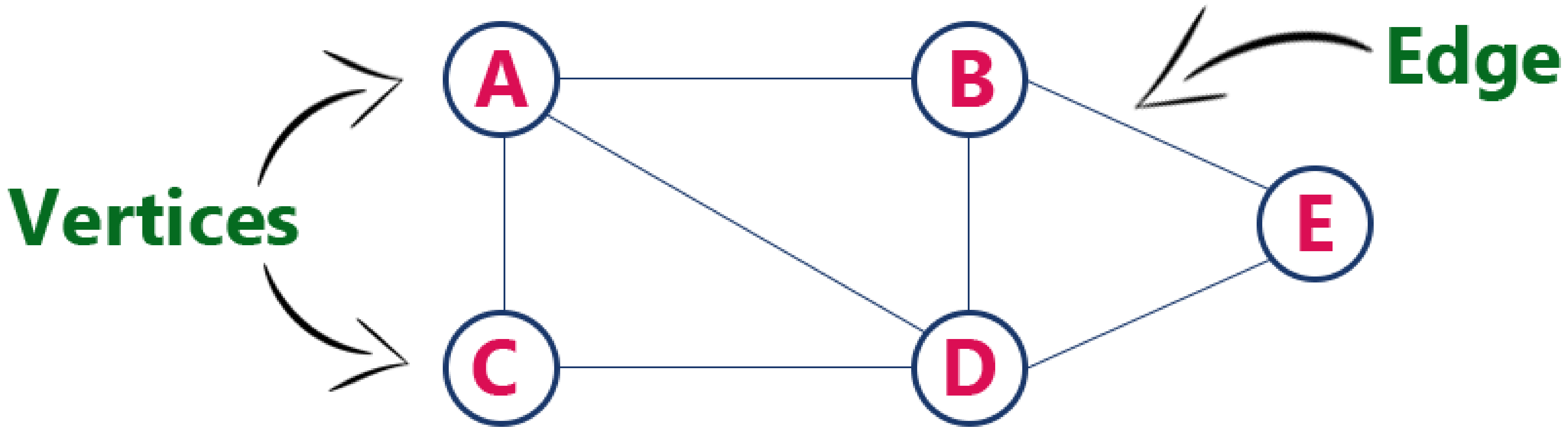
# Graph

## Definition

- Graph is a non-linear data structure.
- It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs).
- Here edges are used to connect the vertices.
- Definition: **Graph is a collection of nodes and edges in which nodes are connected with edges**

## Example



## Terminology

- **Vertex**

Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

- **Edge**

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

## Types of Edges

- Edges are three types.

1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge -** A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted egde is a edge with value (cost) on it.

## Terminology (Contd.)

- **Undirected Graph**

A graph with only undirected edges is said to be undirected graph.

- **Directed Graph**

A graph with only directed edges is said to be directed graph.

- **Mixed Graph**

A graph with both undirected and directed edges is said to be mixed graph.

## Terminology (Contd.)

- **Adjacent**

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

- **Incident**

Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

## Terminology (Contd.)

- **Degree**

Total number of edges connected to a vertex is said to be degree of that vertex.

- **Indegree**

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

- **Outdegree**

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

## Terminology (Contd.)

- **Parallel edges or Multiple edges**

If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

- **Self-loop**

Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

## Terminology (Contd.)

- **Simple Graph**

A graph is said to be simple if there are no parallel and self-loop edges.

- **Path**

A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

## Graph Representation

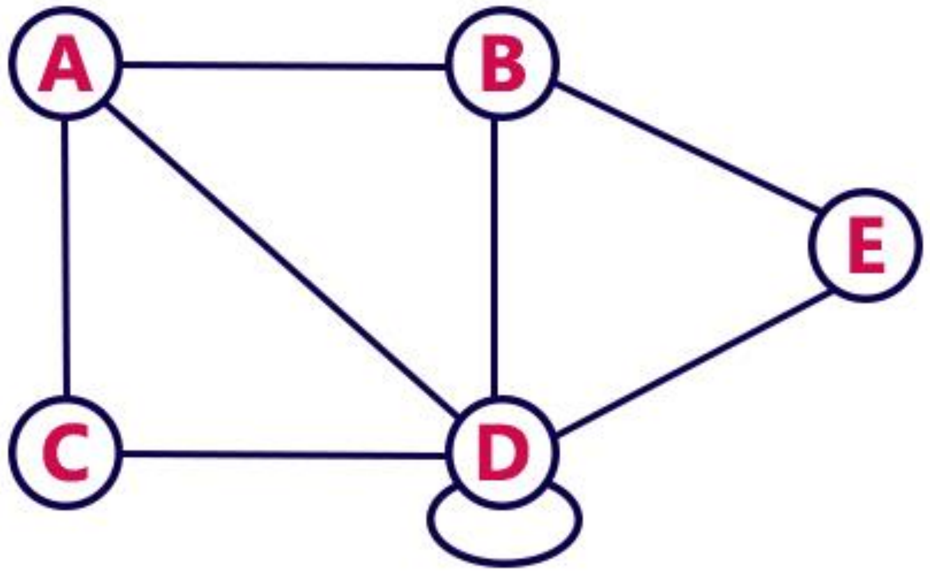
- Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Adjacency List**

## Adjacency Matrix

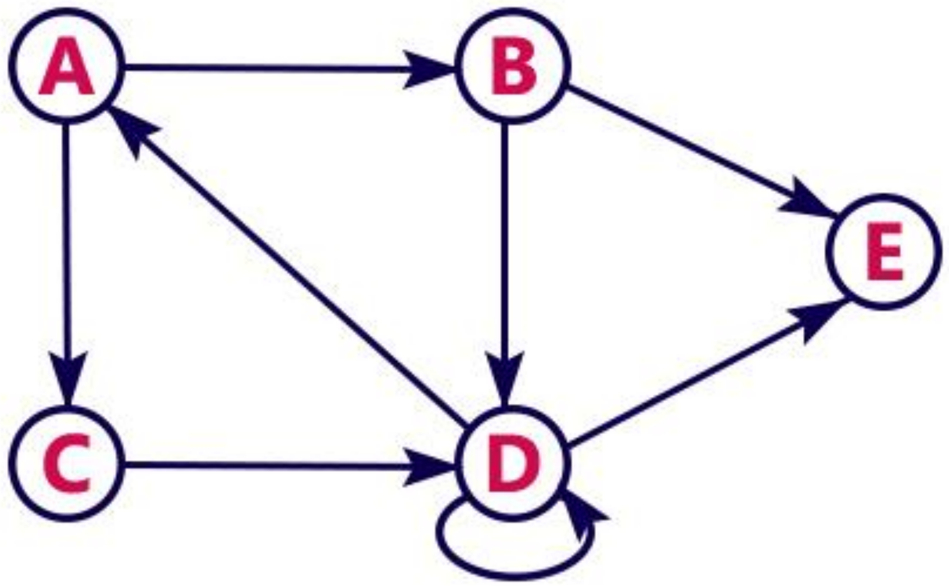
- In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices.
- That means a graph with 4 vertices is represented using a matrix of size 4X4.
- In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

## Example



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

## Example

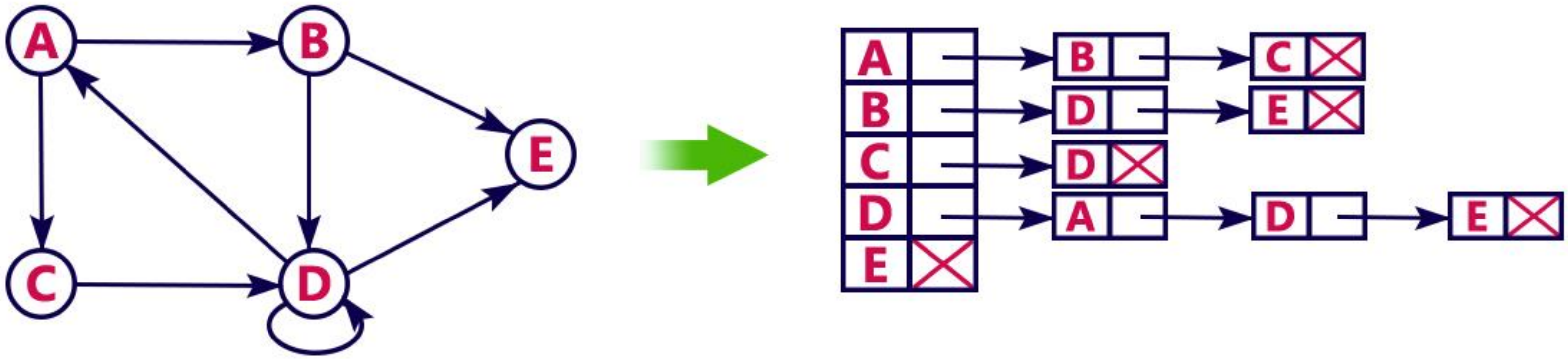


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

## Adjacency List

- In this representation, every vertex of a graph contains list of its adjacent vertices.

## Example



## Graph Traversal

- Graph traversal is a technique used for a searching 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.

## Types

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

## 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.

## Steps

**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.

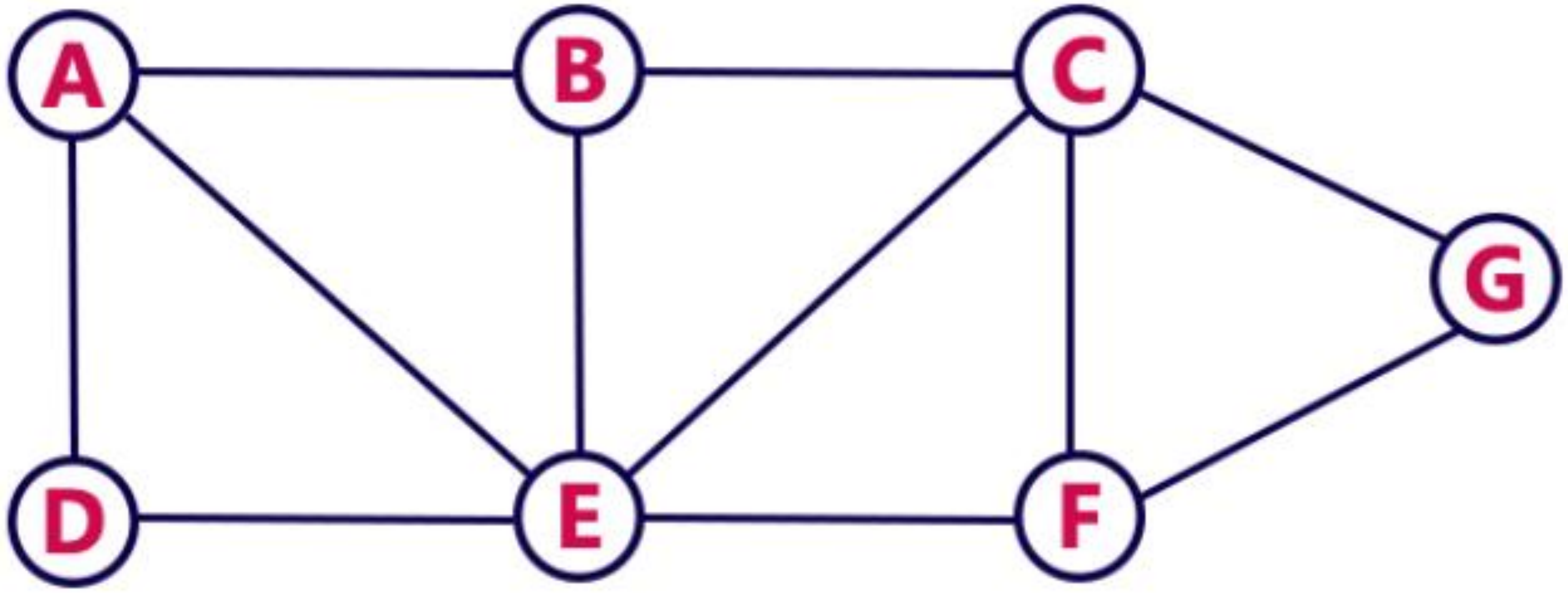
**Step 5 -** When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.

**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

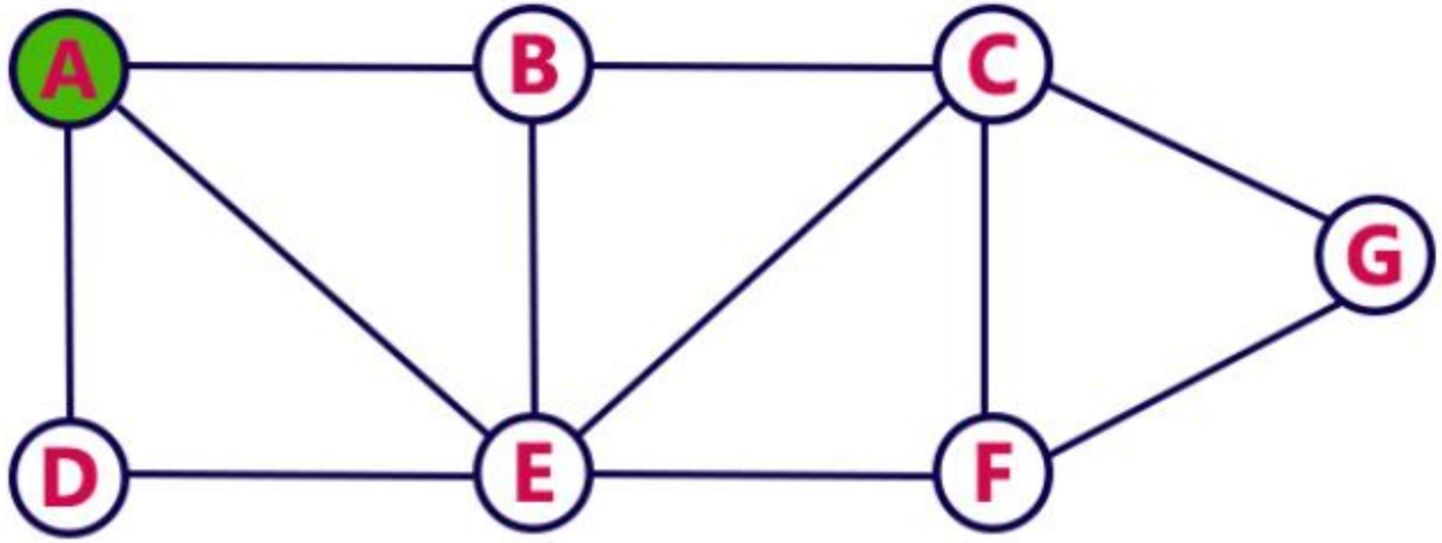
## Example

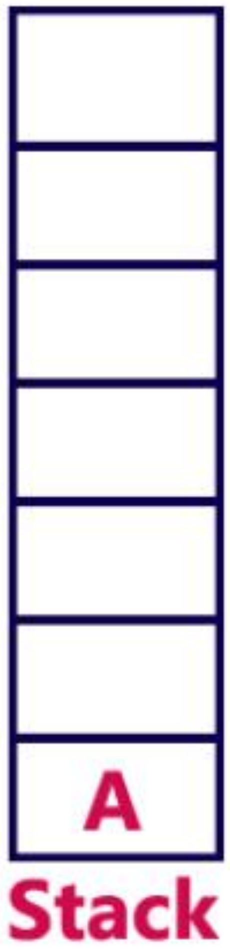
Consider the following example graph to perform DFS traversal



