Graph Traversal

- Graph traversal is a technique used for searching a vertex in a graph.
- The graph traversal is also used to decide the order of vertices is visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops.

- There are two graph traversal techniques and they are as follows...
- **DFS (Depth First Search)**
- **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

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

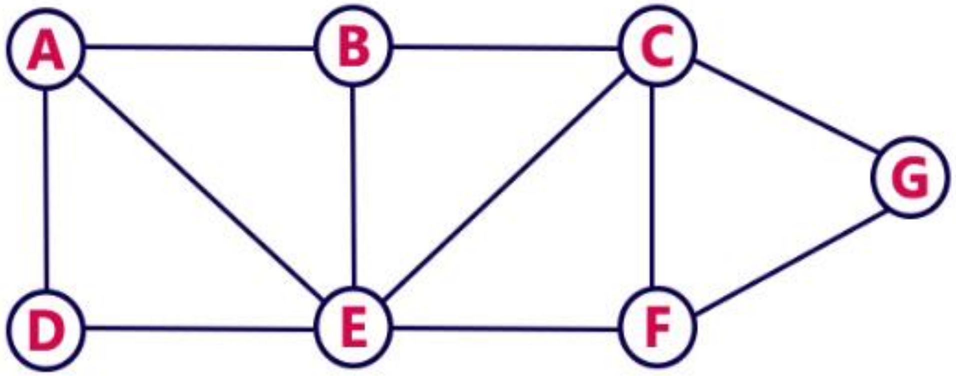
- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.

- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

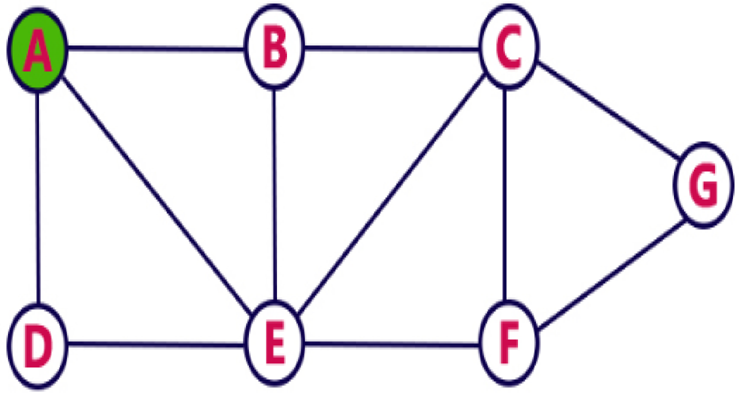
Example

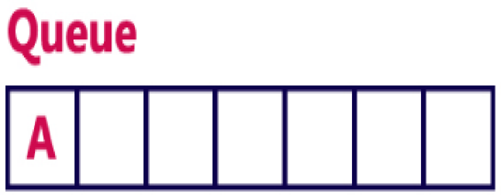
Consider the following example graph to perform BFS traversal



Step 1:

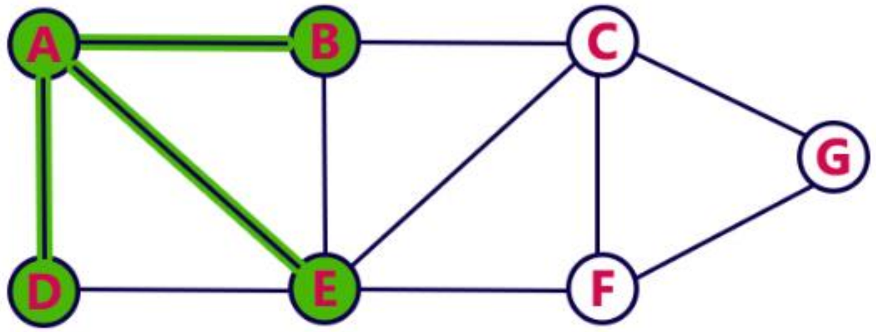
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

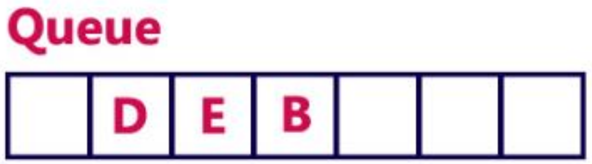




Step 2:

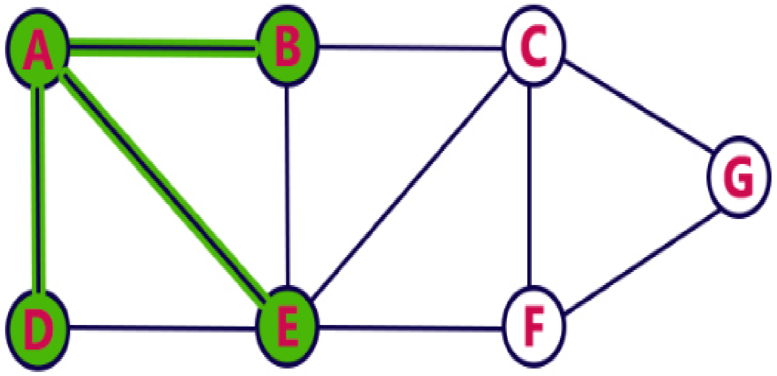
- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..





Step 3:

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

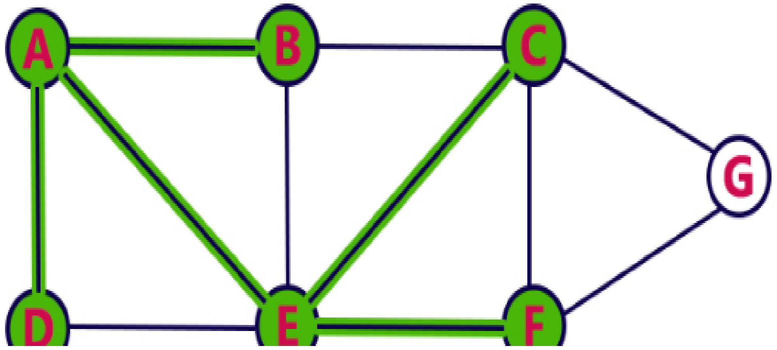


Queue

		E	B			

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.

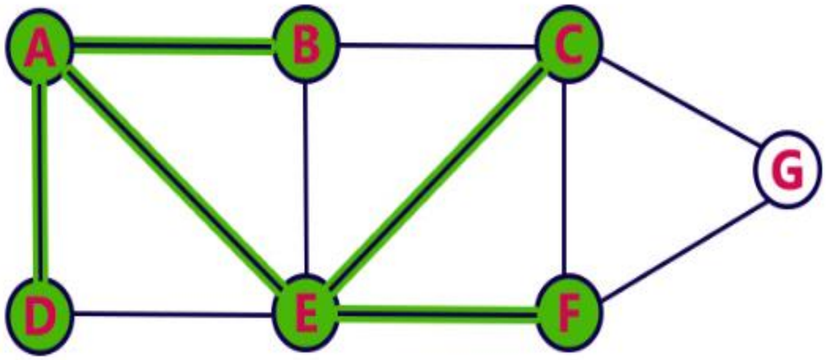


Queue

			B	C	F	

Step 5:

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

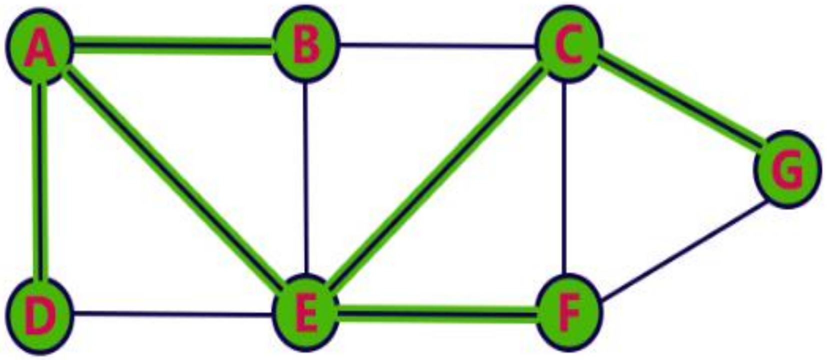


Queue

				C	F	

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.

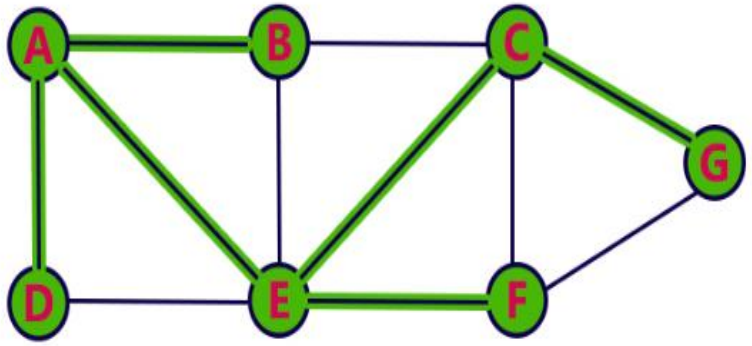


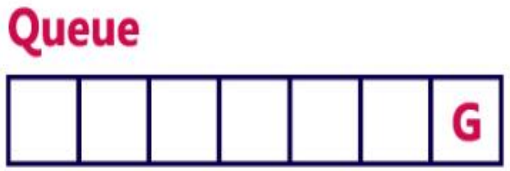
Queue

					F	G

Step 7:

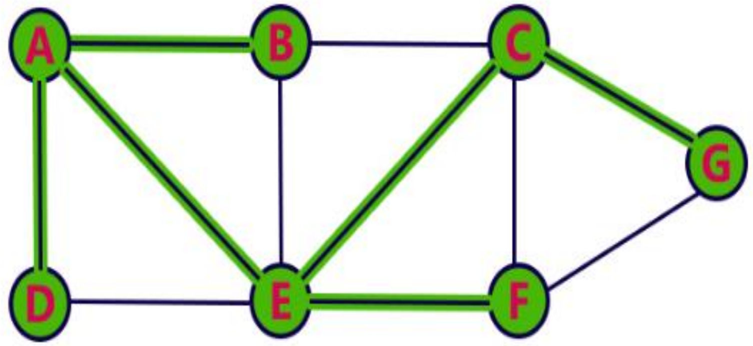
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

