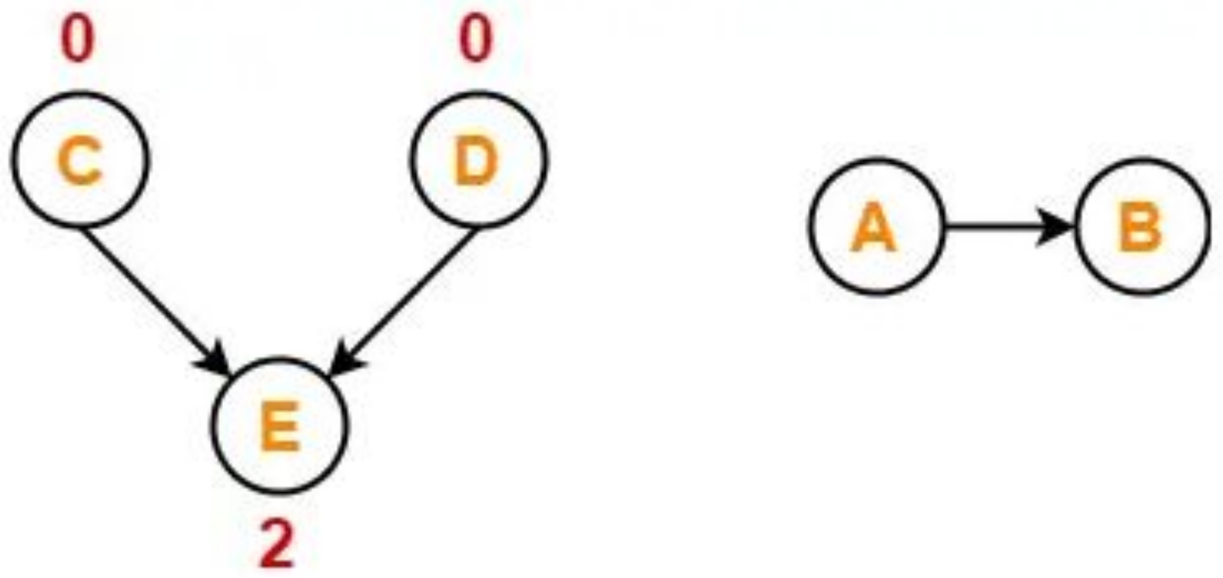
Step-02:

- Vertex-A has the least in-degree.
- So, remove vertex-A and its associated edges.
- Now, update the in-degree of other vertices.



Step-03:

- Vertex-B has the least in-degree.
- So, remove vertex-B and its associated edges.
- Now, update the in-degree of other vertices.



Step-04:

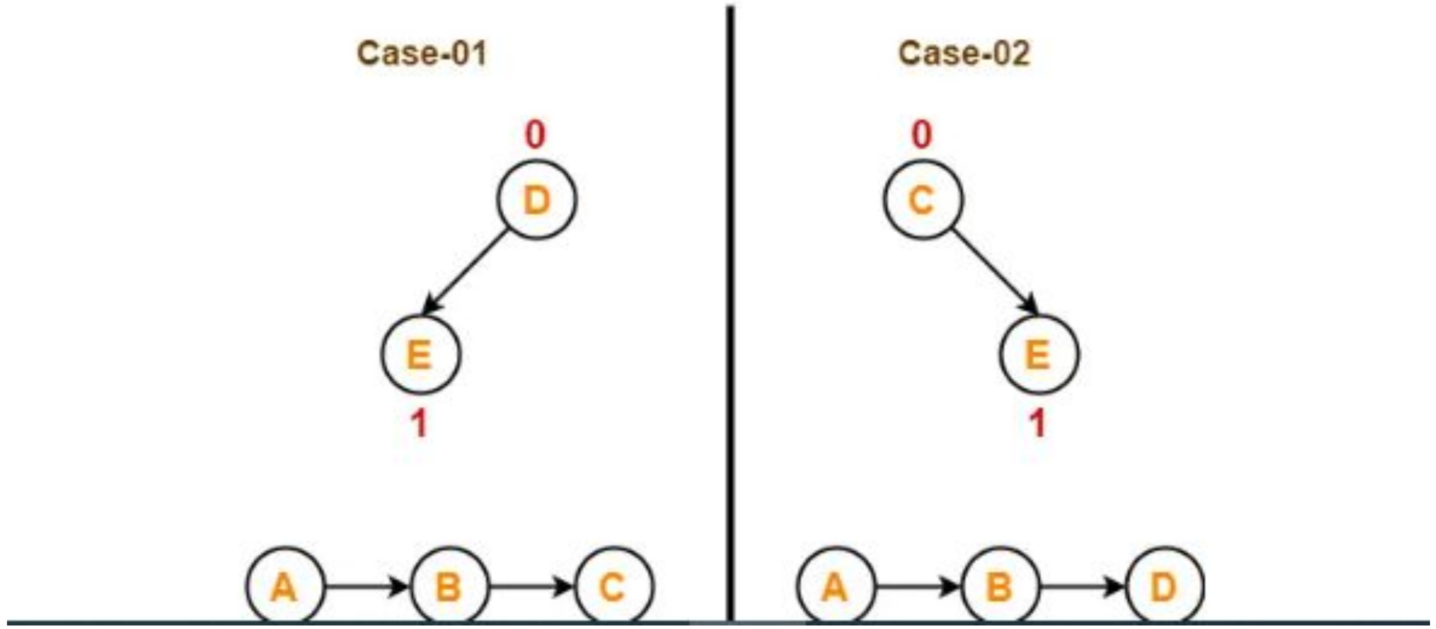
There are two vertices with the least in-degree. So, following 2 cases are possible-

In case-01,

- Remove vertex-C and its associated edges.
- Then, update the in-degree of other vertices.

In case-02,

- Remove vertex-D and its associated edges.
- Then, update the in-degree of other vertices.



Step-05:

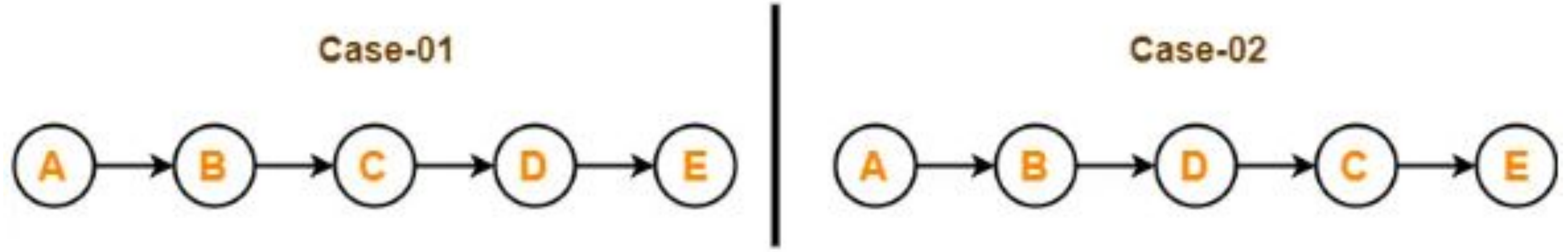
Now, the above two cases are continued separately in the similar manner.

In case-01,

- Remove vertex-D since it has the least in-degree.
- Then, remove the remaining vertex-E.

In case-02:

•Remove vertex-C since it has the least in-degree.
•Then, remove the remaining vertex-E.