## Step 1:

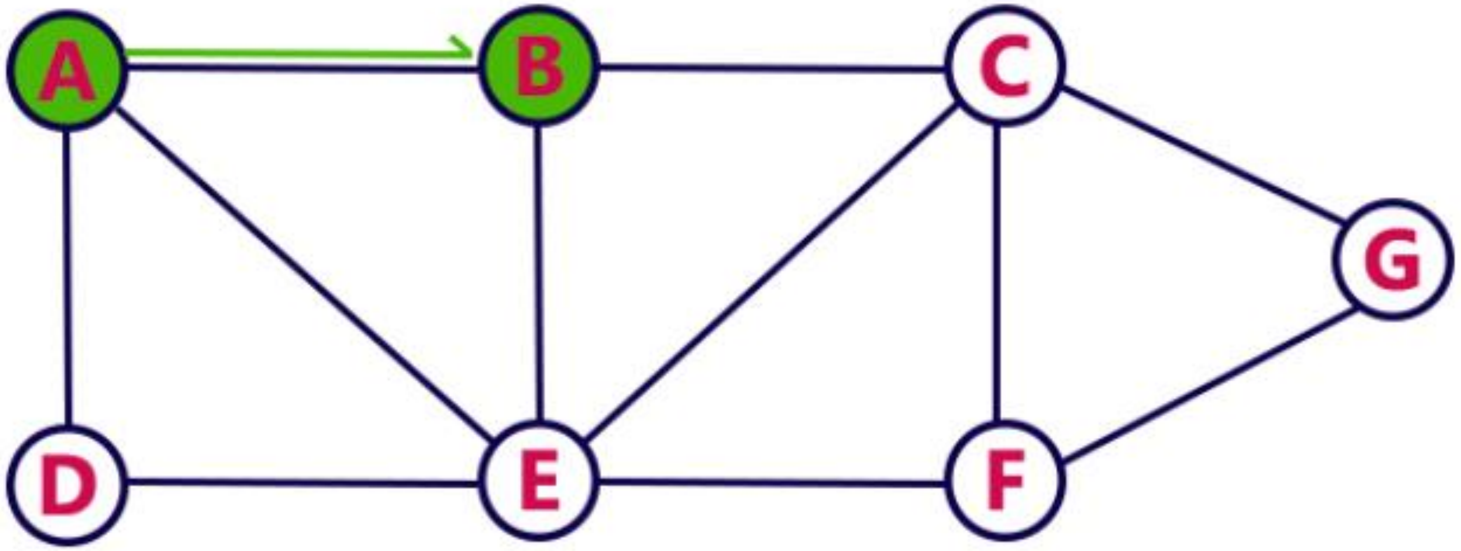
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

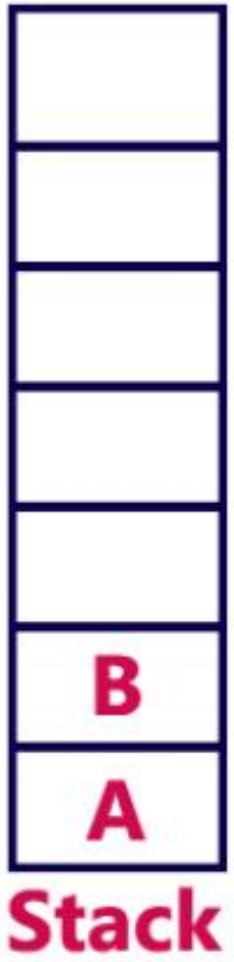




**Step 2:**

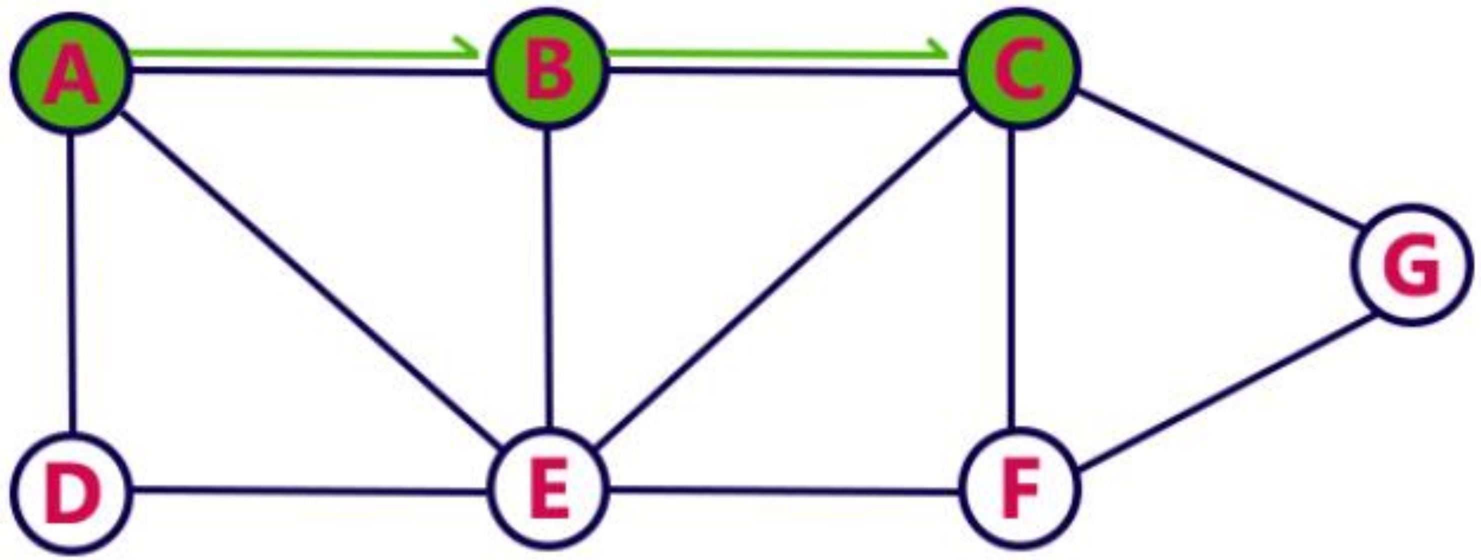
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

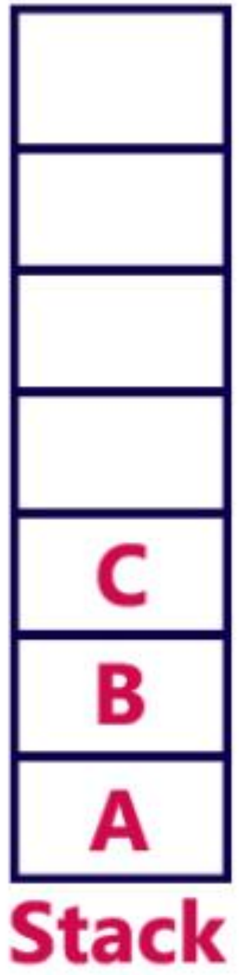




## Step 3:

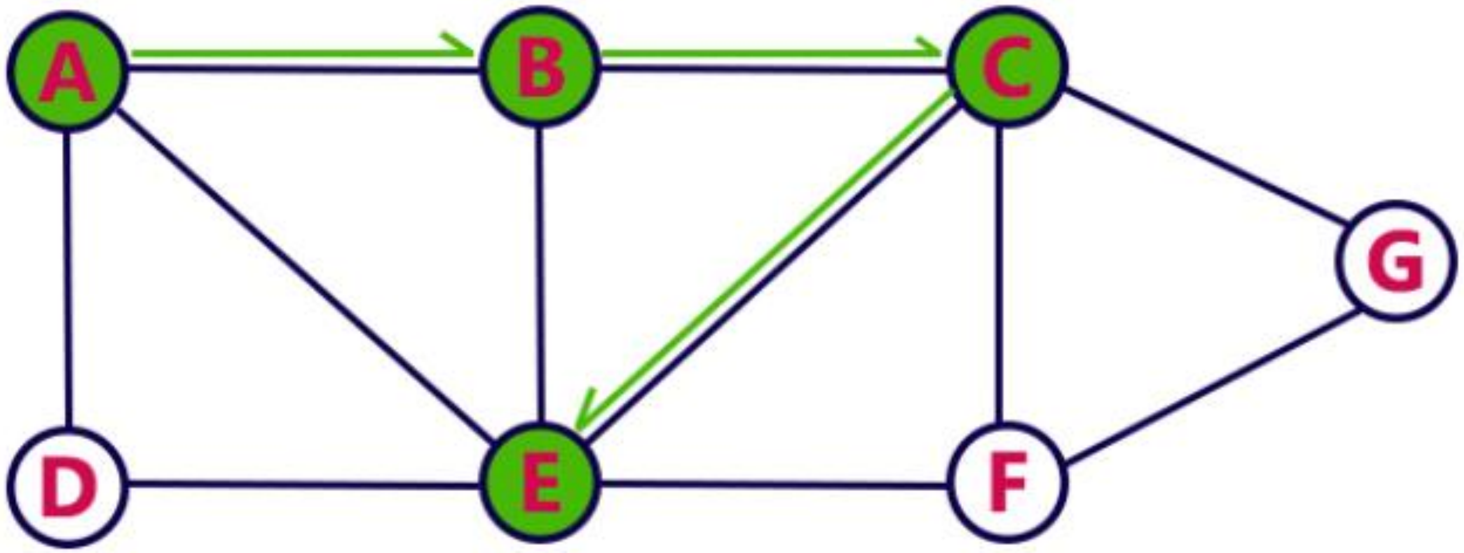
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

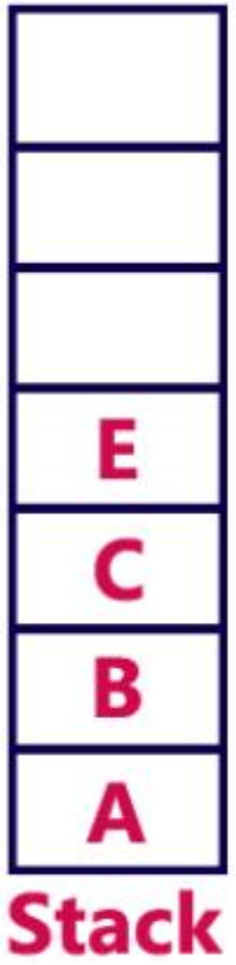




## Step 4:

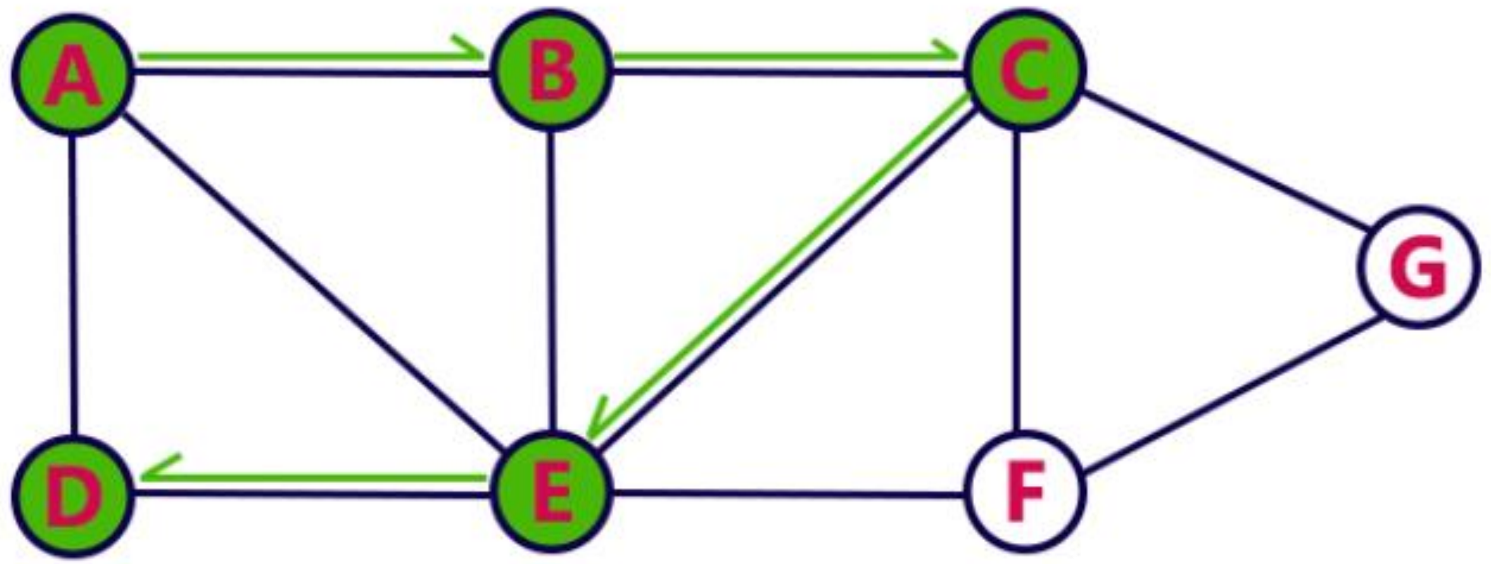
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