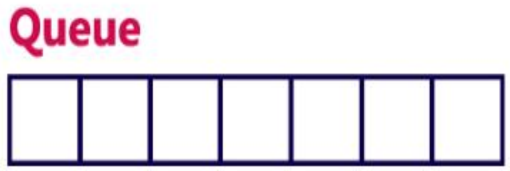




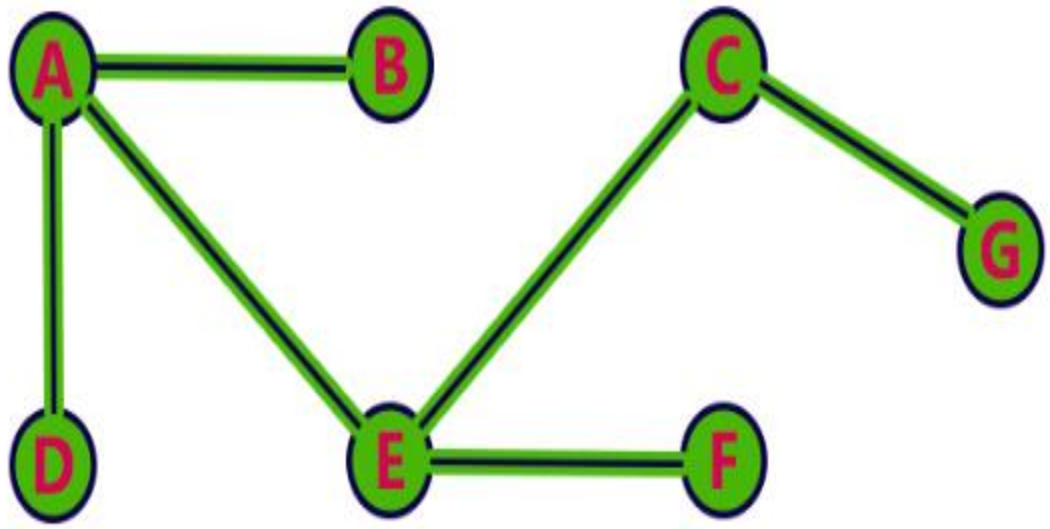
Step 8:

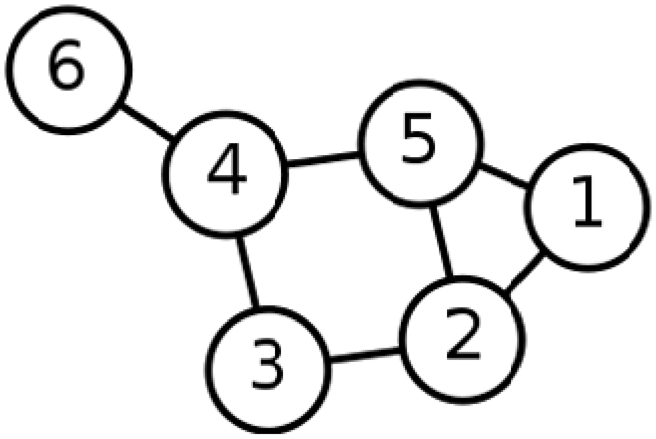
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



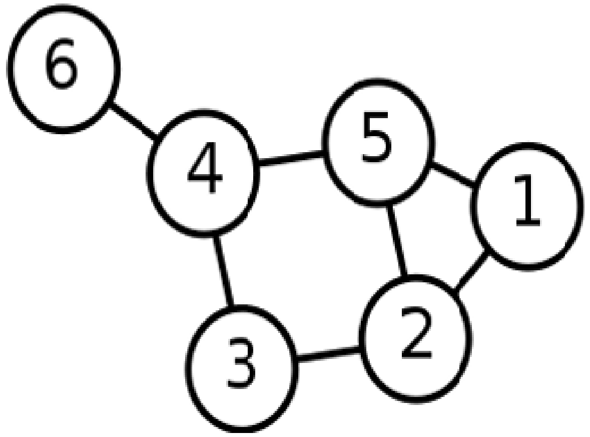


- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...




6
4
5
1
3
2

Action	Current Node	Queue	Unvisited Nodes	Visited Nodes
Start with node 1		1	2, 3, 4, 5, 6	1
Dequeue node 1	1		2, 3, 4, 5, 6	1
Node 1 has unvisited children nodes 2 and 5	1		2, 3, 4, 5, 6	1
Mark 2 as visited and enqueue into queue	1	2	3, 4, 5, 6	1, 2
Mark 5 as visited and enqueue into queue	1	5, 2	3, 4, 6	1, 2, 5
Node 1 has no more unvisited children, dequeue a new current node 2	2	5	3, 4, 6	1, 2, 5
Mark 3 as visited and enqueue into queue	2	3, 5	4, 6	1, 2, 5, 3
Node 2 has no more unvisited children, dequeue a new current node 5	5	3	4, 6	1, 2, 5, 3
Mark 4 as visited and enqueue into queue	5	4, 3	6	1, 2, 5, 3, 4
Node 5 has no more unvisited children, dequeue a new current node 3	3	4	6	1, 2, 5, 3, 4
Node 3 has no more unvisited children, dequeue a new current node 4	4		6	1, 2, 5, 3, 4
Mark 6 as visited and enqueue into queue	4	6		1, 2, 5, 3, 4, 6



There are no more unvisited nodes so the nodes will be dequeued from the queue and the algorithm will terminate.

BFS Applications

- **Shortest Path and MST for unweighted graph:** The path with the least number of edges is the **shortest path**. With the help of BFS, we can reach a vertex from a given source using the minimum number of edges.

- The crawlers in the search engine use the BFS approach. The idea used behind it is to start from the source page and repeat it.

- BFS is also helpful in navigation as it is easier to find the neighboring locations on maps.

- Due to better locality of reference, It is also used in Chenney’s algorithm in garbage collection.

- To find all the reachable nodes from a given node BFS is much beneficial.

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.

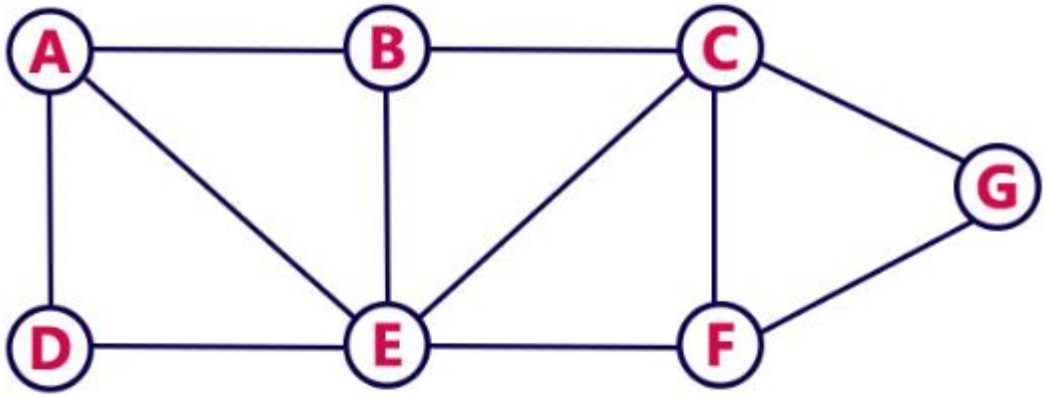
We use the following steps to implement DFS traversal...

- **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

---

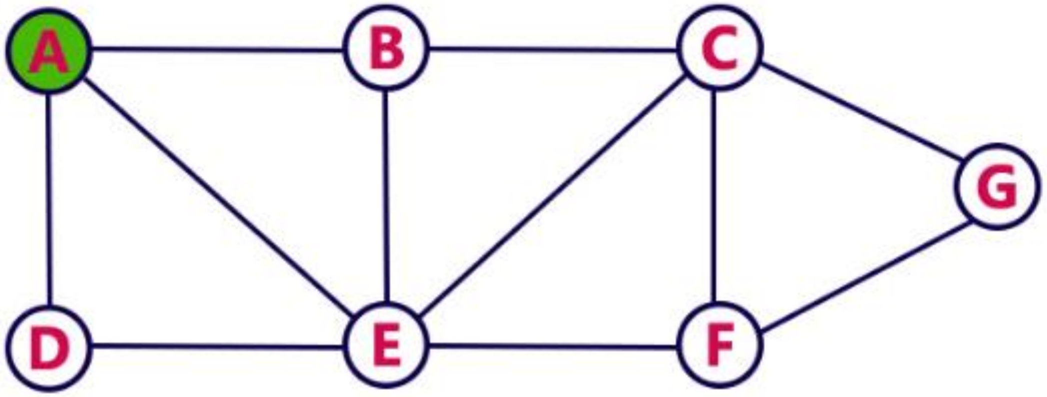
Back tracking is coming back to the vertex from which we reached the current vertex.