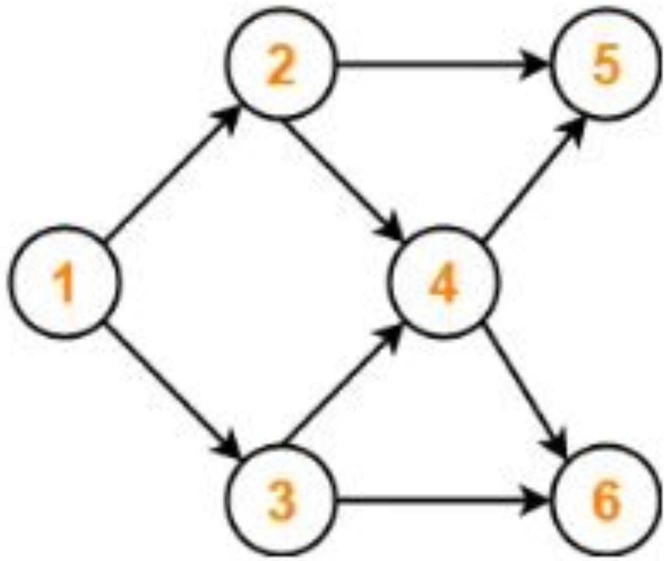
For the given graph, following **2** different topological orderings are possible-

- **A B C D E**
- **A B D C E**

Topological Sort :

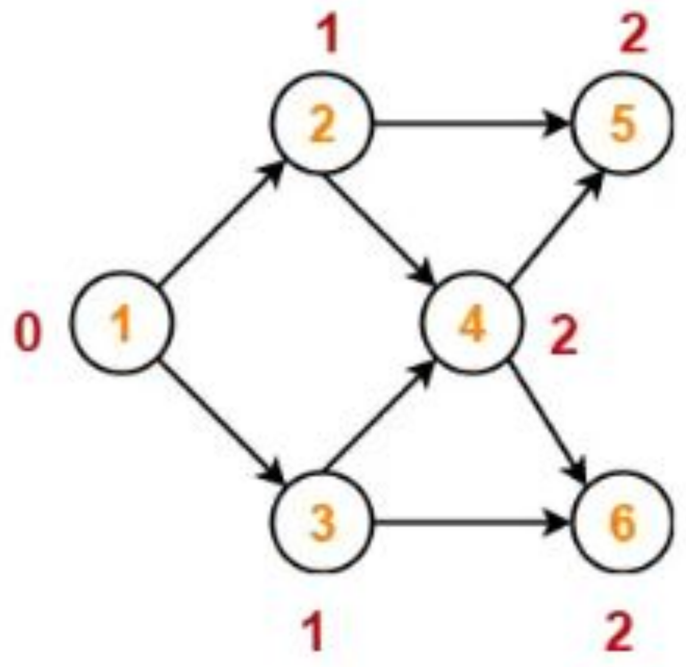
Problem-02:

Find the number of different topological orderings possible for the given graph-



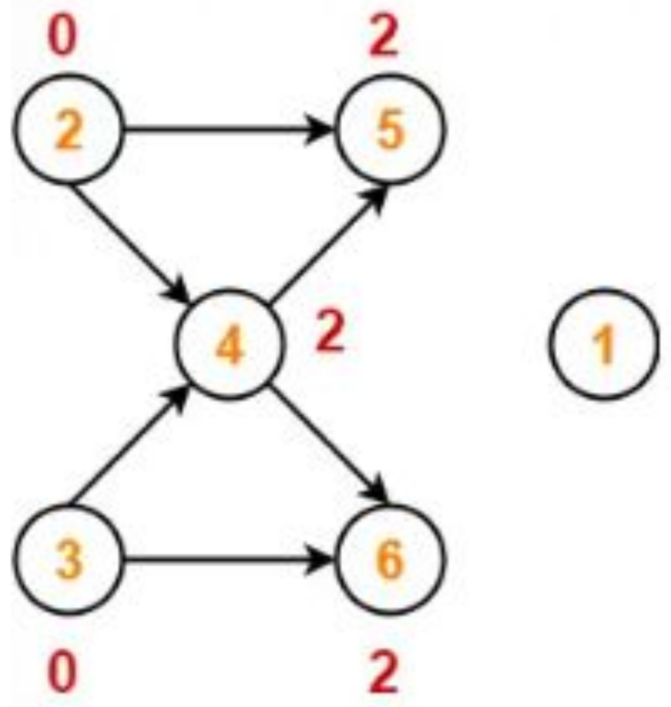
Step-01:

Write in-degree of each vertex-



Step-02:

- Vertex-1 has the least in-degree.
- So, remove vertex-1 and its associated edges.
- Now, update the in-degree of other vertices.