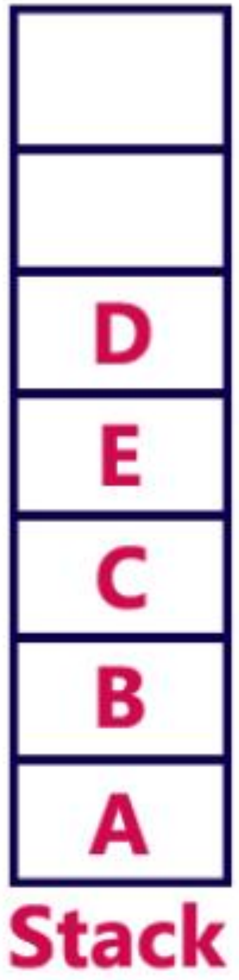




## Step 5:

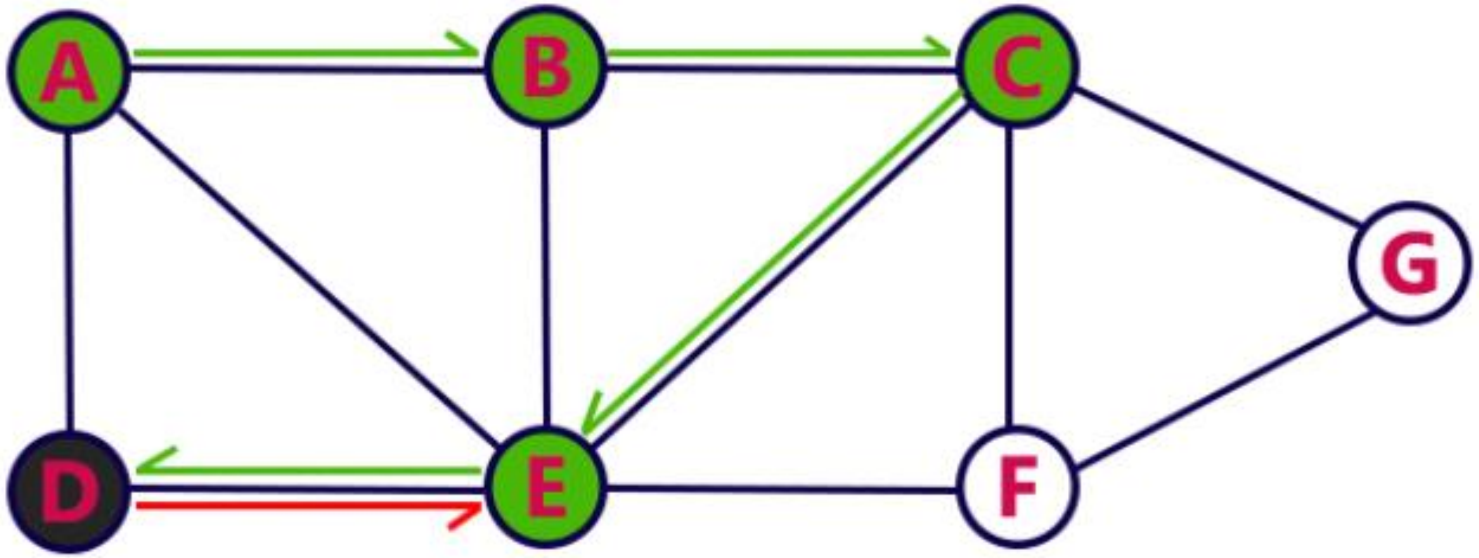
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

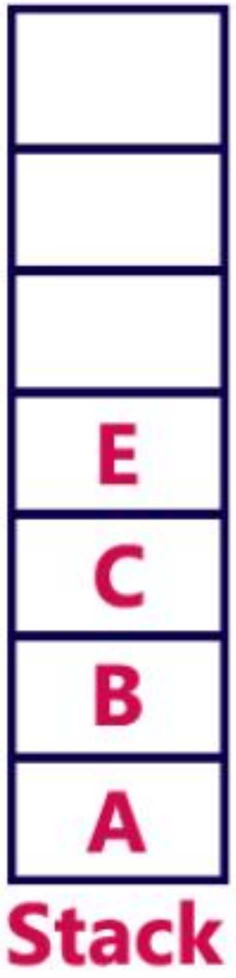




**Step 6:**

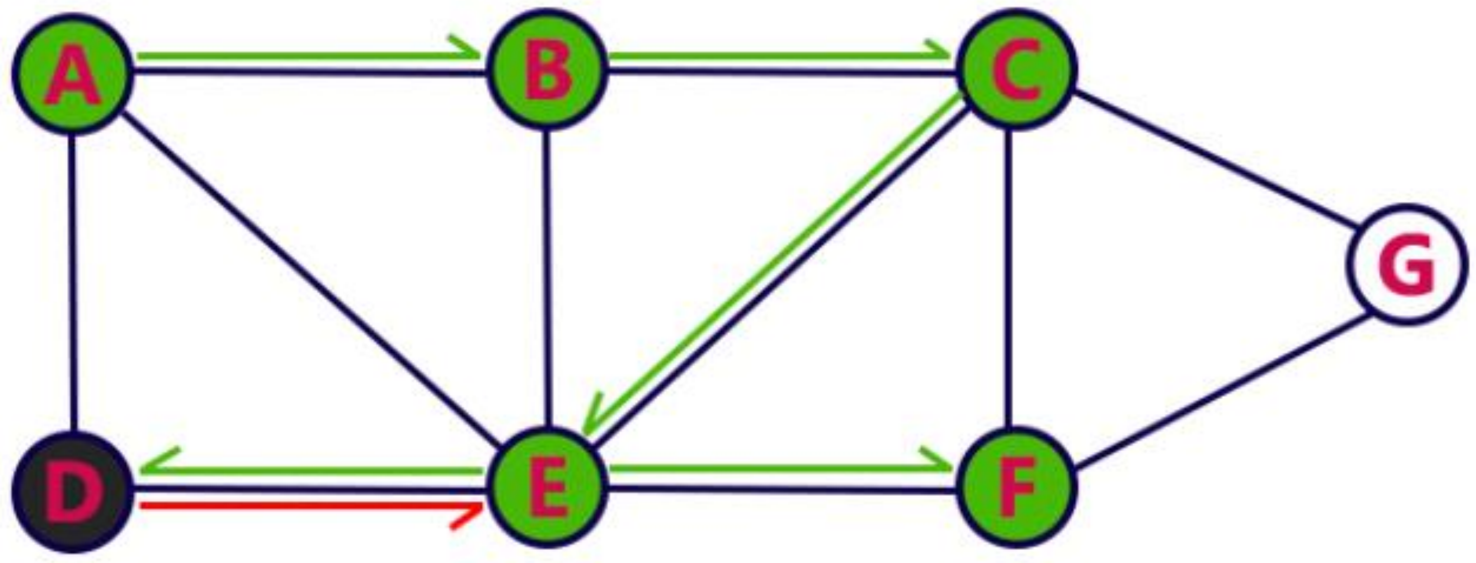
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

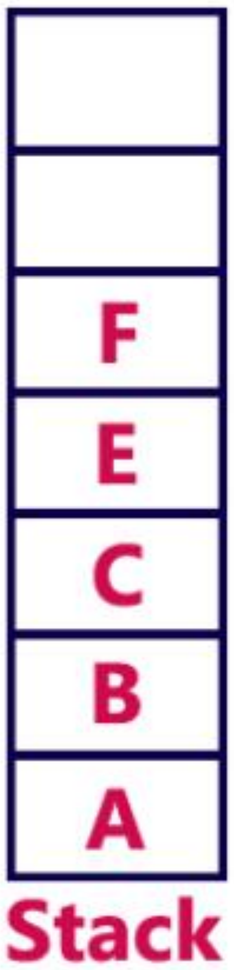




## Step 7:

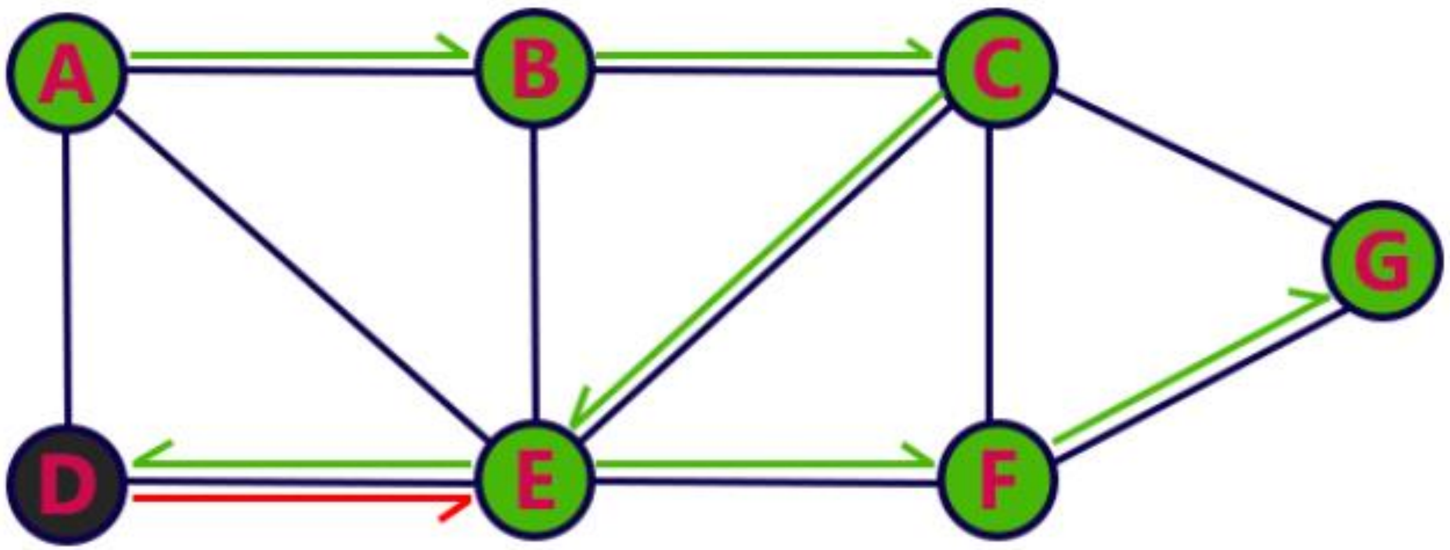
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

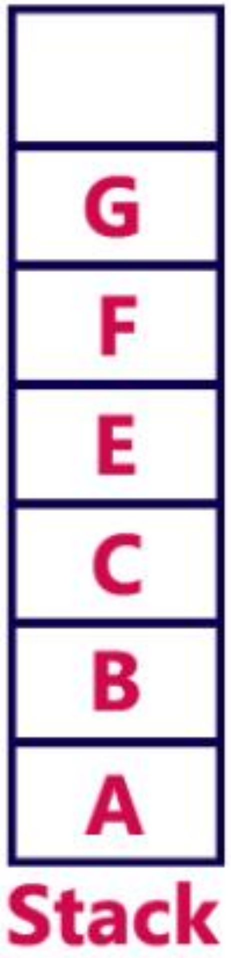




## Step 8:

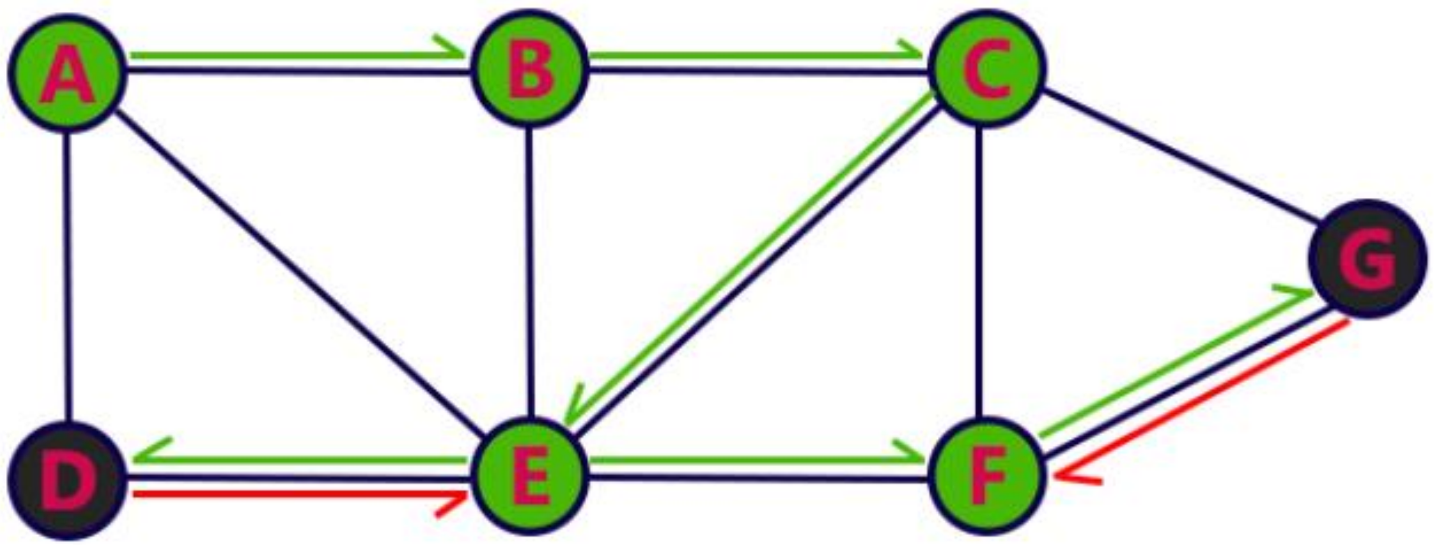
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