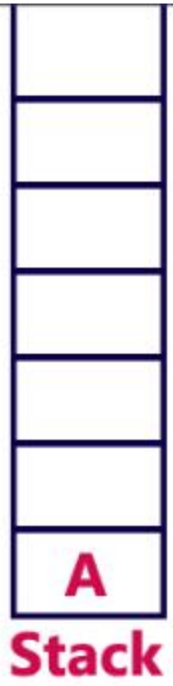
Consider the following example graph to perform DFS traversal



Step 1:

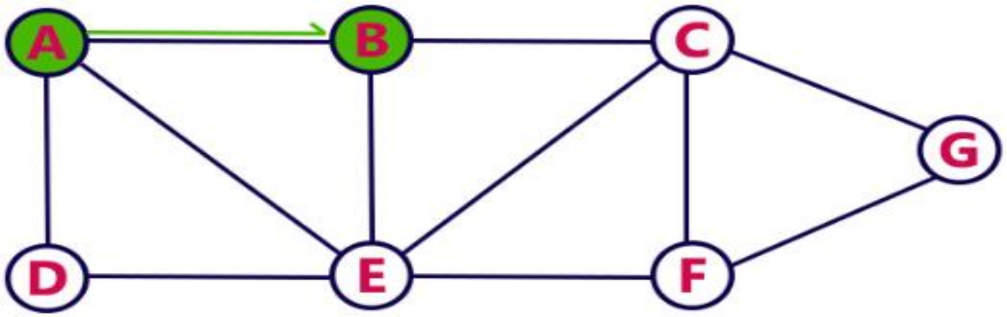
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

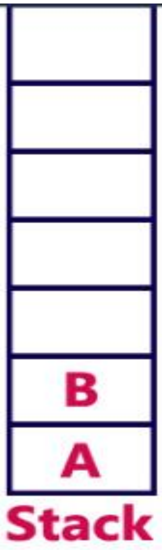




Step 2:

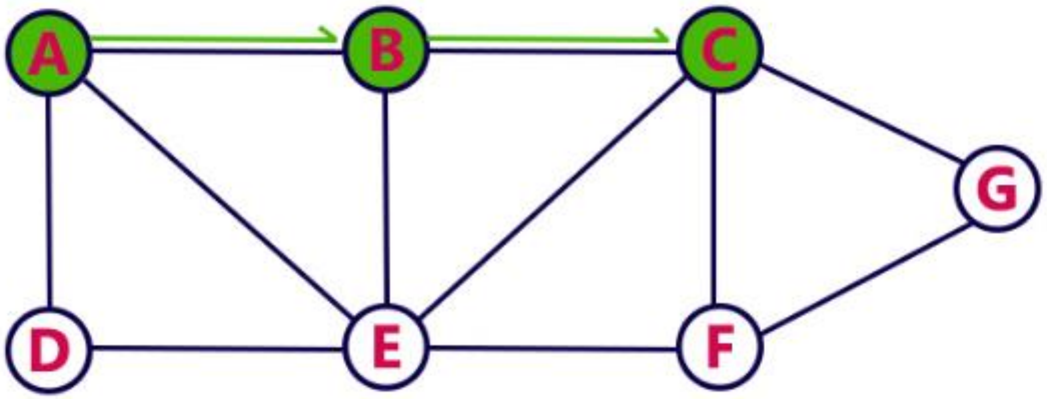
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

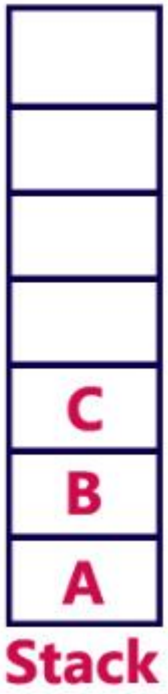




Step 3:

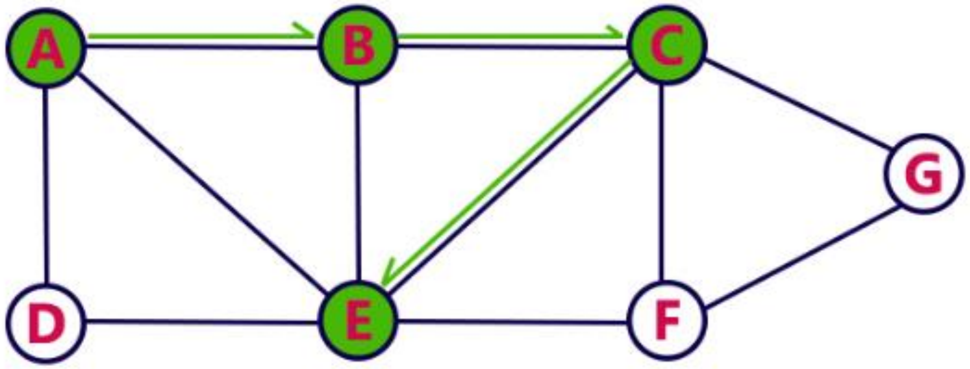
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

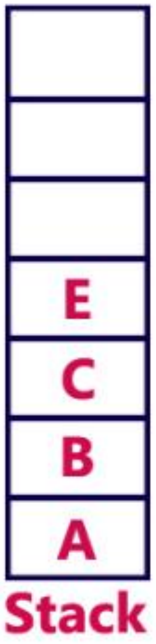




Step 4:

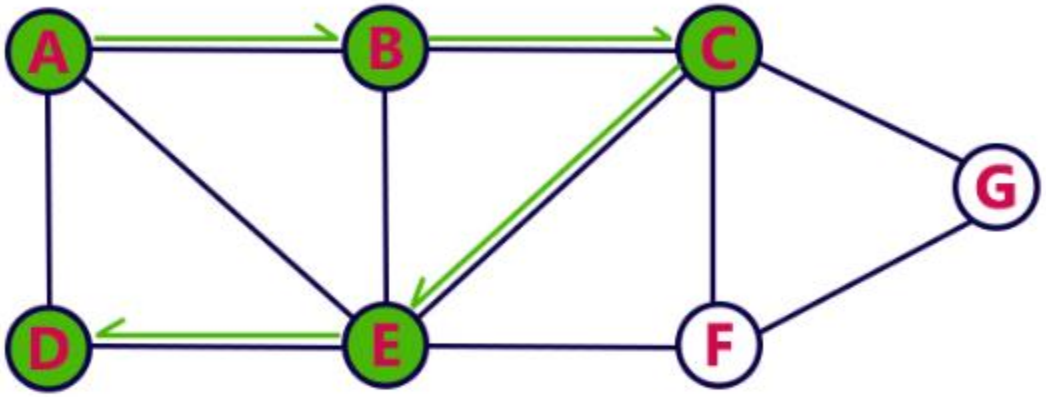
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

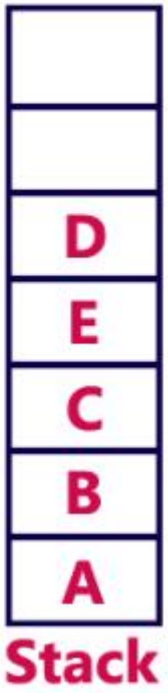




Step 5:

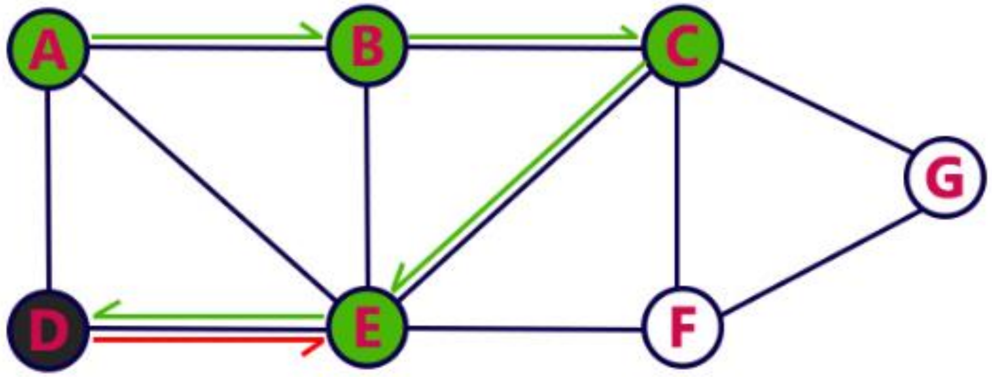
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

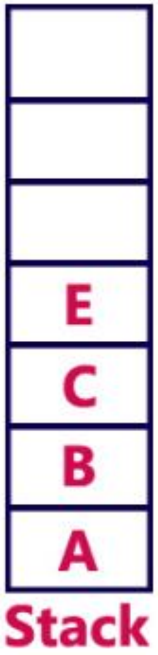




Step 6:

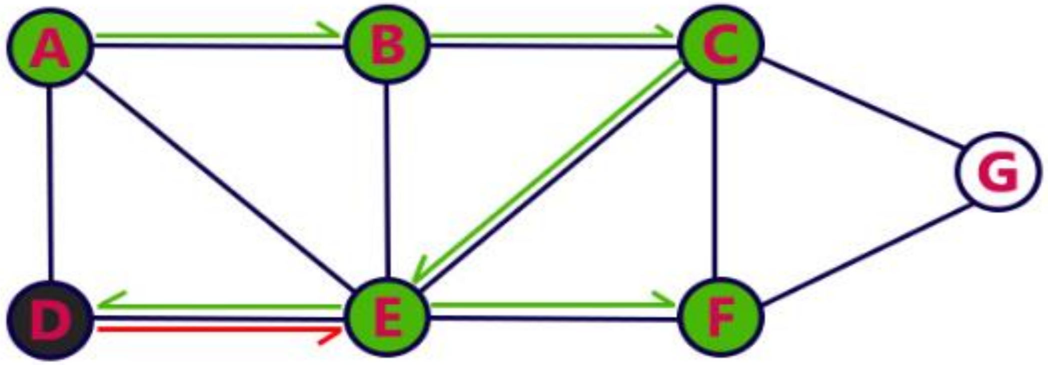
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

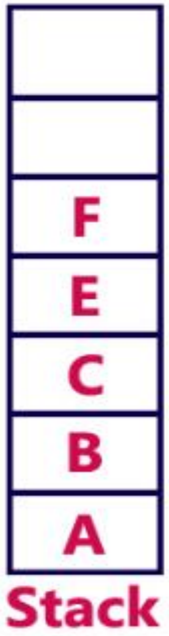




Step 7:

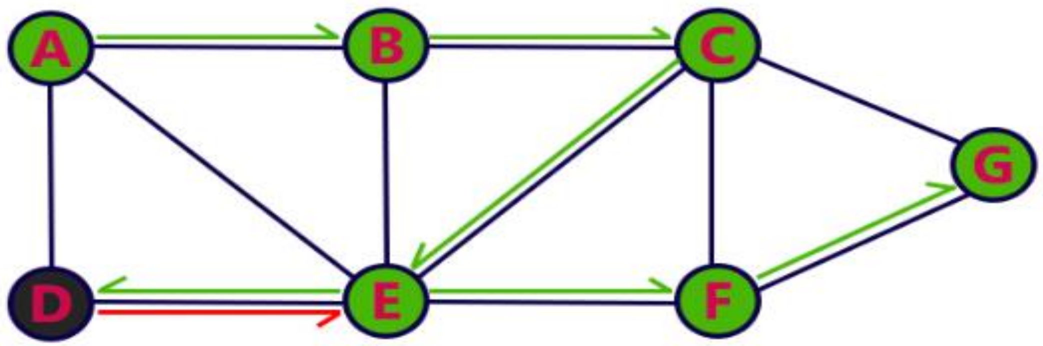
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

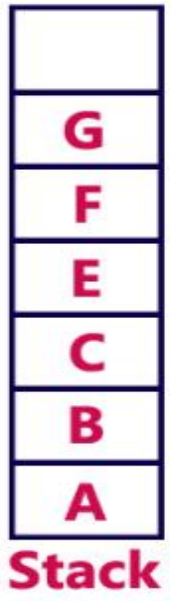




Step 8:

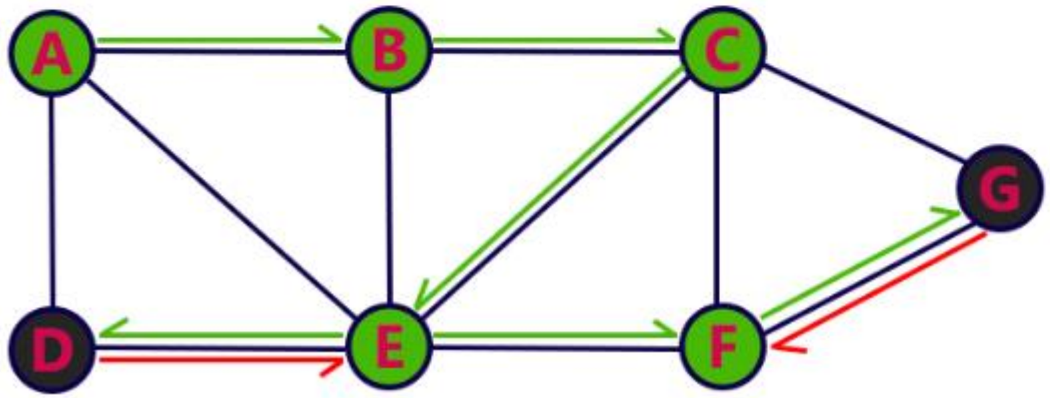
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

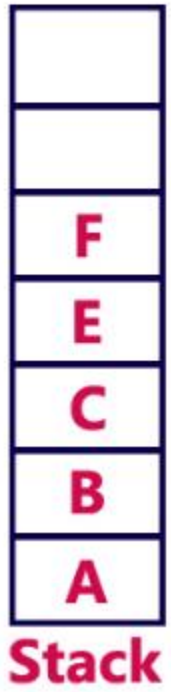




Step 9:

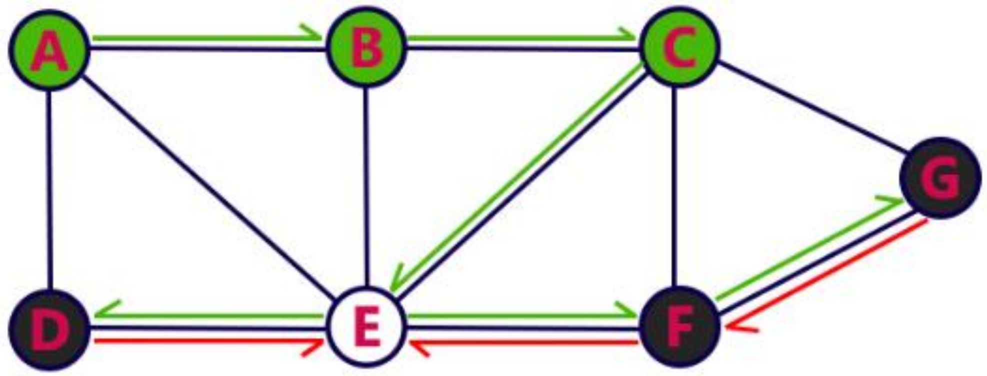
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

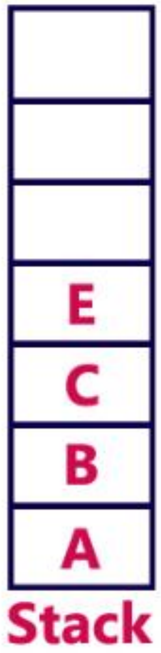




Step 10:

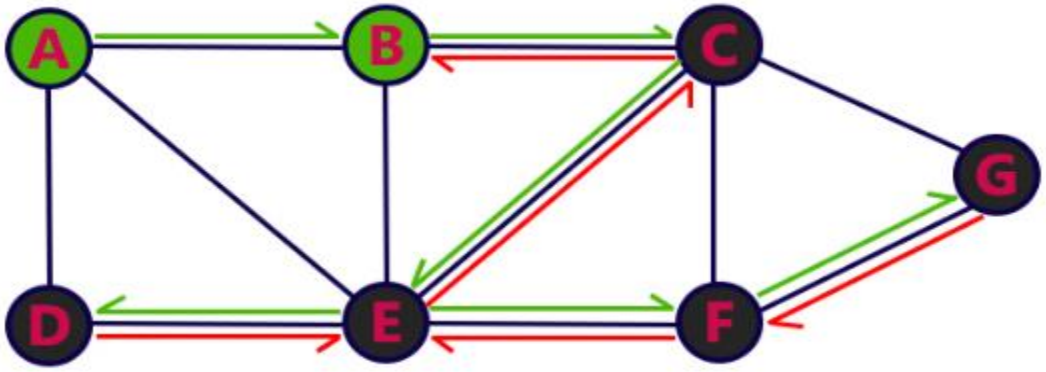
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

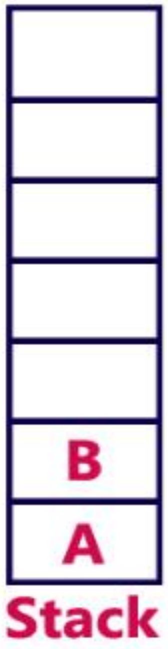




Step 12:

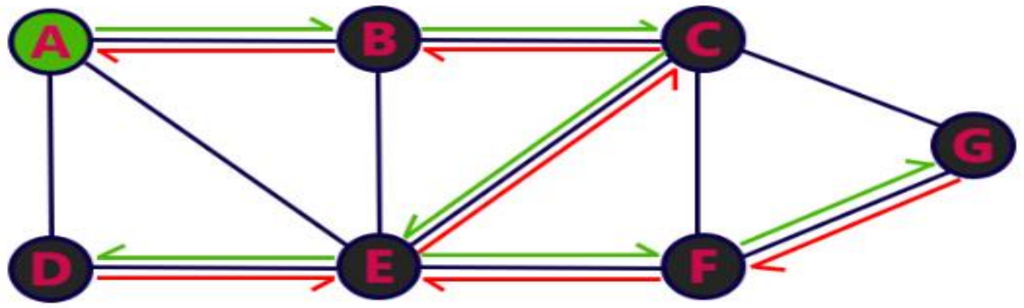
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

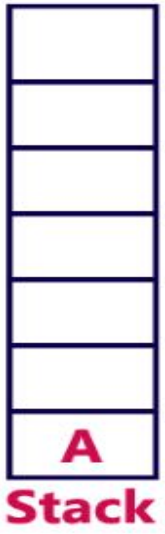




Step 13:

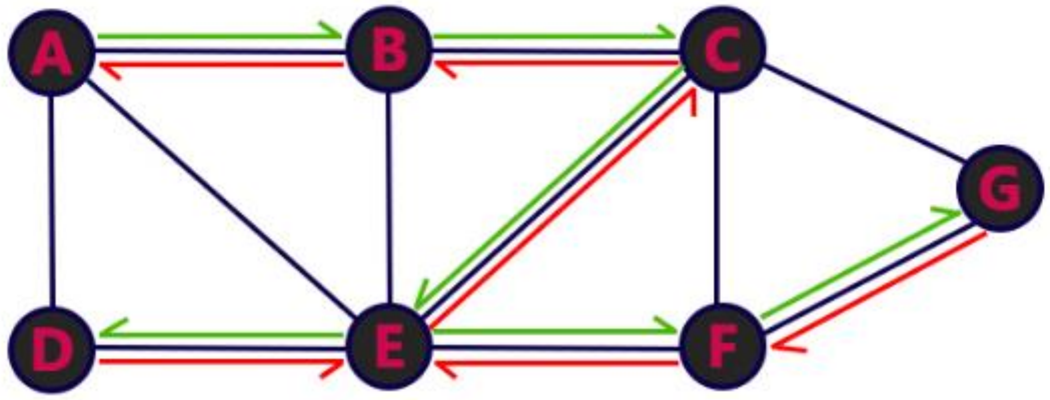
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

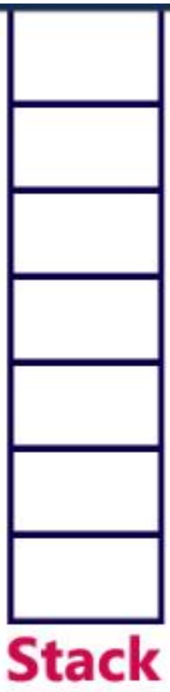




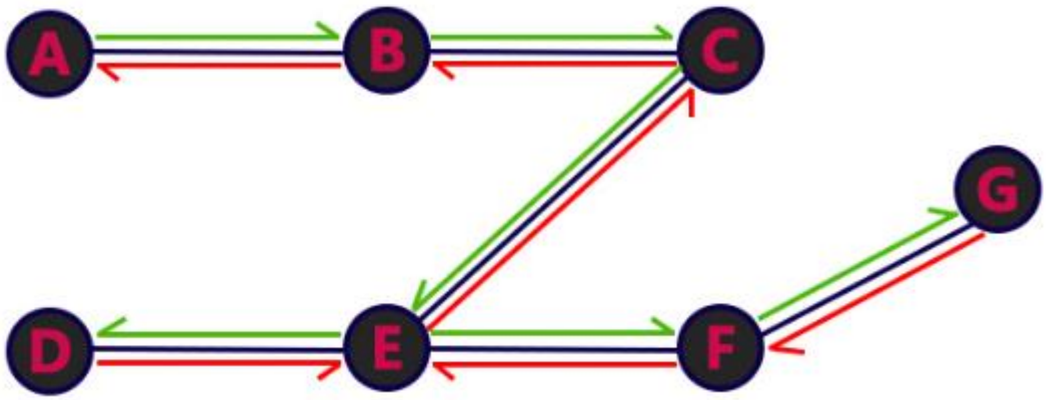
Step 14:

- 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.