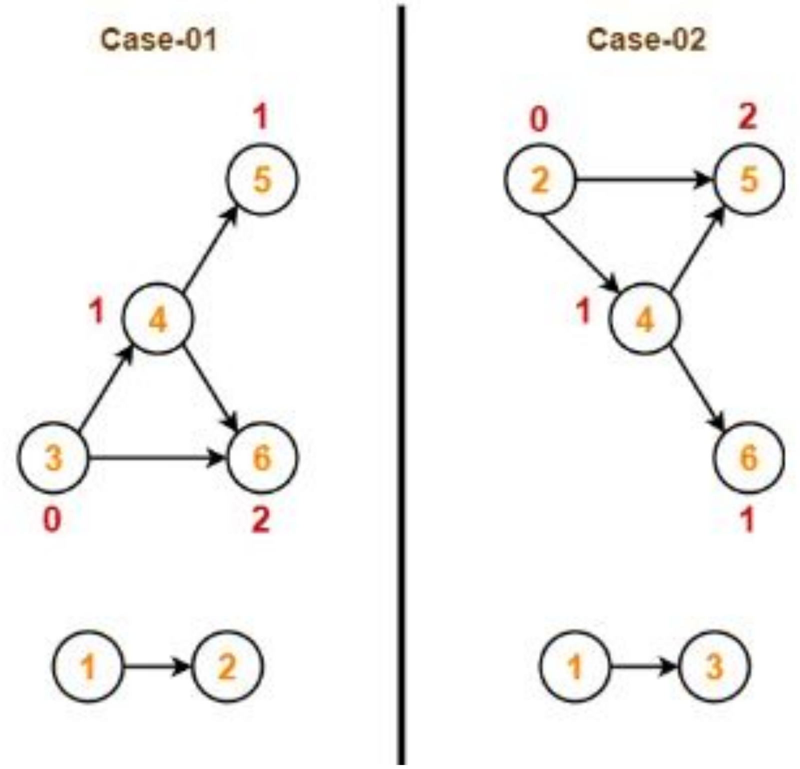
There are two vertices with the least in-degree. So, following 2 cases are possible-

In case-01,

- Remove vertex-2 and its associated edges.
- Then, update the in-degree of other vertices.

In case-02,

- Remove vertex-3 and its associated edges.
- Then, update the in-degree of other vertices.