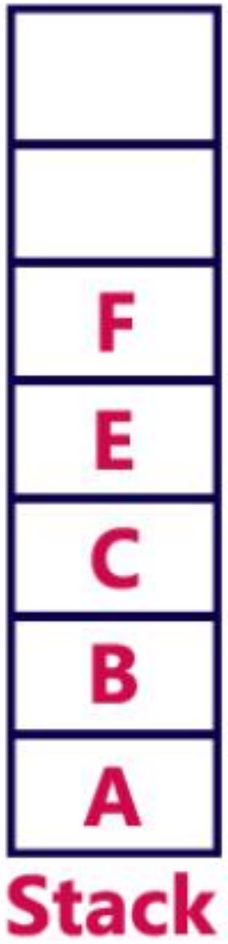




## Step 9:

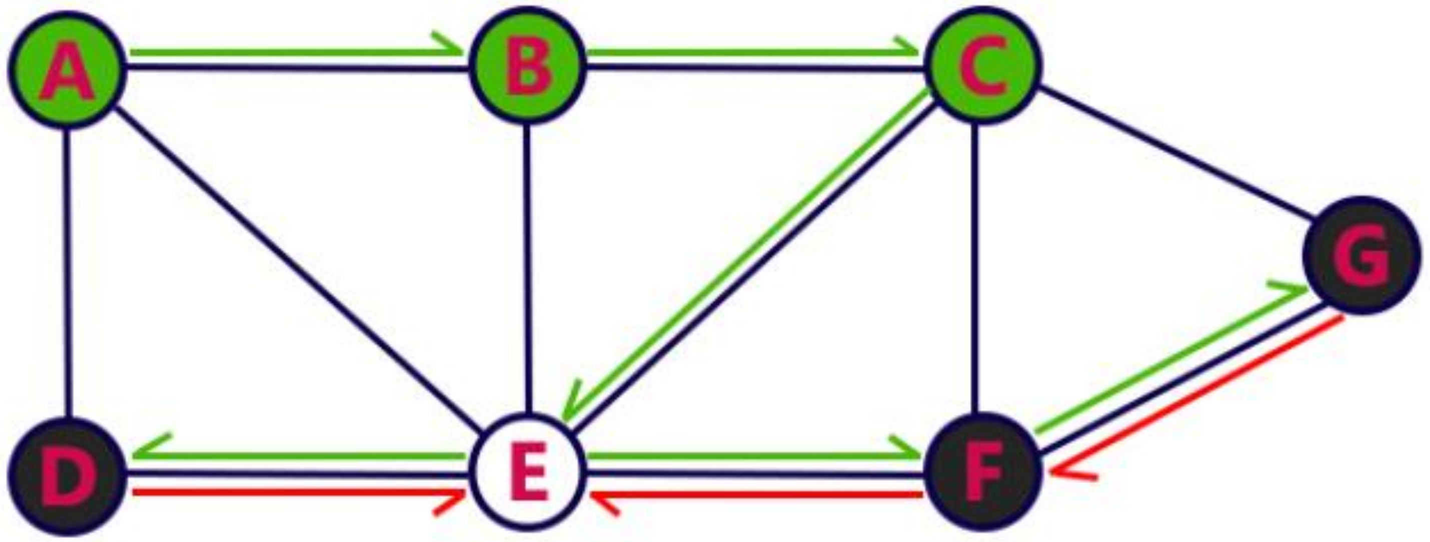
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

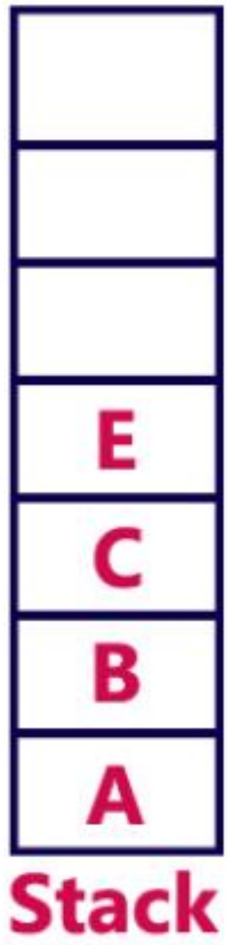




**Step 10:**

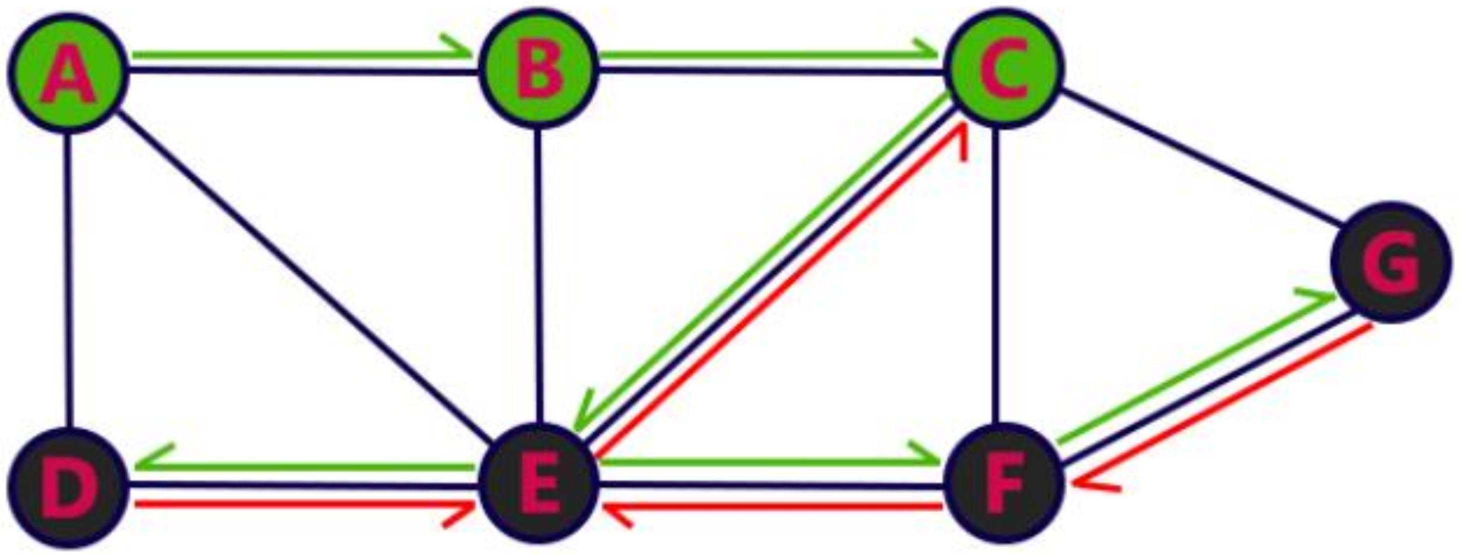
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

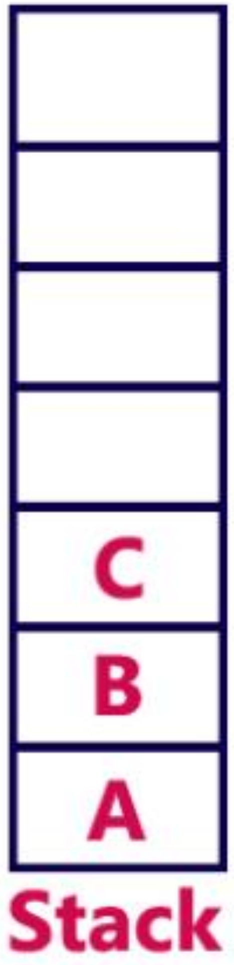




**Step 11:**

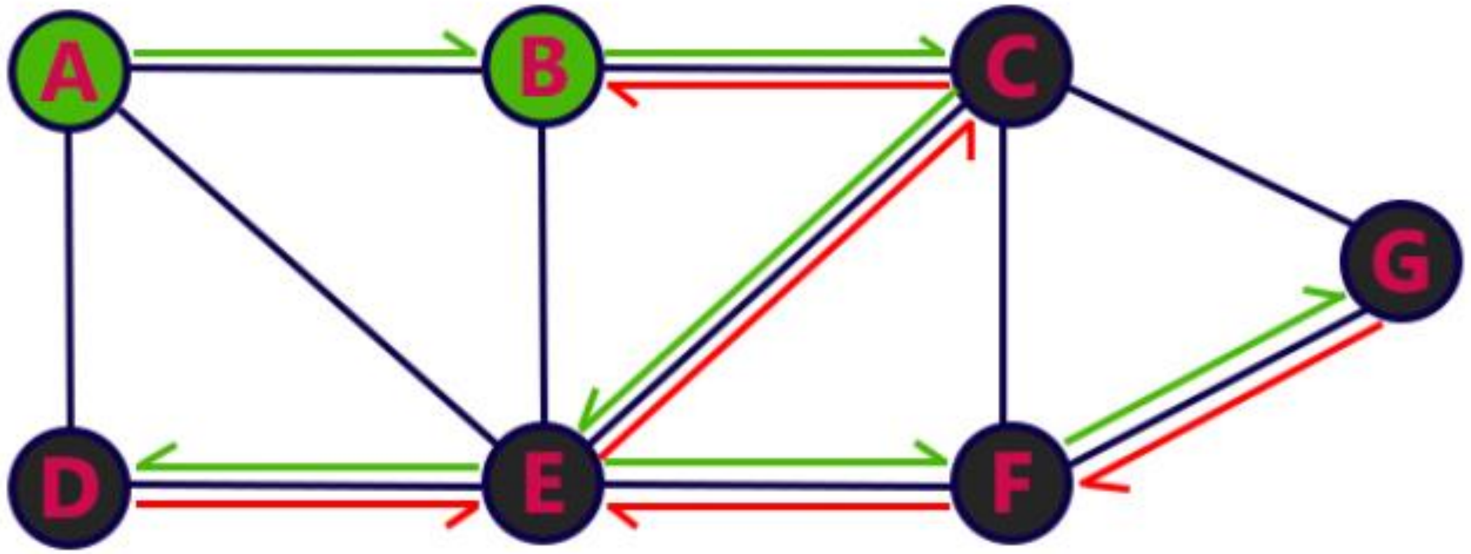
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

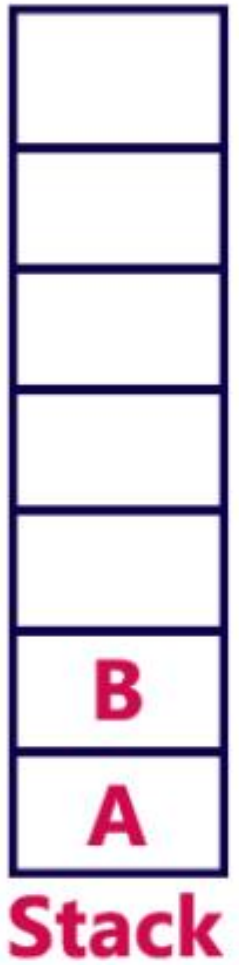




**Step 12:**

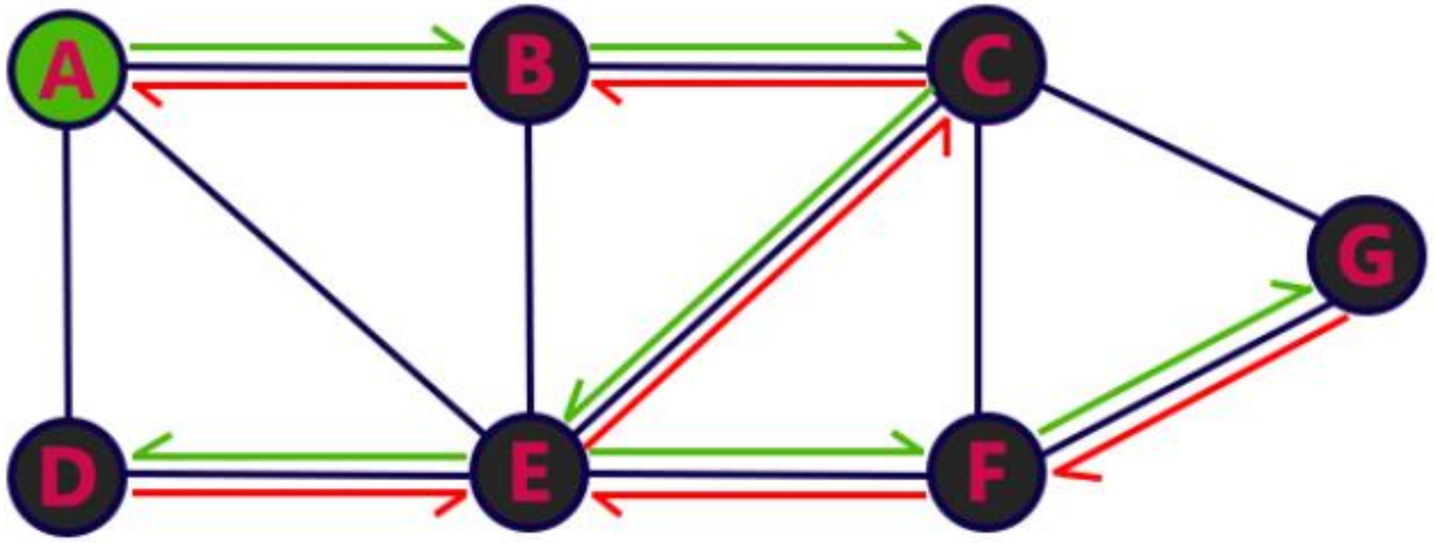
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

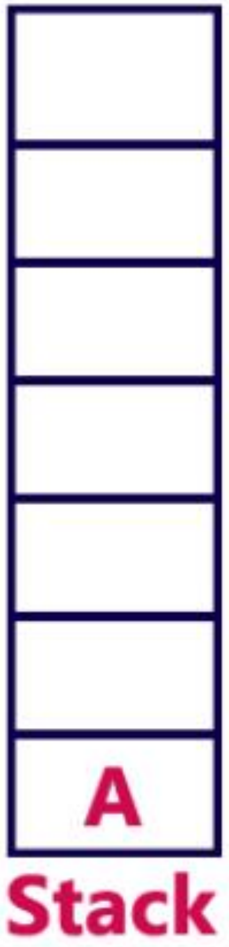




**Step 13:**

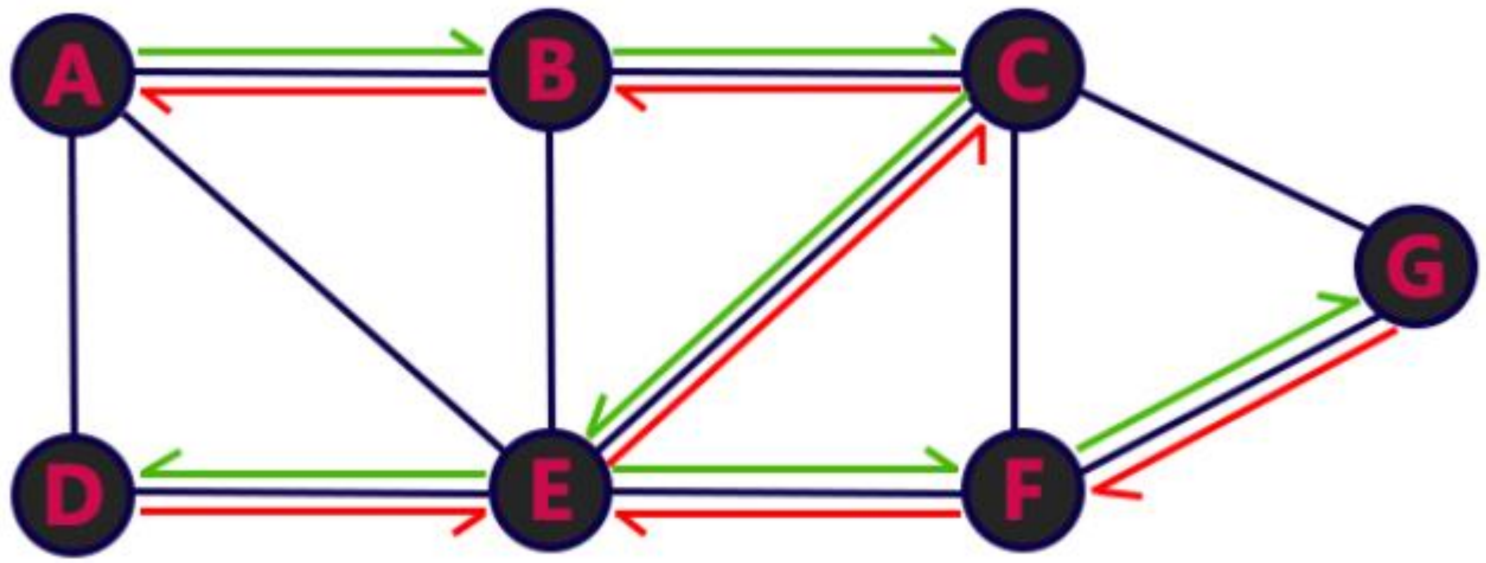
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