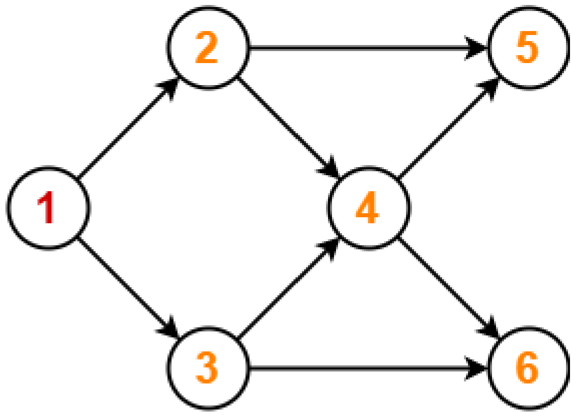


Topological Sort

- 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.
- 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.

Topological Sort Example-

Consider the following directed acyclic graph



Topological Sort Example

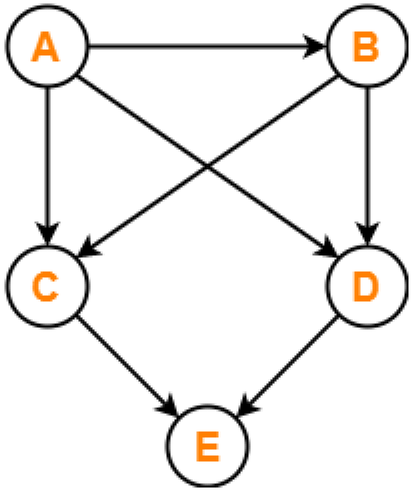
For this graph, following 4 different topological orderings are possible

1 2 3 4 5 6
1 2 3 4 6 5
1 3 2 4 5 6
1 3 2 4 6 5

Applications of Topological Sort-

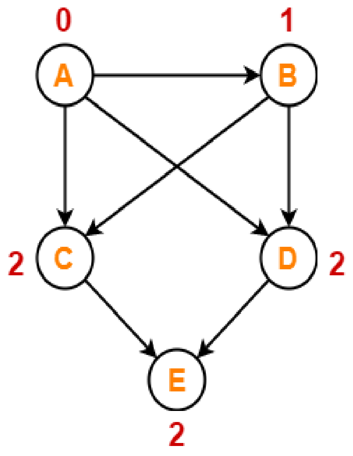
- Few important applications of topological sort are-
  - Scheduling jobs from the given dependencies among jobs
  - Instruction Scheduling
  - Determining the order of compilation tasks to perform in makefiles
  - Data Serialization

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



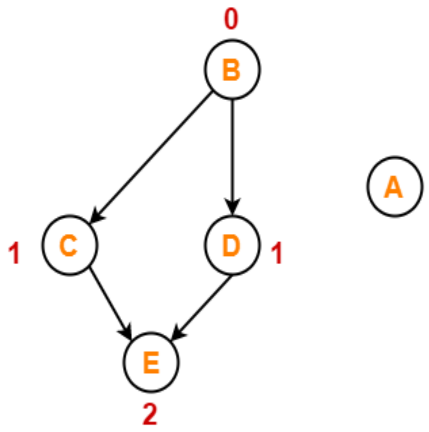
Step-01:

Write in-degree of each vertex-



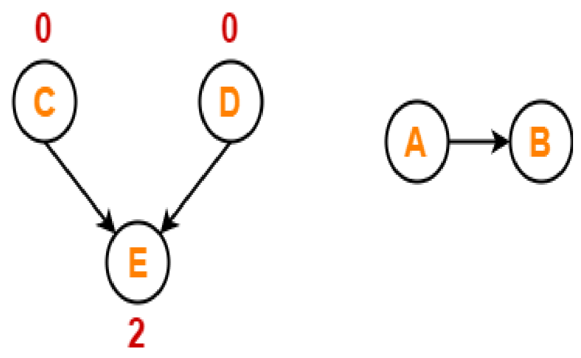
Step-02:

- 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:

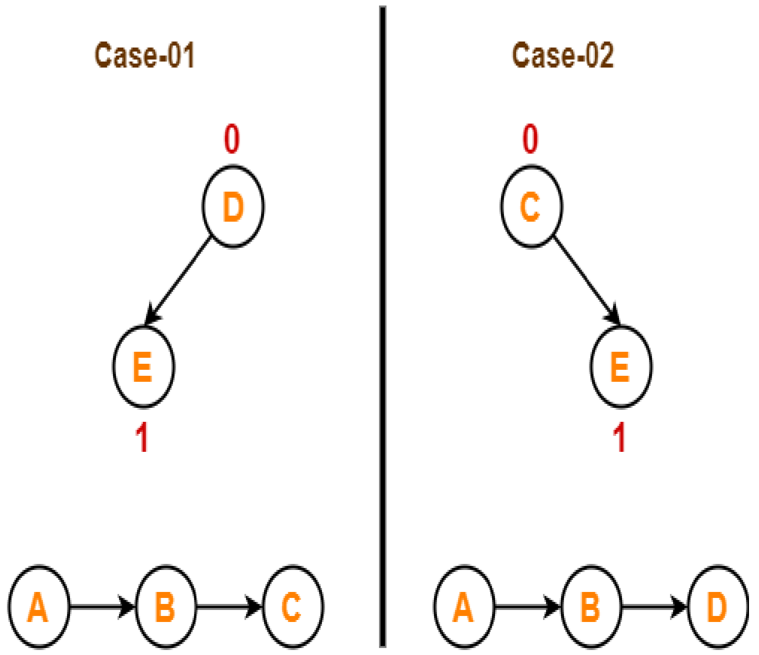
There are two vertices with the least in-degree. So, following 2 cases are possible-

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:

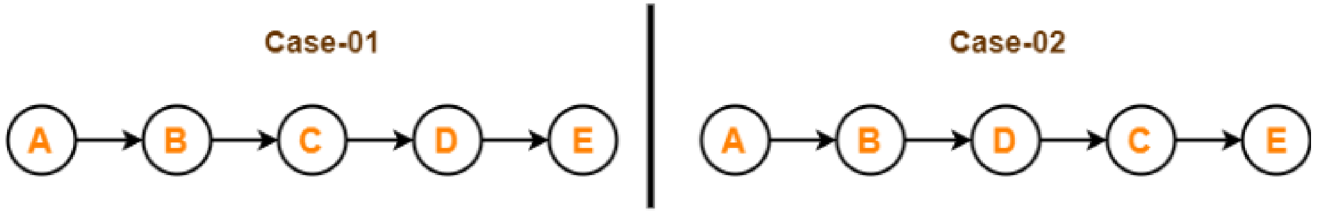
Now, the above two cases are continued separately in the similar manner.

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.



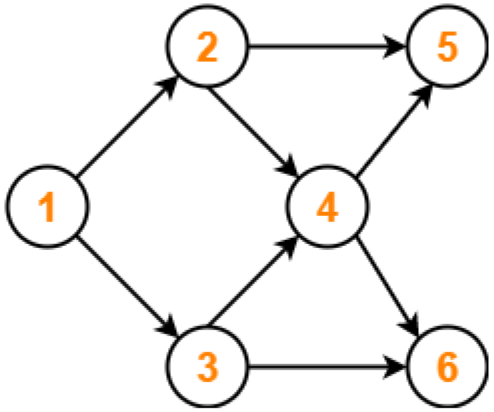
Conclusion-

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

A B C D E

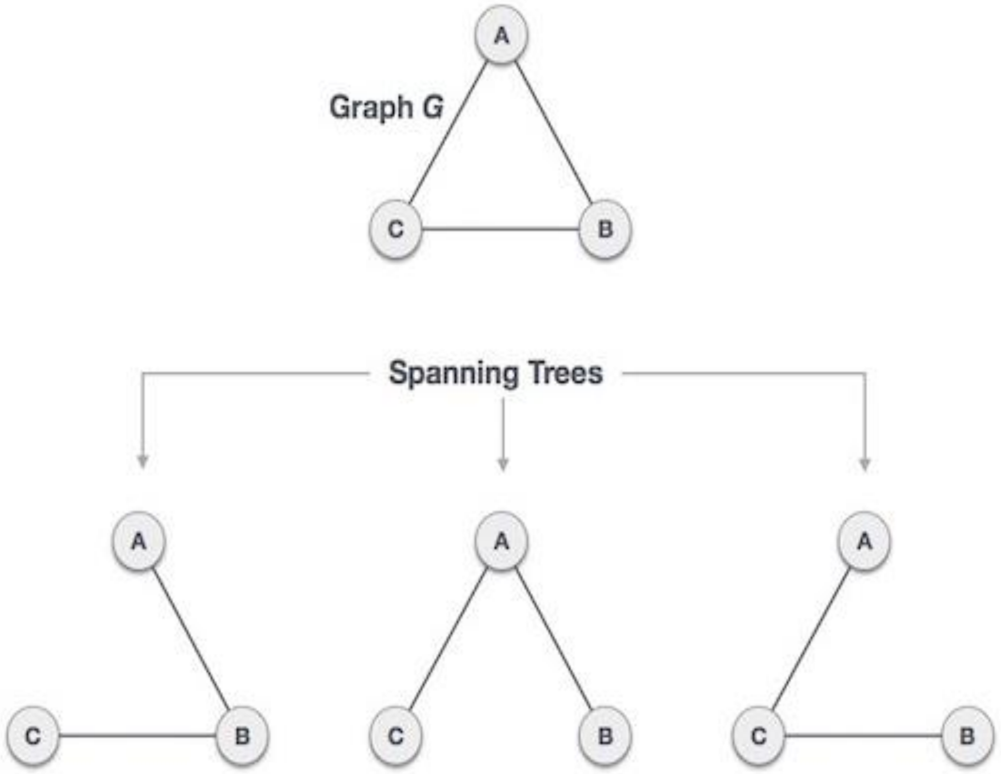
A B D C E

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



Spanning Tree

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.
- Hence, a spanning tree does not have cycles and it cannot be disconnected..