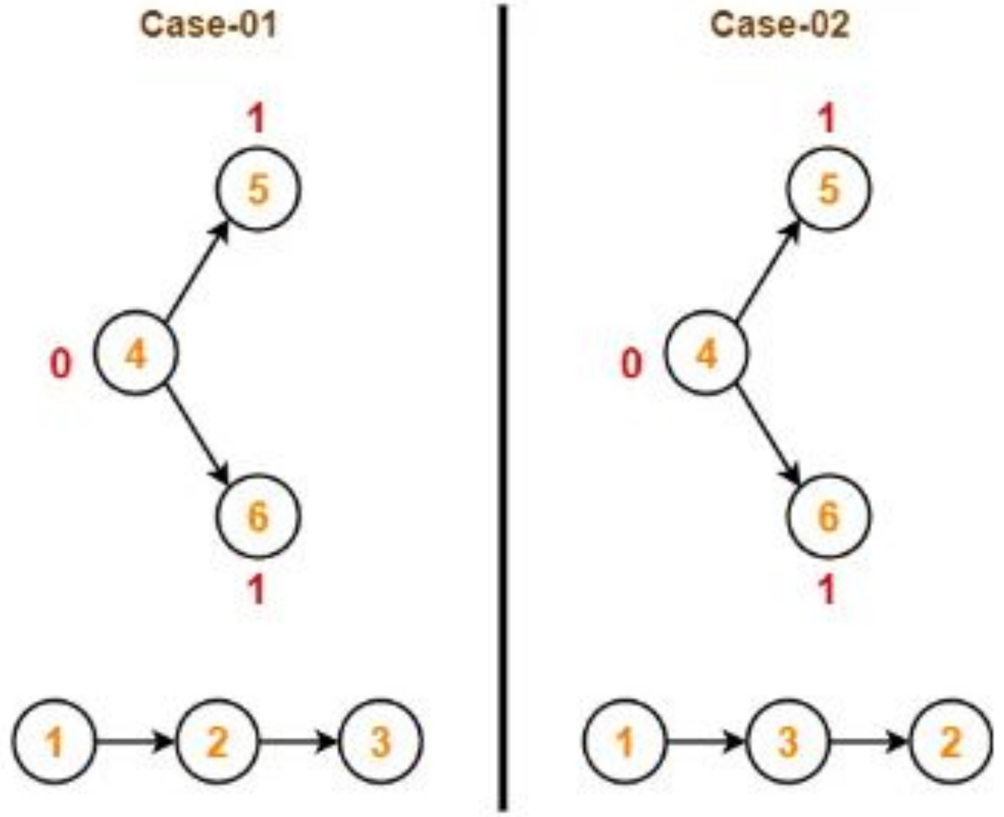


In case-01,

- Remove vertex-3 since it has the least in-degree.
- Then, update the in-degree of other vertices.

In case-02,

- Remove vertex-2 since it has the least in-degree.
- Then, update the in-degree of other vertices.