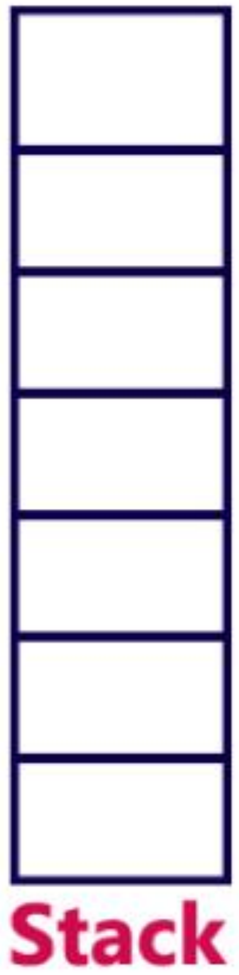




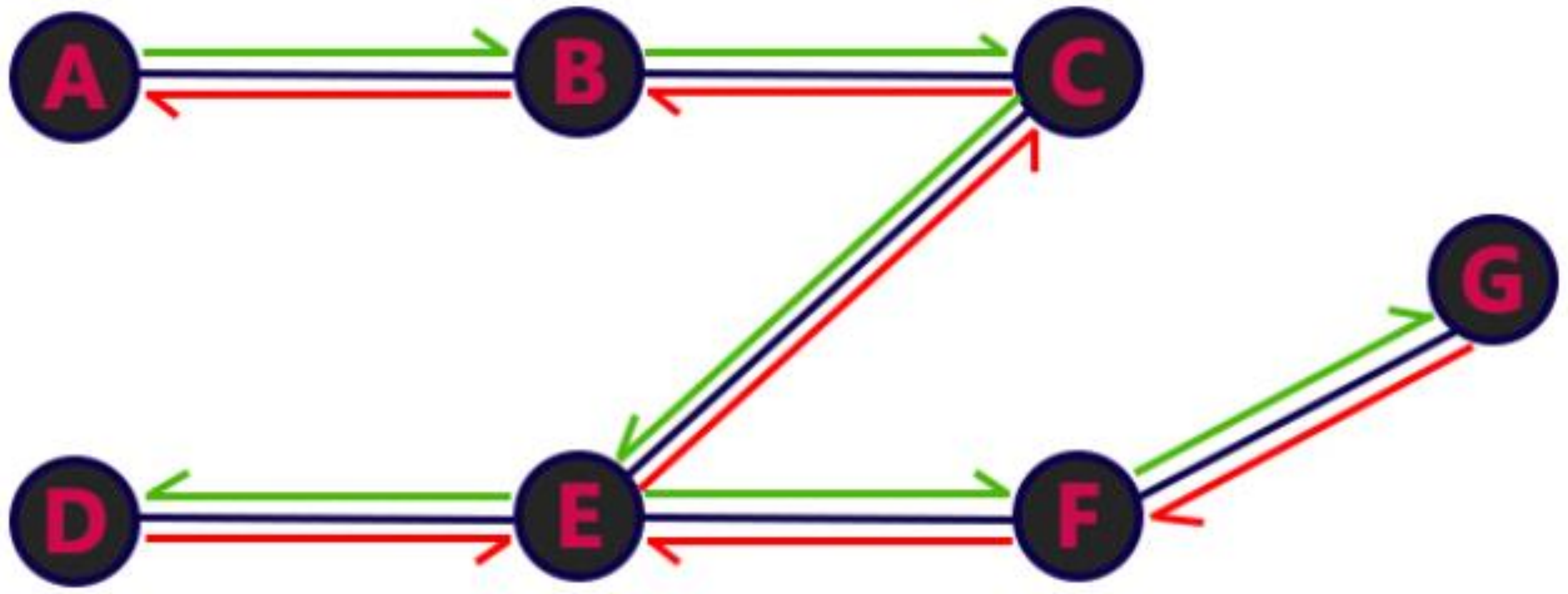
## Step 14:

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





## Final Result



## Breadth First Search (BFS)

- 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.

## Steps

**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.

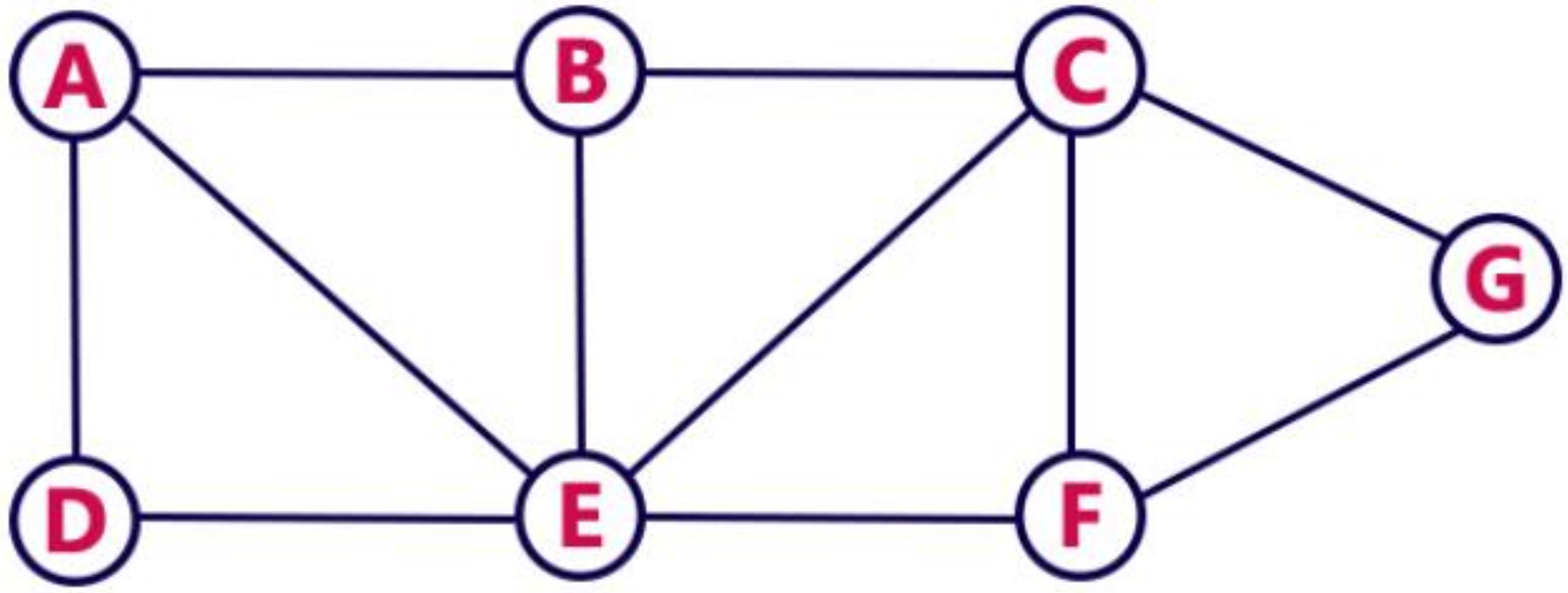
**Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

**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

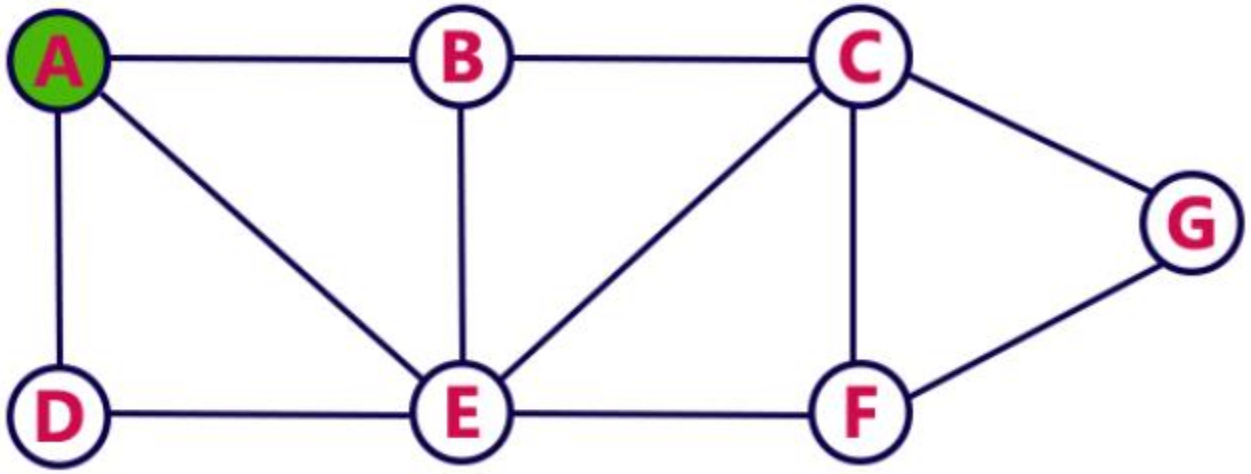
## Example

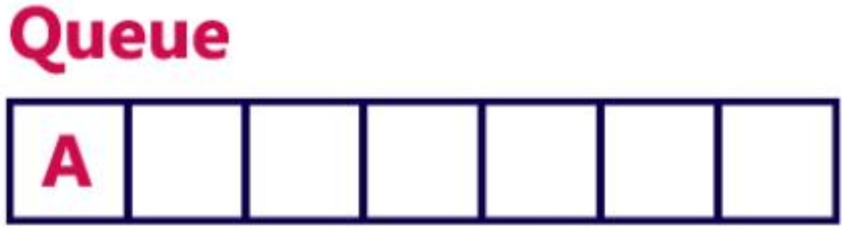
Consider the following example graph to perform BFS traversal



**Step 1:**

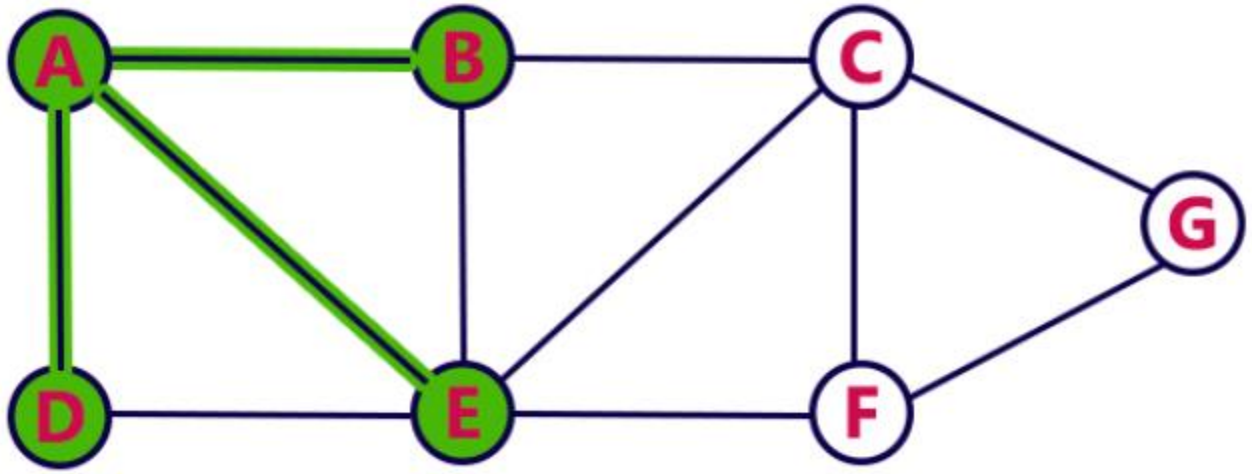
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

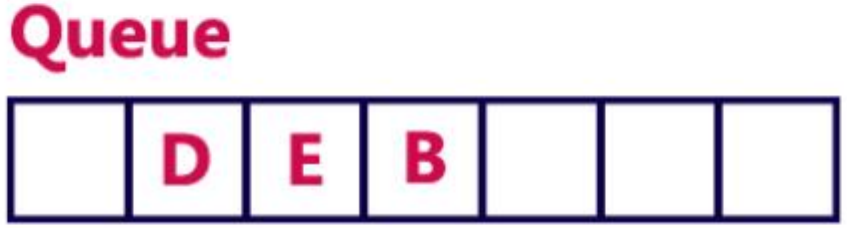




## Step 2:

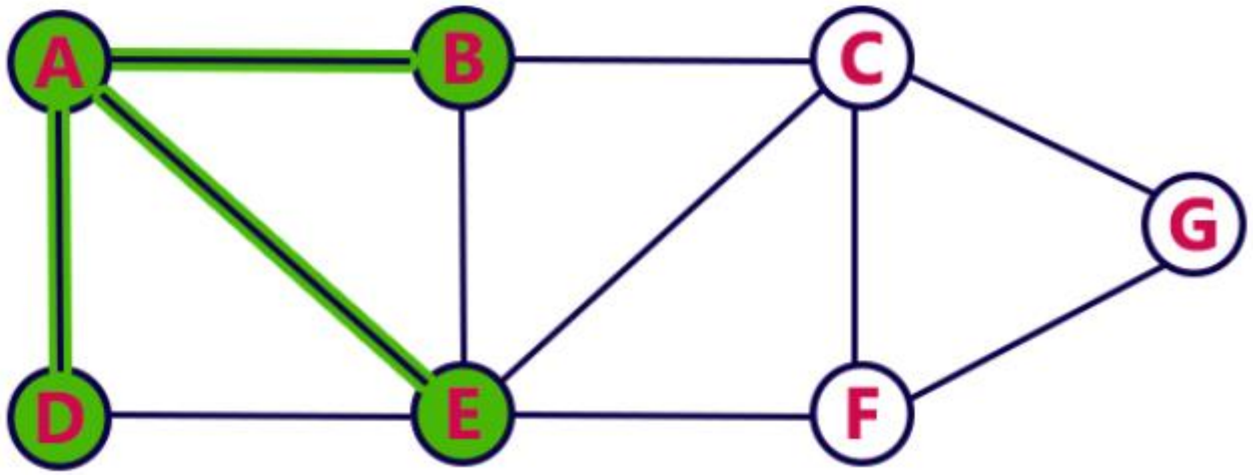
- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