A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence $\mathbf{3^{3-2} = 3}$ spanning trees are possible.

General Properties of Spanning Tree

One graph can have more than one spanning tree.

Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph.

Common application of spanning trees are −

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

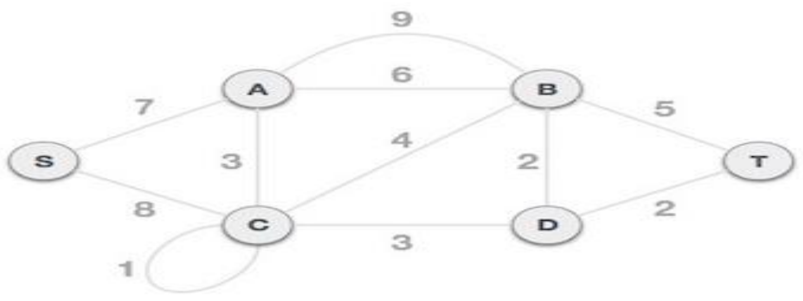
Minimum Spanning Tree (MST)

- In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.
- In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

- two most important spanning tree algorithms here −
  - Kruskal's Algorithm
  - Prim's Algorithm

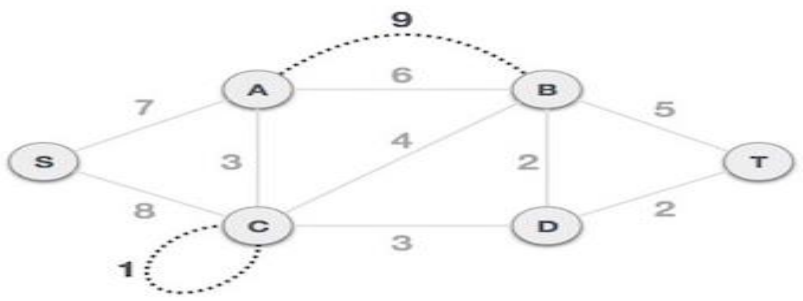
Kruskal's algorithm

- Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach.
- This algorithm treats the graph as a forest and every node it has as an individual tree.
- A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties

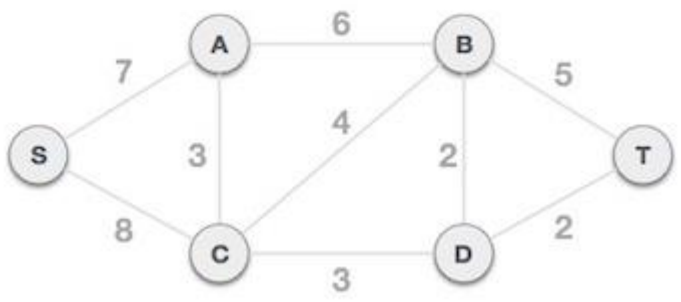


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



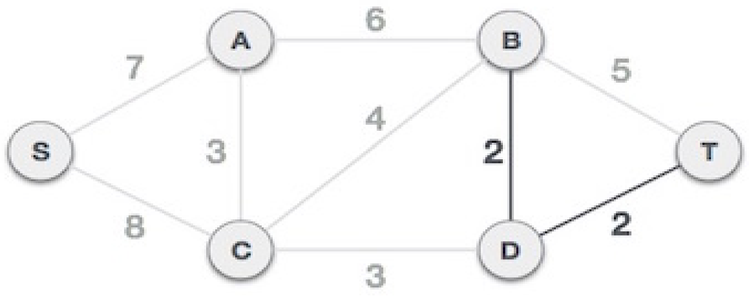
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

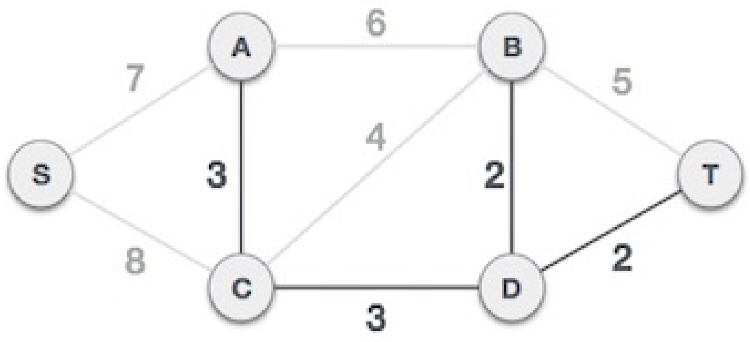
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

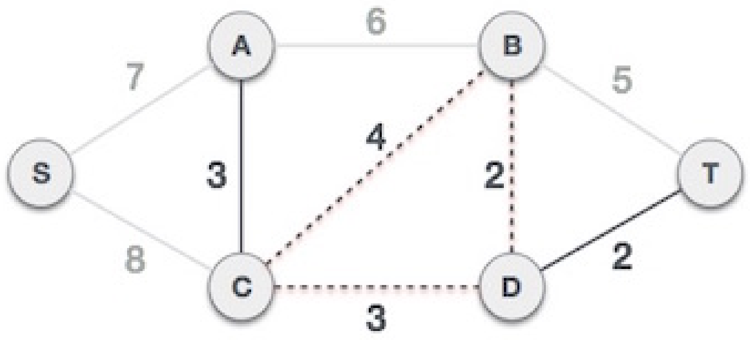


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

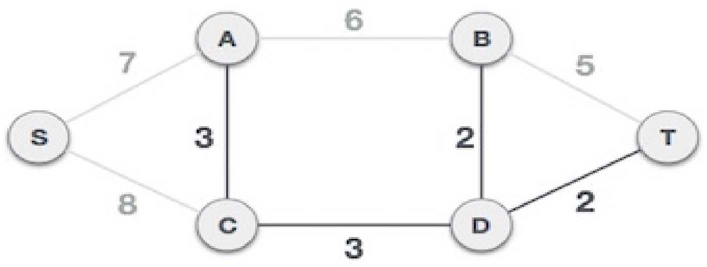
Next cost is 3, and associated edges are A,C and C,D. We add them again −



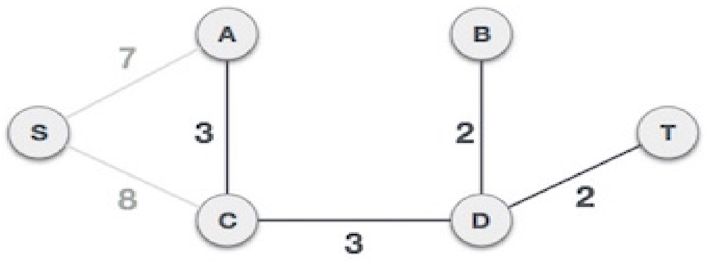
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



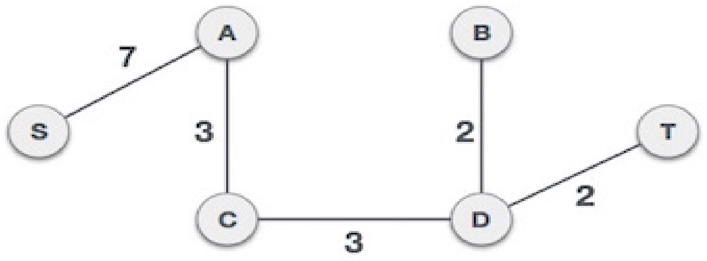
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.