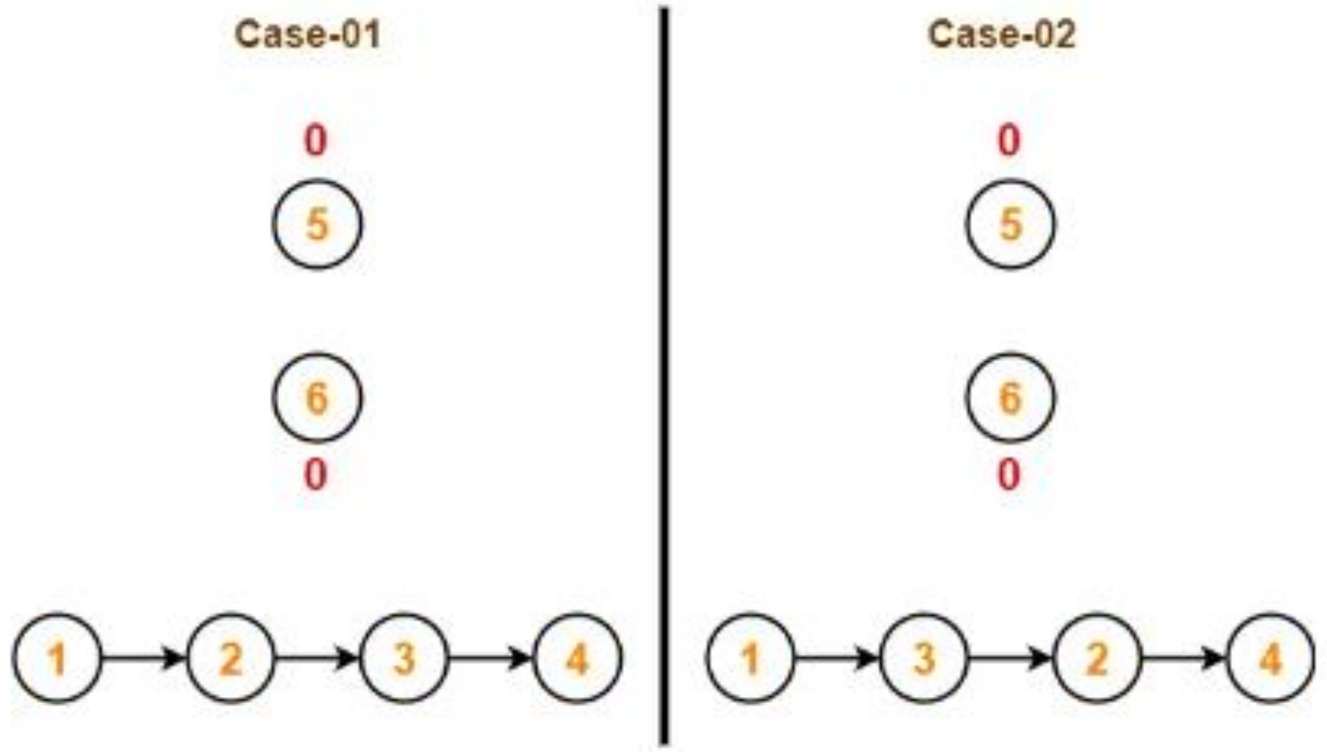


In case-01,

- Remove vertex-4 since it has the least in-degree.
- Then, update the in-degree of other vertices.

In case-02,

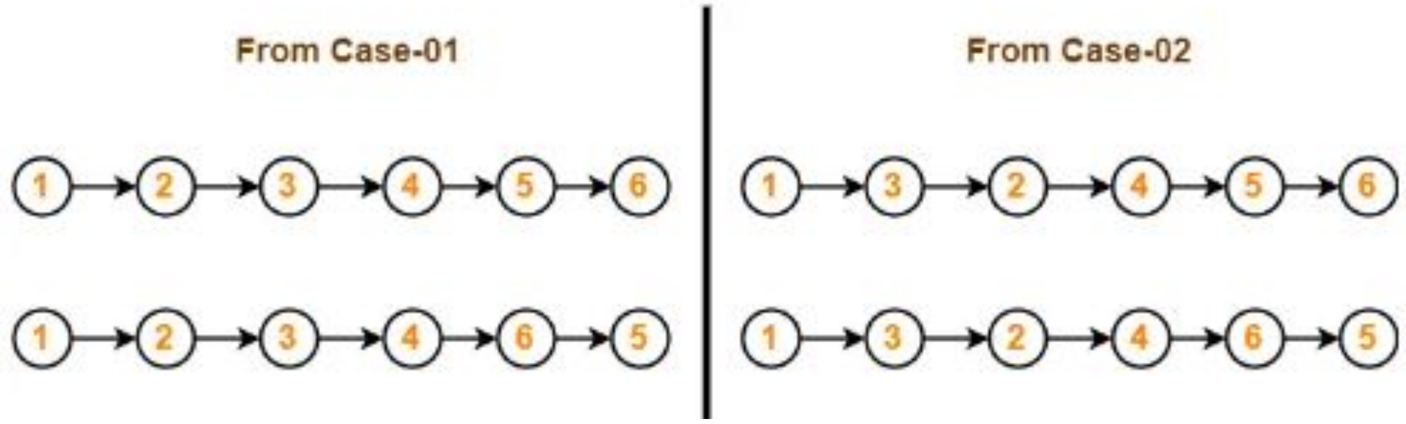
- Remove vertex-4 since it has the least in-degree.
- Then, update the in-degree of other vertices.



In case-01,

- There are 2 vertices with the least in-degree.
- So, 2 cases are possible.
- Any of the two vertices may be taken first.

Same is with case-02.



$$\begin{tikzcd} 1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[r] & 5 \arrow[r] & 6 \\ 1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[r] & 6 \arrow[r] & 5 \end{tikzcd}$$

$$\begin{tikzcd} 1 \arrow[r] & 3 \arrow[r] & 2 \arrow[r] & 4 \arrow[r] & 5 \arrow[r] & 6 \\ 1 \arrow[r] & 3 \arrow[r] & 2 \arrow[r] & 4 \arrow[r] & 6 \arrow[r] & 5 \end{tikzcd}$$

For the given graph, following **4** different topological orderings are possible-

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