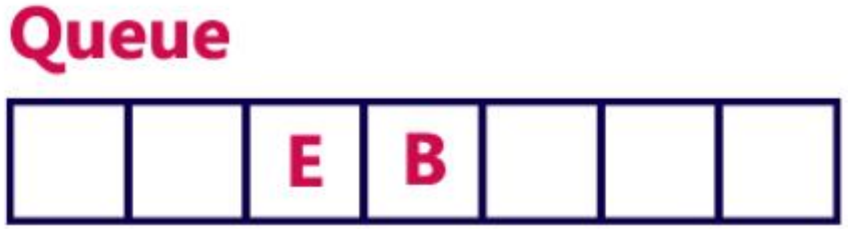




## Step 3:

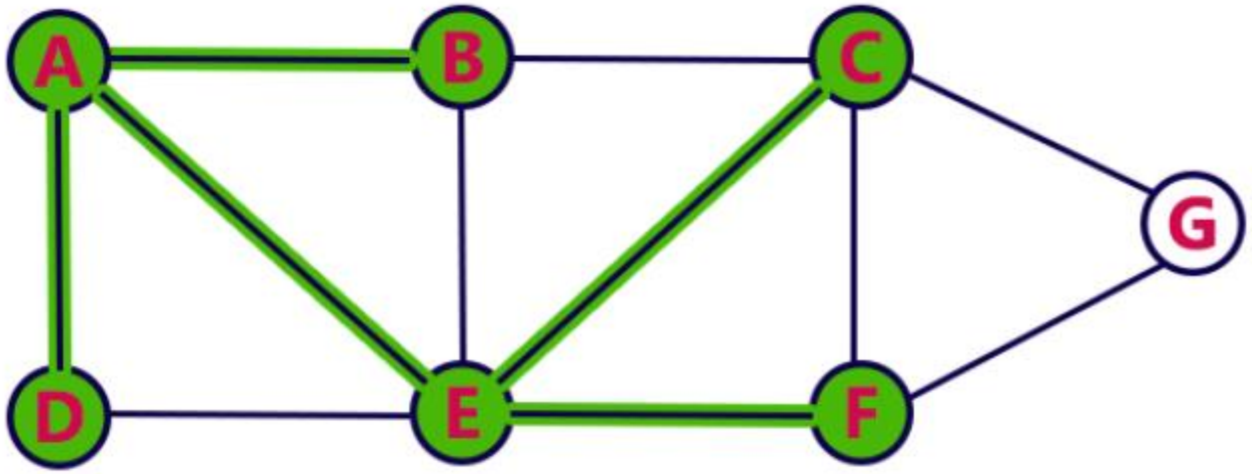
- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

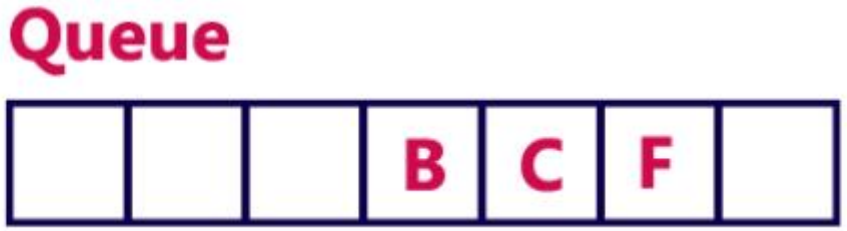




**Step 4:**

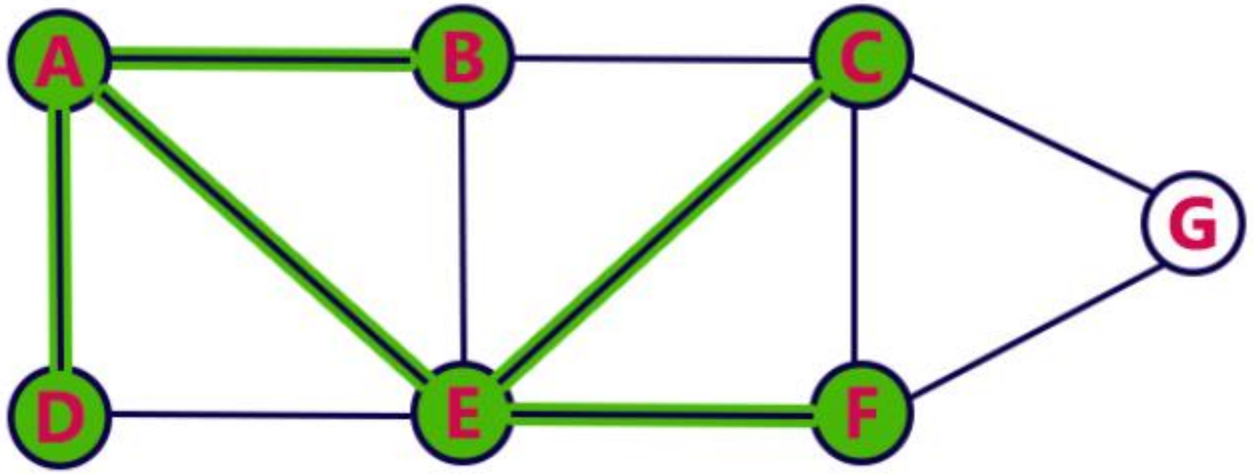
- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

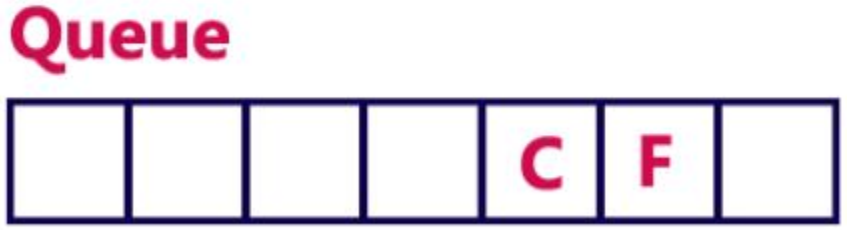




## Step 5:

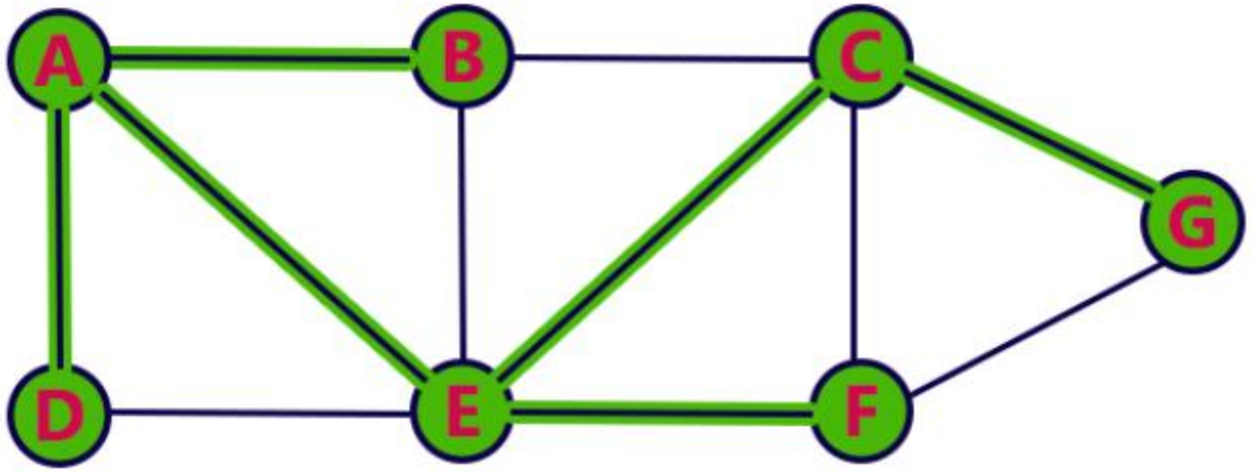
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

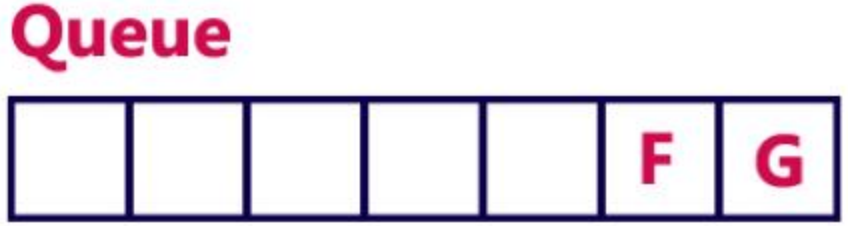




**Step 6:**

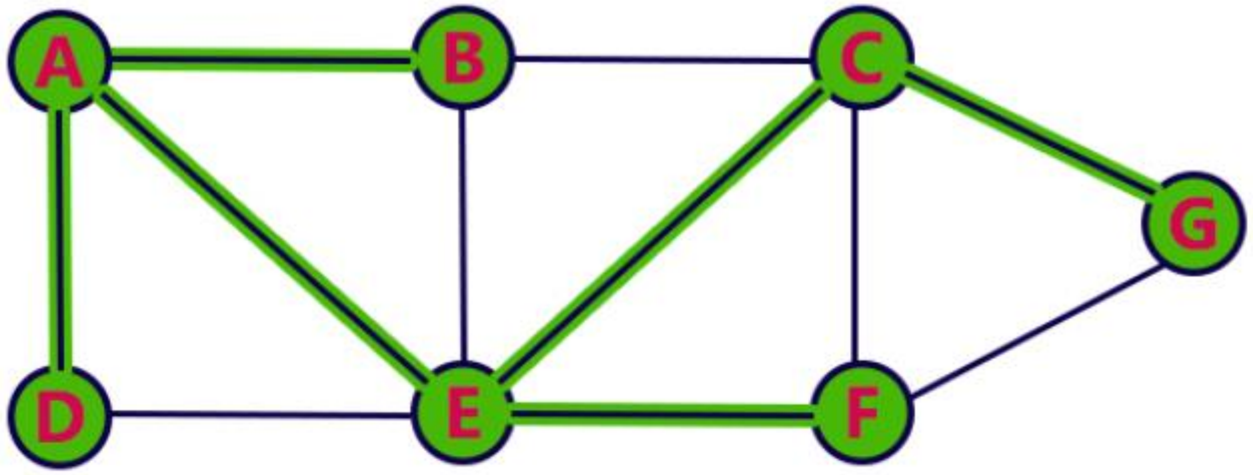
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

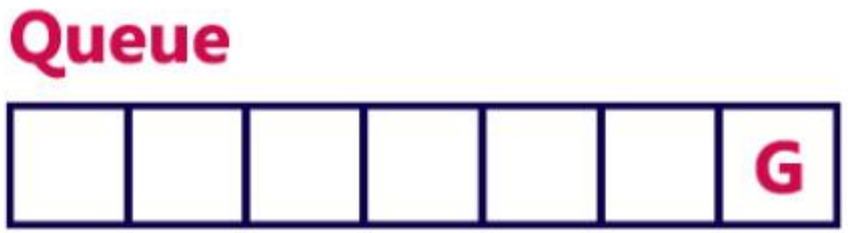




## Step 7:

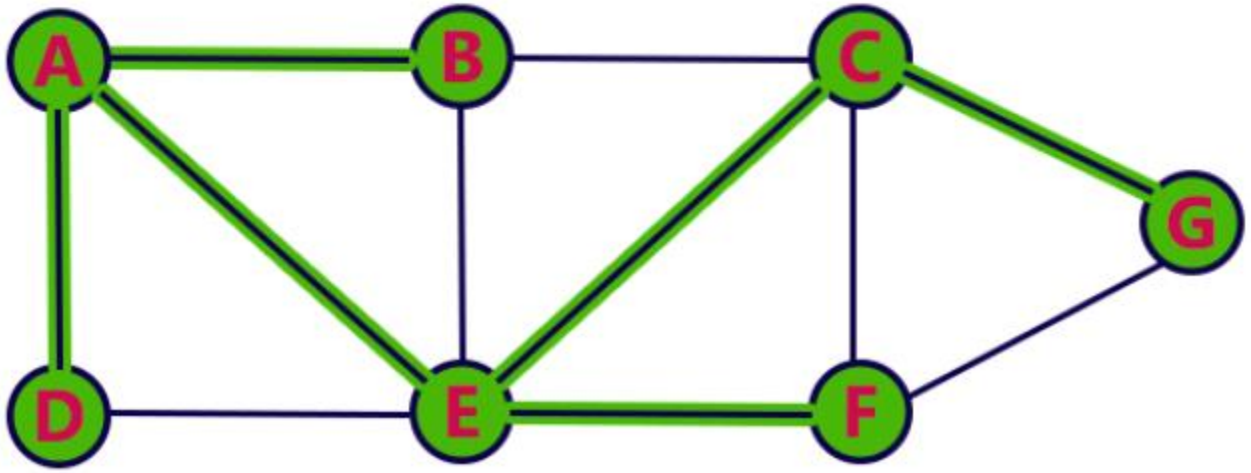
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



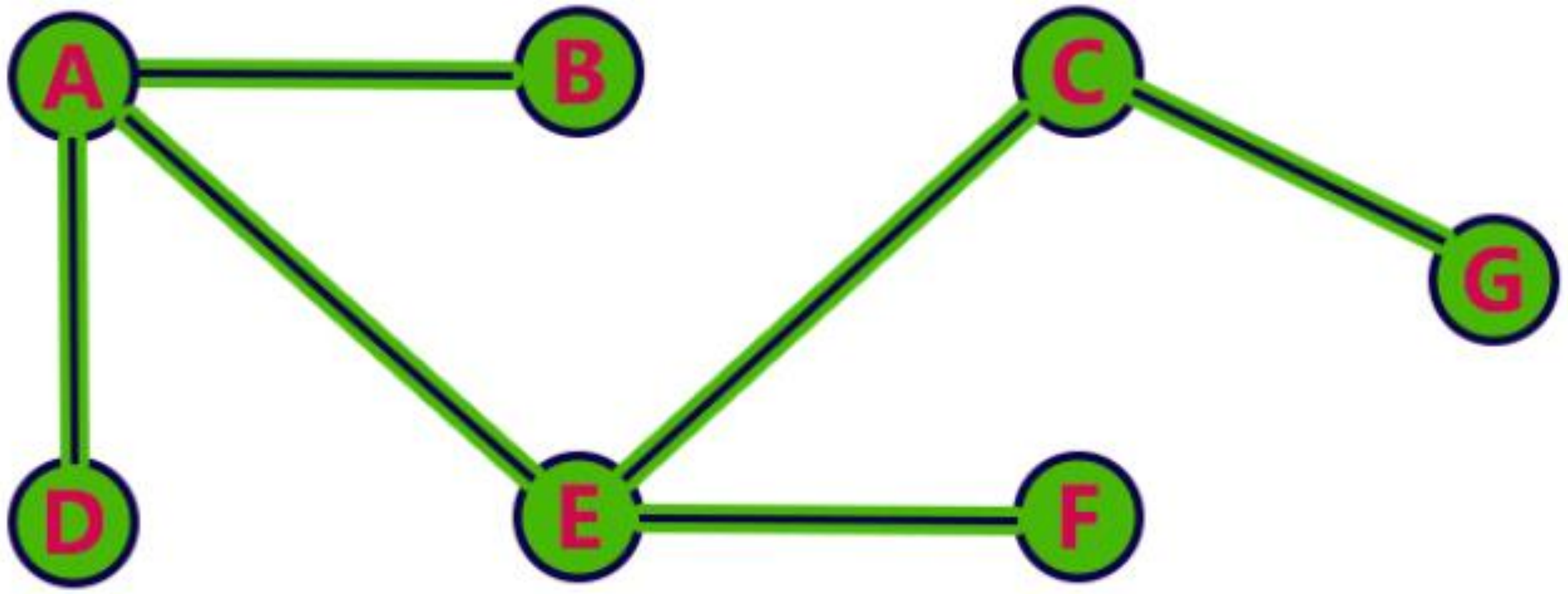


## Step 8:

- 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...