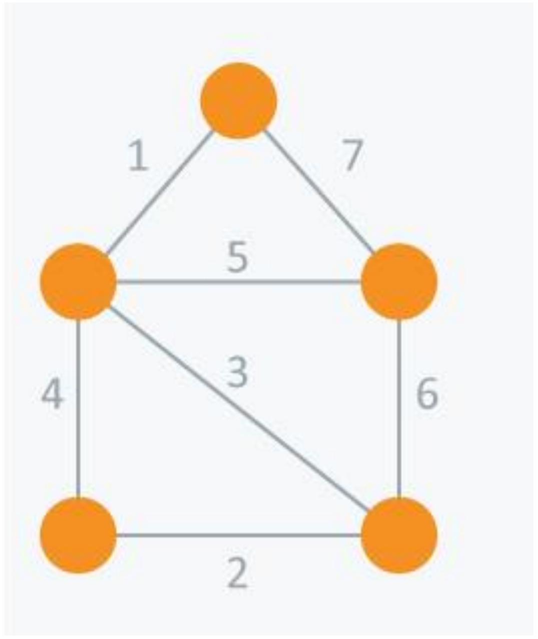


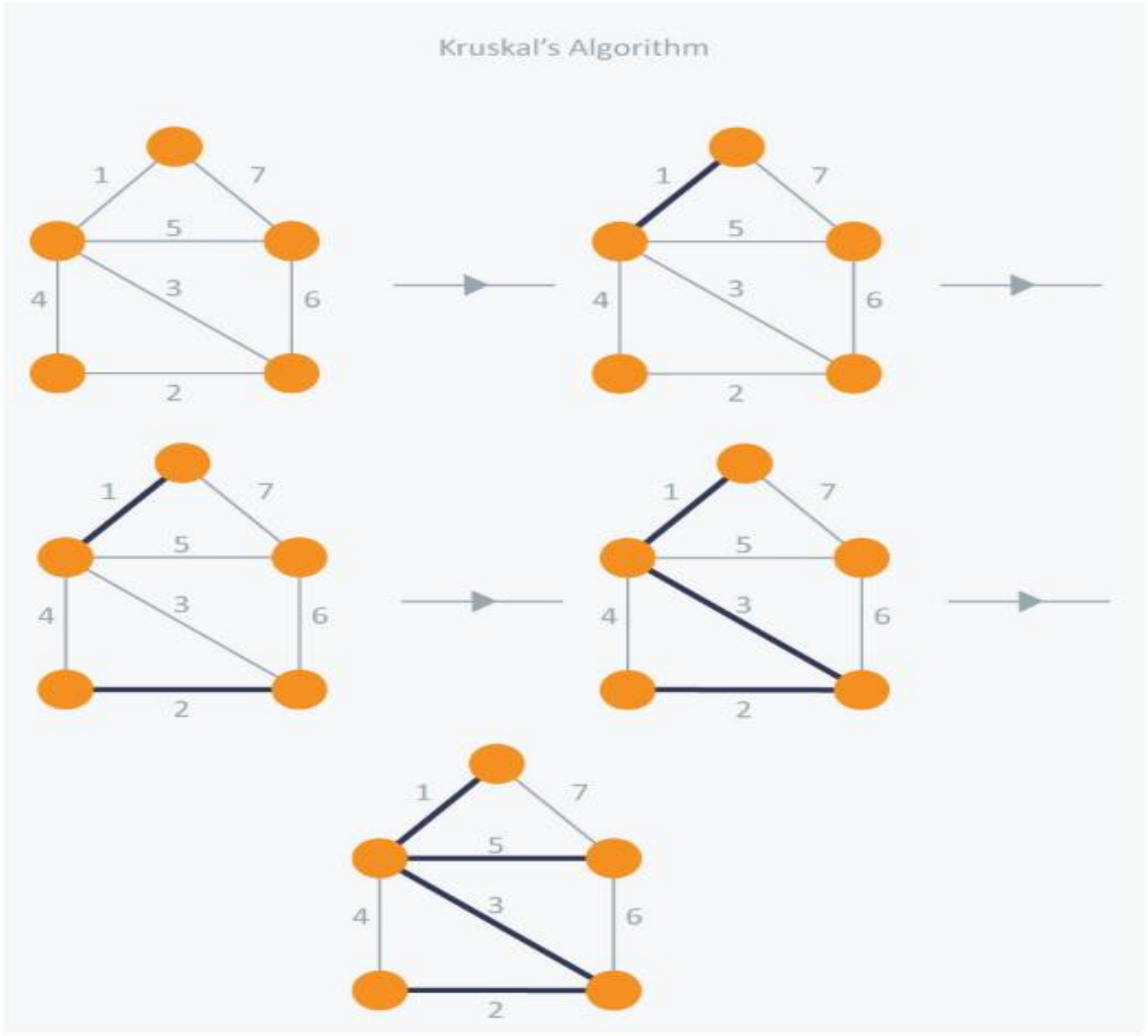
Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



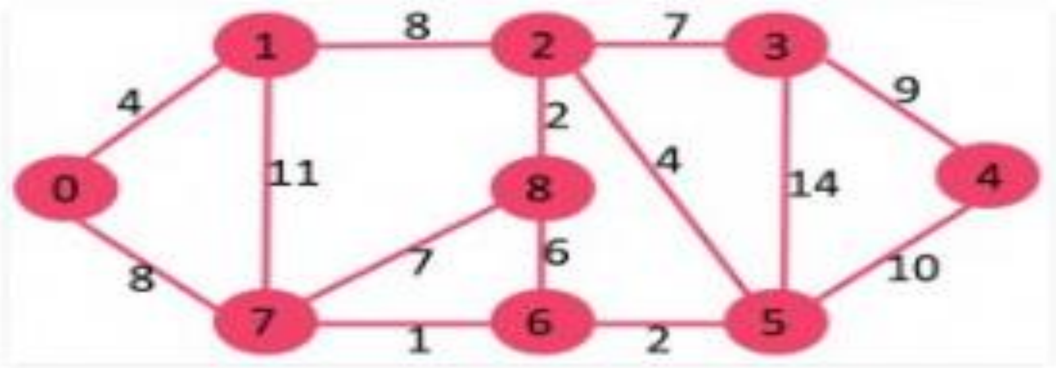
By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Kruskal's Algorithm

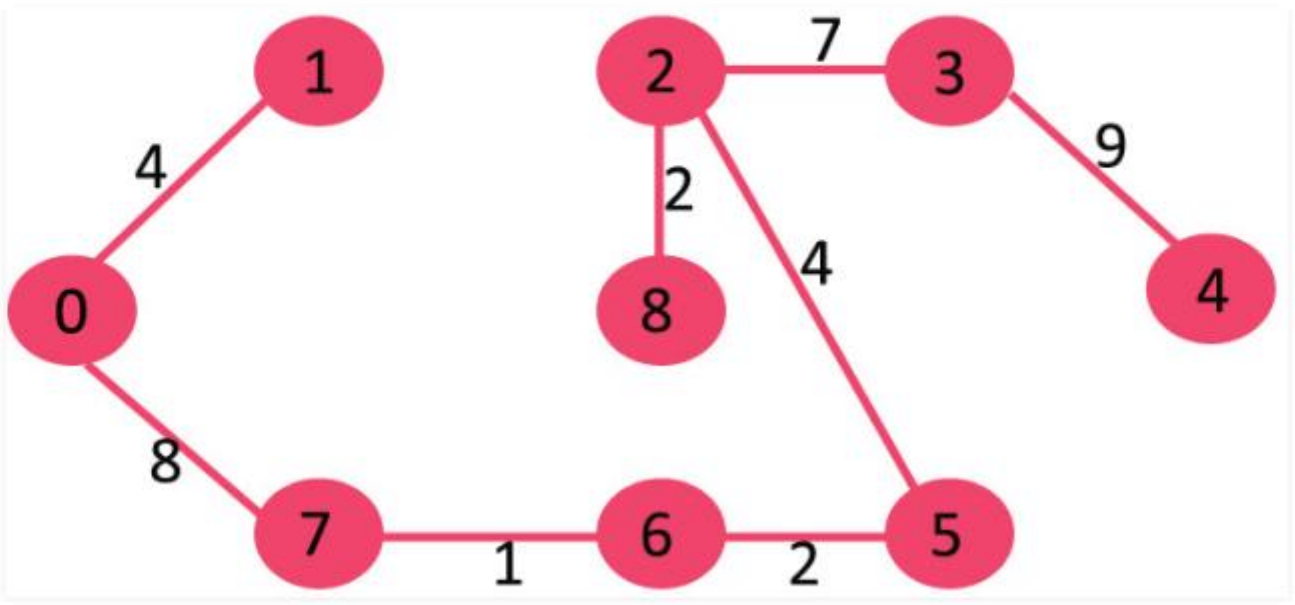




minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5)



0
1
2
3
4
5
6
7
8
4
8
11
8
7
9
2
4
14
10
7
6
1
2

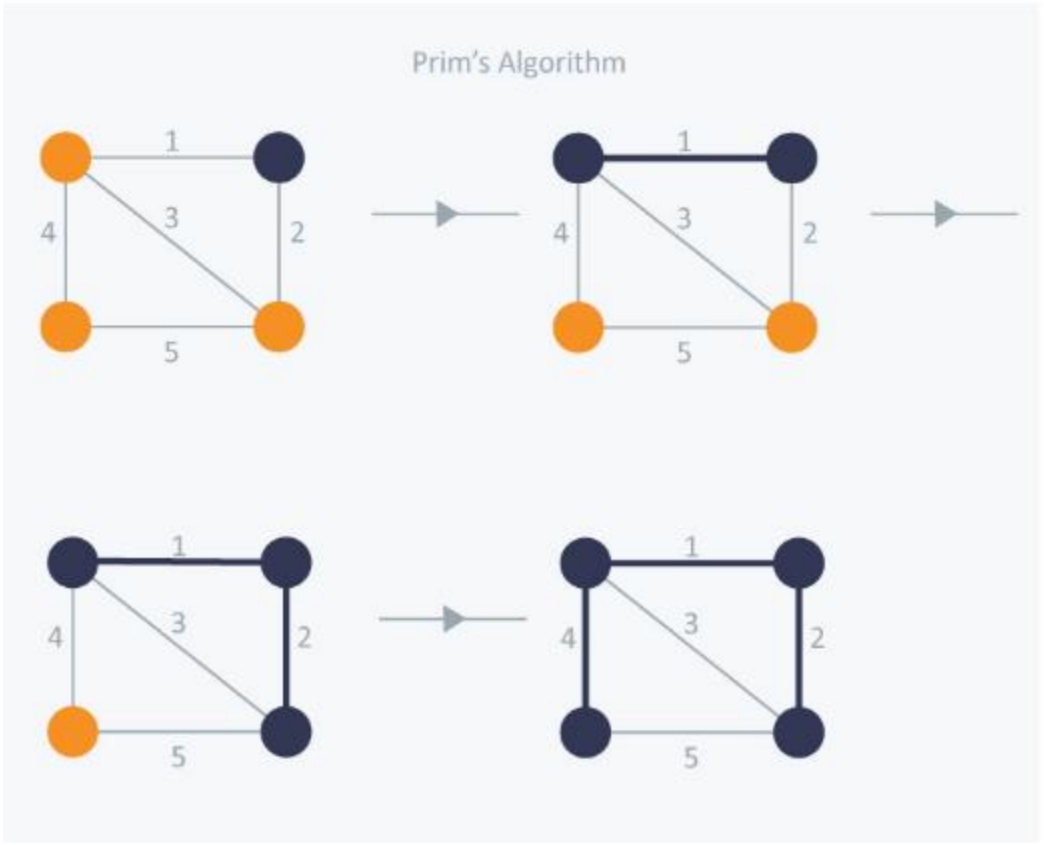

0
1
2
3
4
5
6
7
8
4
8
1
2
4
2
7
9

- **Time Complexity:**
  In Kruskal’s algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(ElogV), which is the overall Time Complexity of the algorithm.