## Graph

- A graph can be defined as a group of vertices and edges to connect these vertices. The types of graphs are given as follows -

**1. Undirected graph:** An undirected graph is a graph in which all the edges do not point to any particular direction, i.e., they are not unidirectional; they are bidirectional. It can also be defined as a graph with a set of V vertices and a set of E edges, each edge connecting two different vertices.

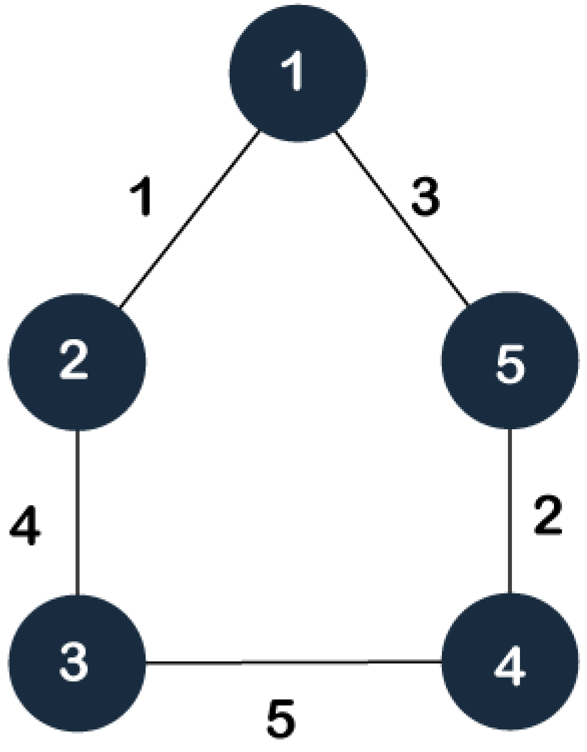
**2. Directed graph:** Directed graphs are also known as digraphs. A graph is a directed graph (or digraph) if all the edges present between any vertices or nodes of the graph are directed or have a defined direction.

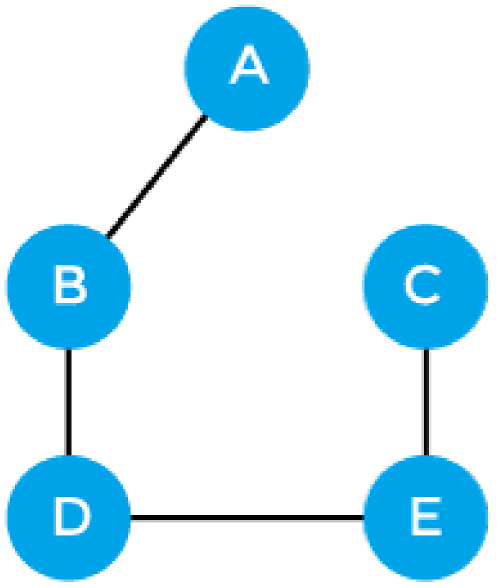
**3. Connected graph:** A connected graph is a graph in which a path always exists from a vertex to any other vertex. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction.

## Spanning Tree

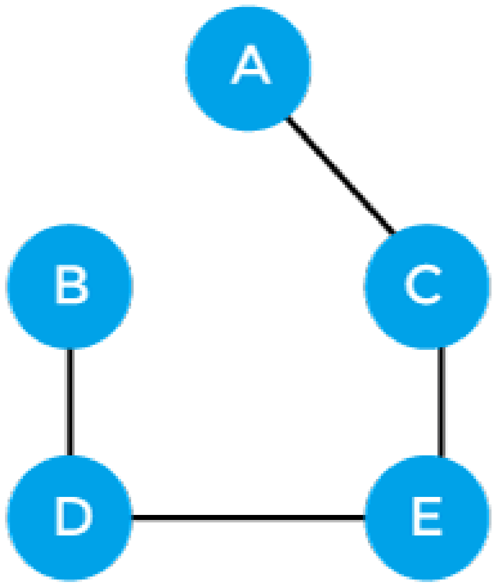
- A spanning tree can be defined as the subgraph of an undirected connected graph.
- It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree.
- A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

## Example

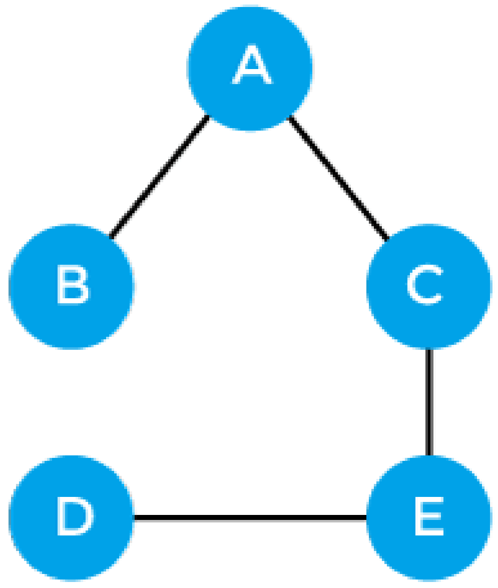




Spanning tree 1



Spanning tree 2

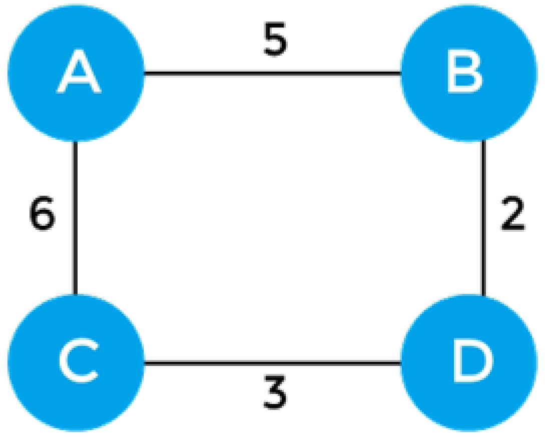


Spanning tree 3

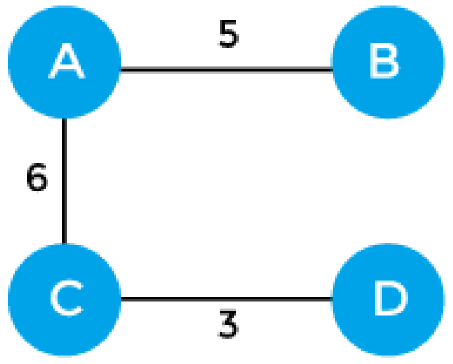
## Properties

- There can be more than one spanning tree of a connected graph G.
- A spanning tree does not have any cycles or loop.
- A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
- A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
- There can be a maximum $\mathbf{n^{n-2}}$ number of spanning trees that can be created from a complete graph.
- A spanning tree has **n-1** edges, where 'n' is the number of nodes.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.

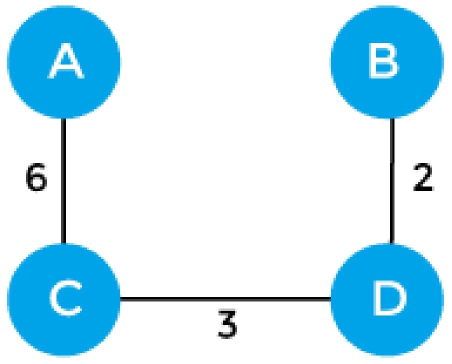
# Minimum Spanning Tree



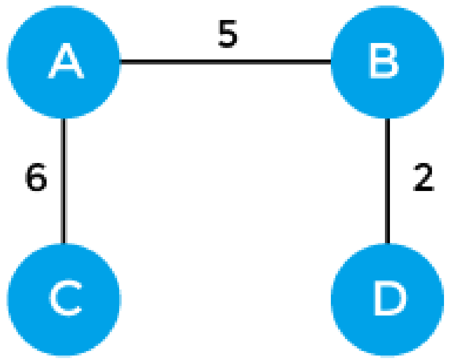
Weighted graph



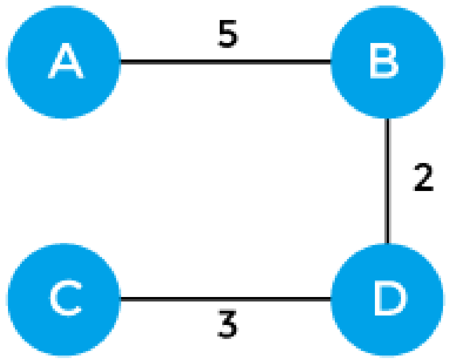
Sum = 14
Minimum spanning tree - 1



Sum = 11
Minimum spanning tree - 2



Sum = 13
Minimum spanning tree - 3



Sum = 10
Minimum spanning tree - 4

## Applications

- Minimum spanning tree can be used to design water-supply networks, telecommunication networks, and electrical grids.
- It can be used to find paths in the map.

## Prim’s Algorithm

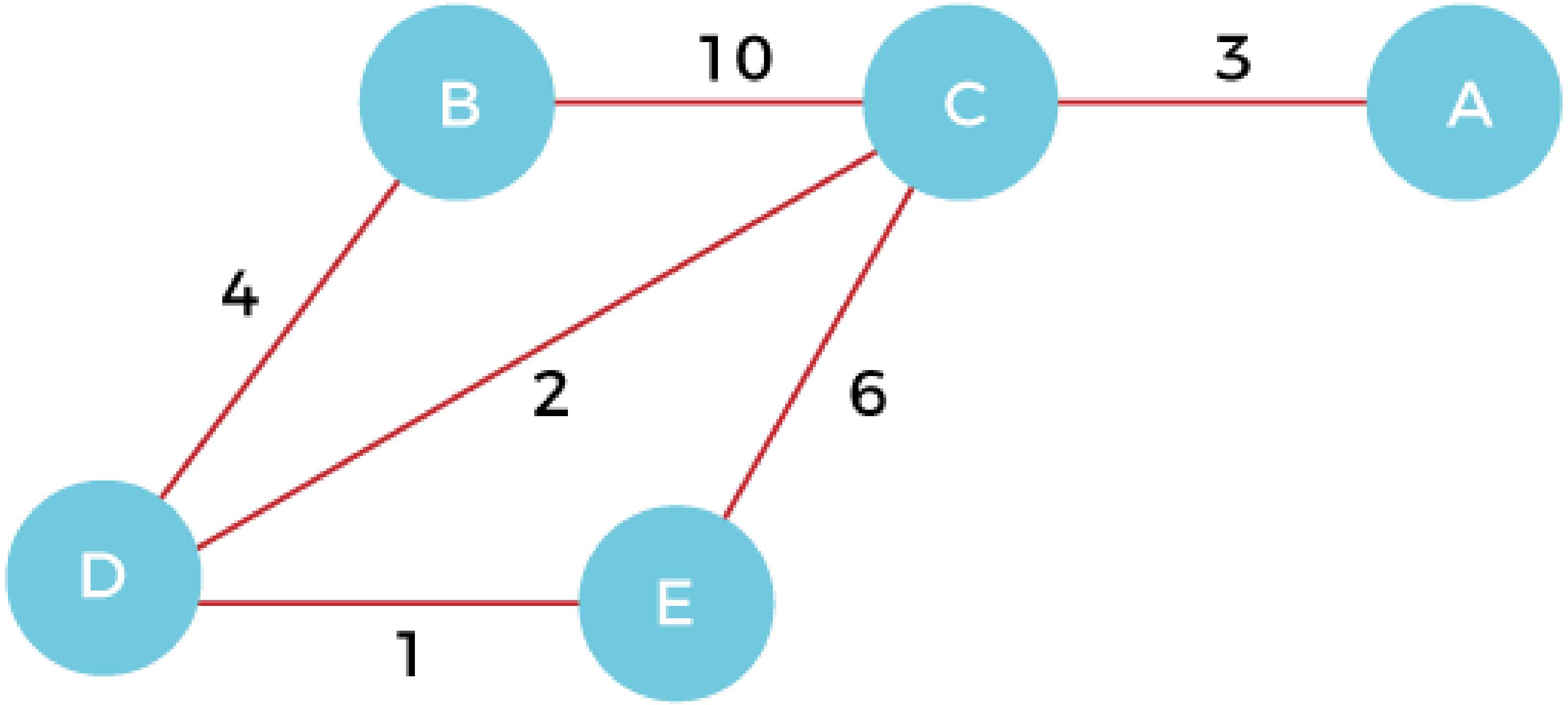
- **Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph.

- Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

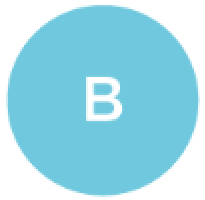
## Steps

- First, we have to initialize an MST with the randomly chosen vertex.
- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed.