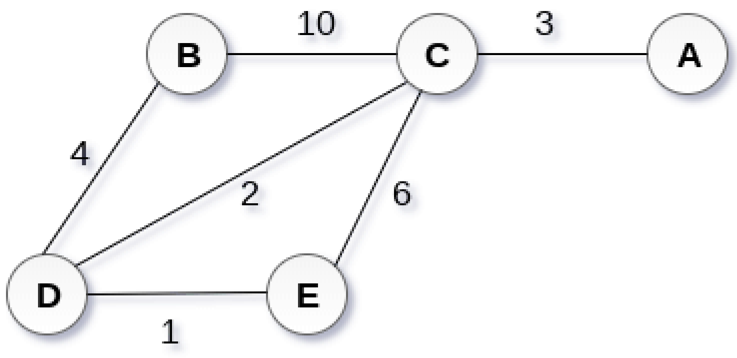
Prim’s Algorithm

- Prim’s Algorithm also use Greedy approach to find the minimum spanning tree.
- In Prim’s Algorithm we grow the spanning tree from a starting position.
- Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

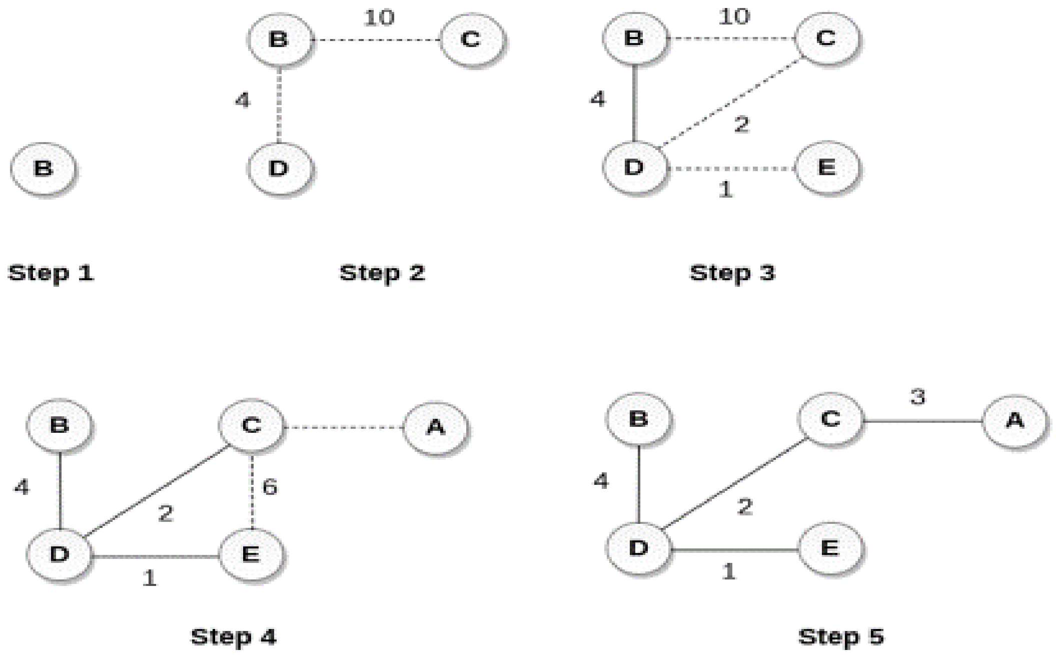
- **Algorithm Steps:**
- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

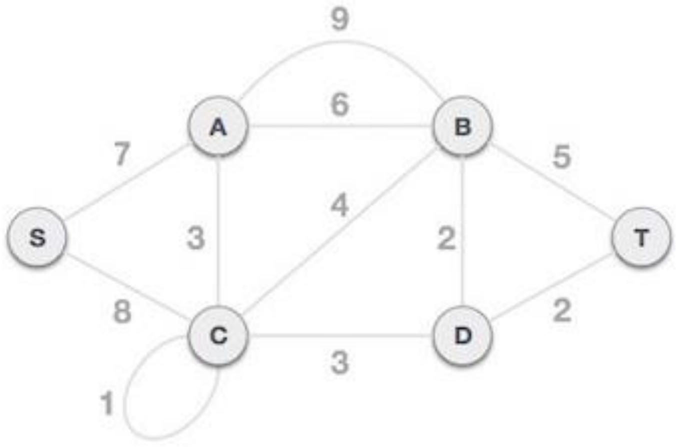

Prim's Algorithm
1
2
3
4
5

- In Prim’s Algorithm, we will start with an arbitrary node (it doesn’t matter which one) and mark it.
- In each iteration we will mark a new vertex that is adjacent to the one that we have already marked.
- As a greedy algorithm, Prim’s algorithm will select the cheapest edge and mark the vertex.
- So we will simply choose the edge with weight 1.
- In the next iteration we have three options, edges with weight 2, 3 and 4.
- So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5.
- But we can’t choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).
- **Time Complexity:**
  The time complexity of the Prim’s Algorithm is O((V+E)logV) because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

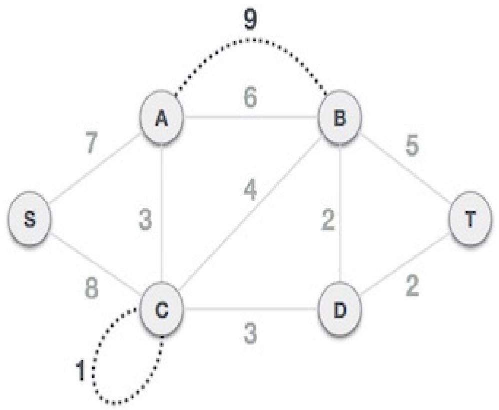


Construct a minimum spanning tree of the graph given in the following figure by using prim's algorithm.

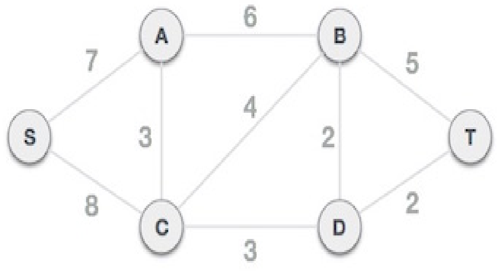

B
Step 1
B
10
C
4
D
Step 2
B
10
C
4
2
D
E
1
Step 3
B
C
A
4
6
2
D
E
1
Step 4
B
C
3
A
4
2
D
E
1
Step 5


9
6
7
5
4
3
2
S
A
B
T
C
D
8
2
3
1

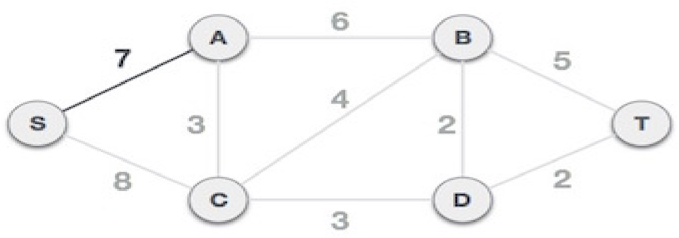
Step 1 - Remove all loops and parallel edges



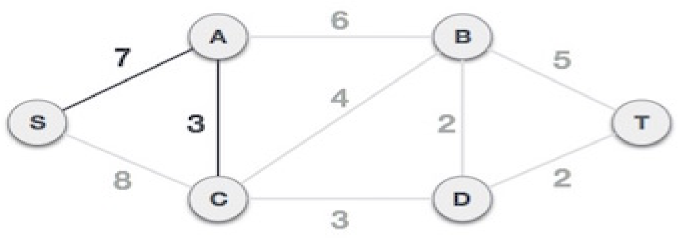
Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.



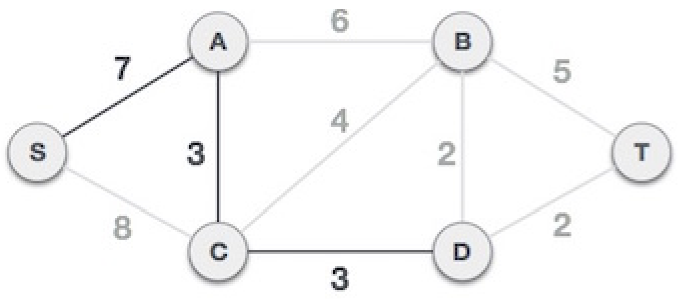
- After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively.
- We choose the edge S,A as it is lesser than the other.



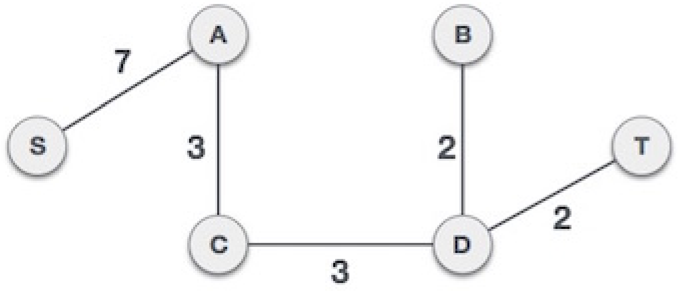
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.


A
B
S
T
C
D
7
3
2
2
3