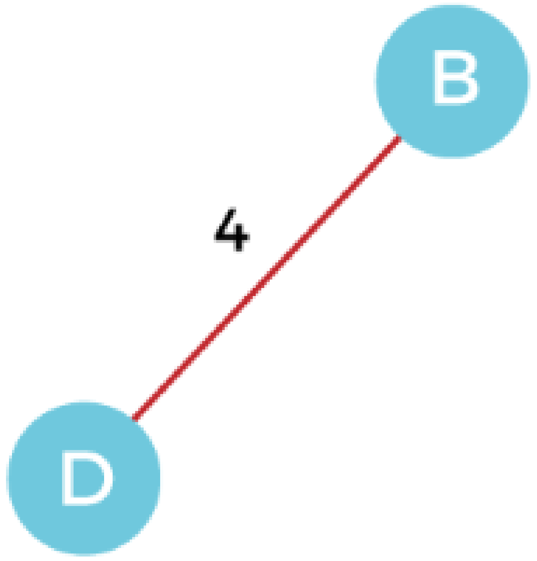
## Example



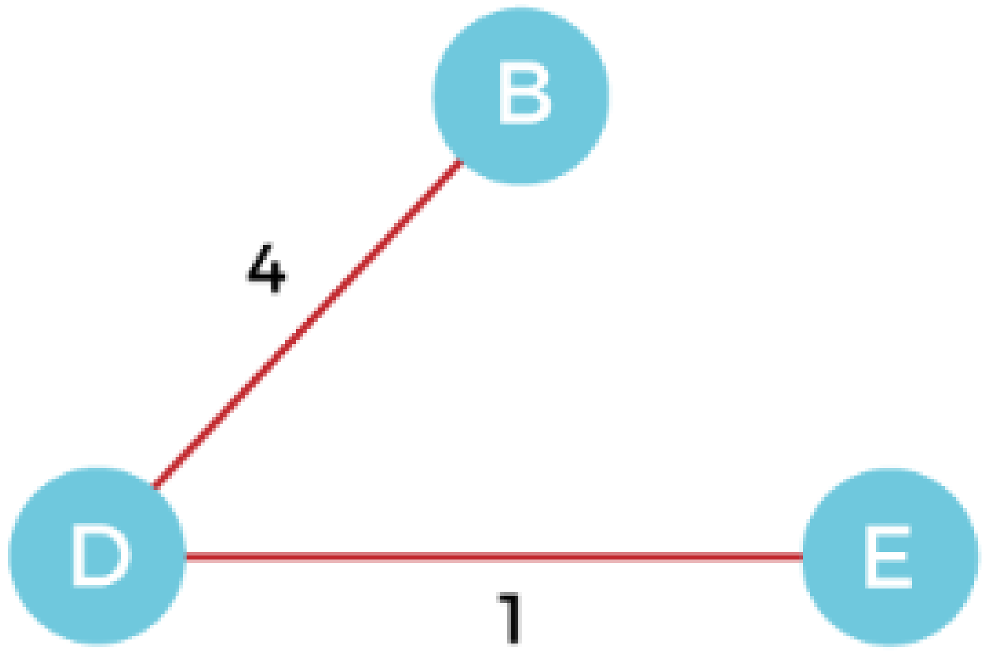
## Step 1



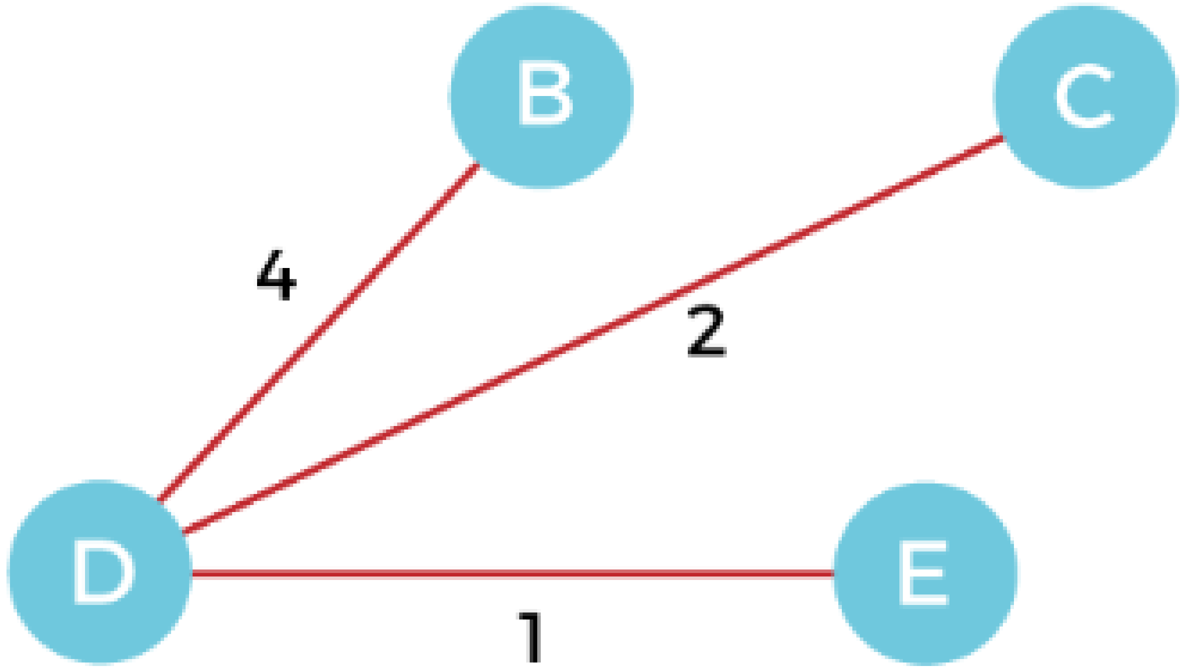
## Step 2



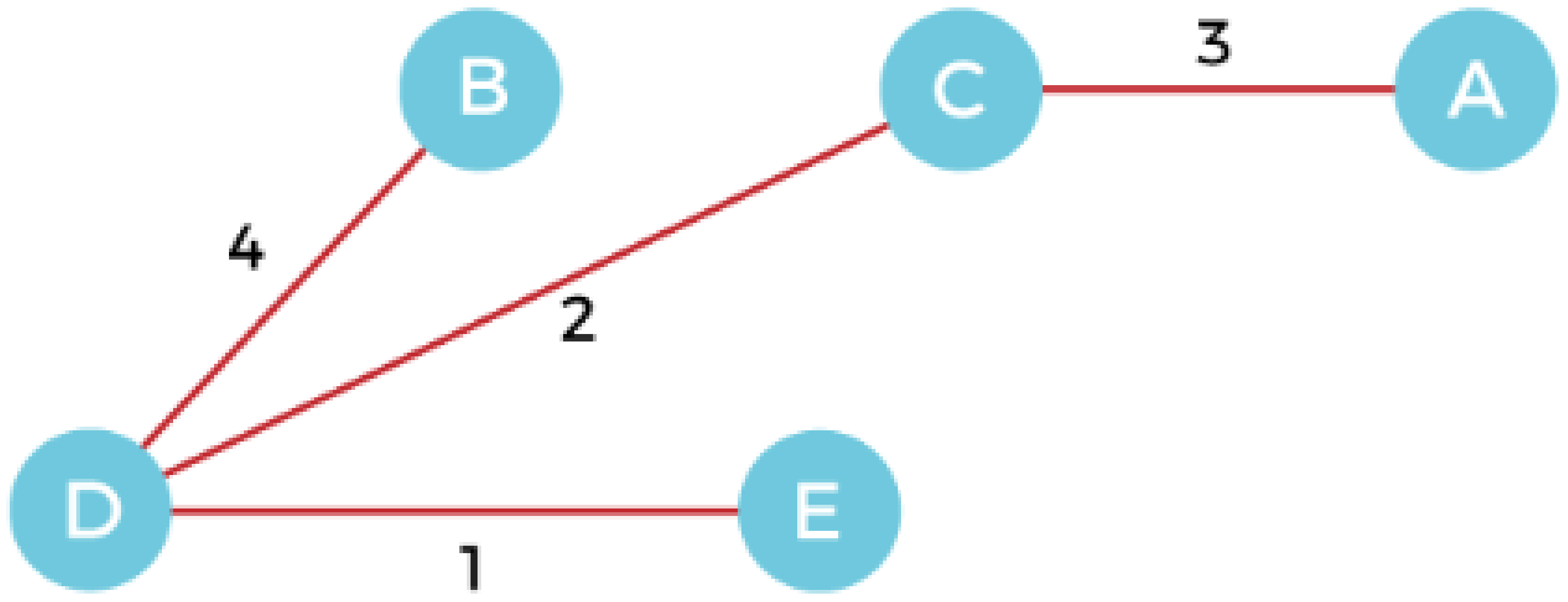
## Step 3



## Step 4



## Step 5



## Cost

- Cost of MST = 4 + 2 + 1 + 3 = 10 units.

## Algorithm

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices
3. Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
4. Step 4: Add the selected edge and the vertex to the minimum spanni ng tree T
5. [END OF LOOP]
6. Step 5: EXIT

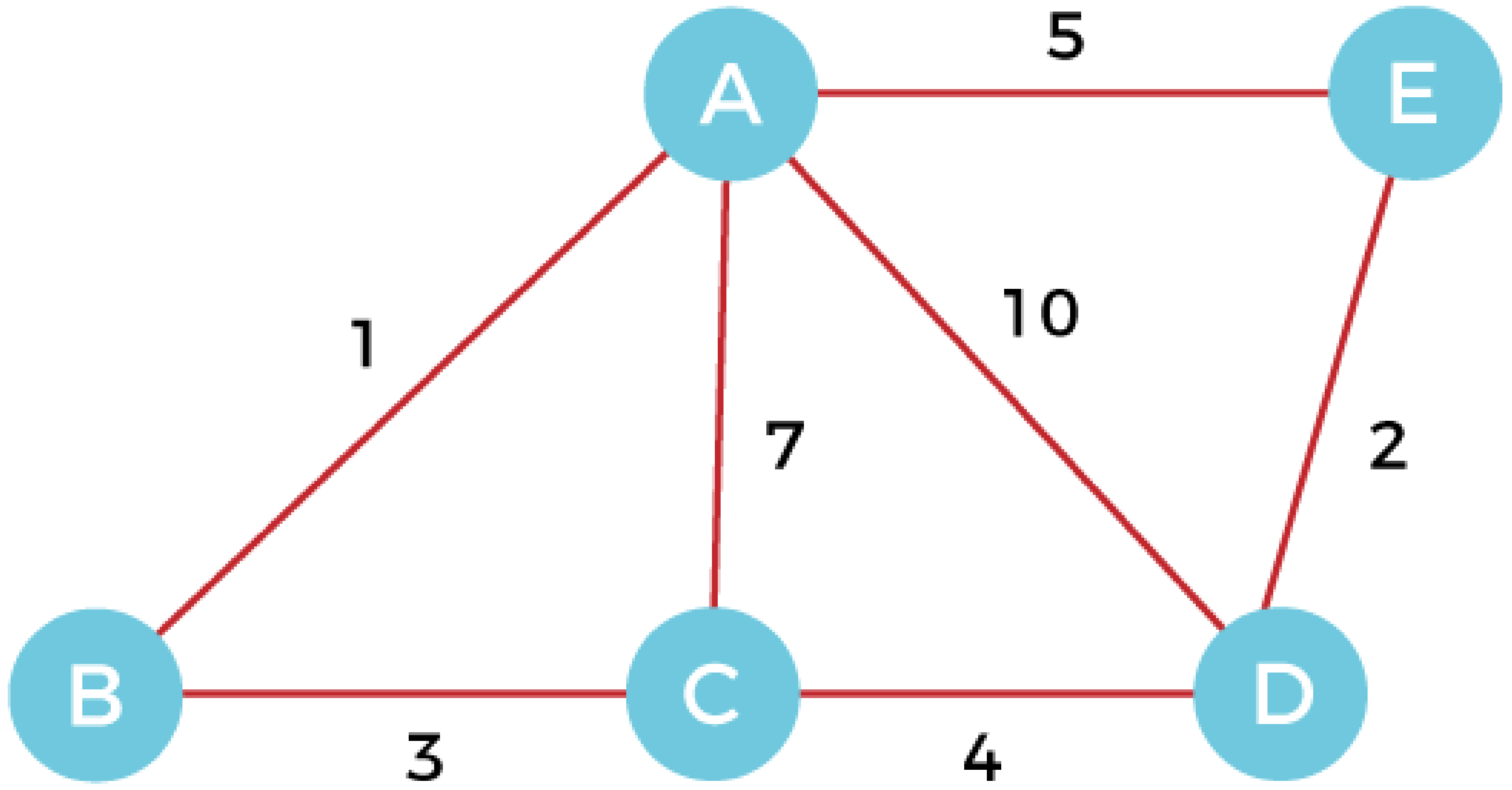
## Kruskal’s Algorithm

- **Kruskal's Algorithm** is used to find the minimum spanning tree for a connected weighted graph.
- The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph.
- It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

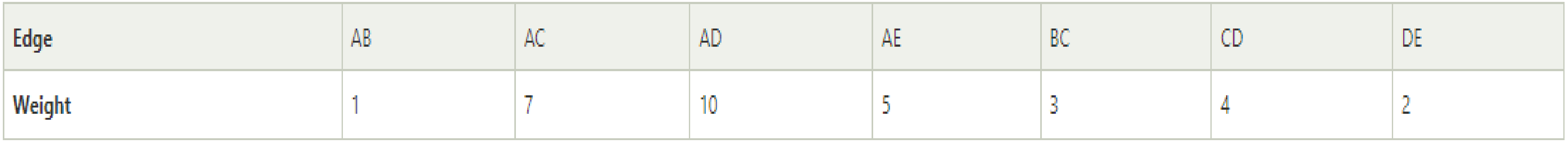
## Steps

- First, sort all the edges from low weight to high.
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

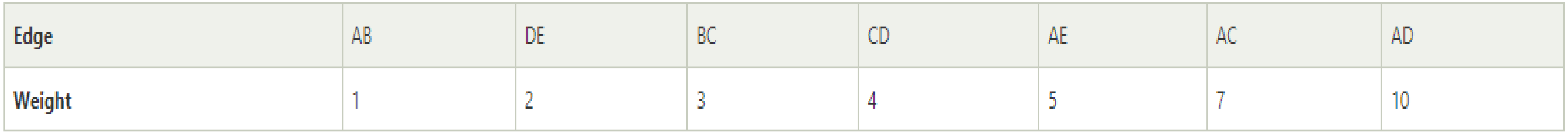
## Example



## Step 1

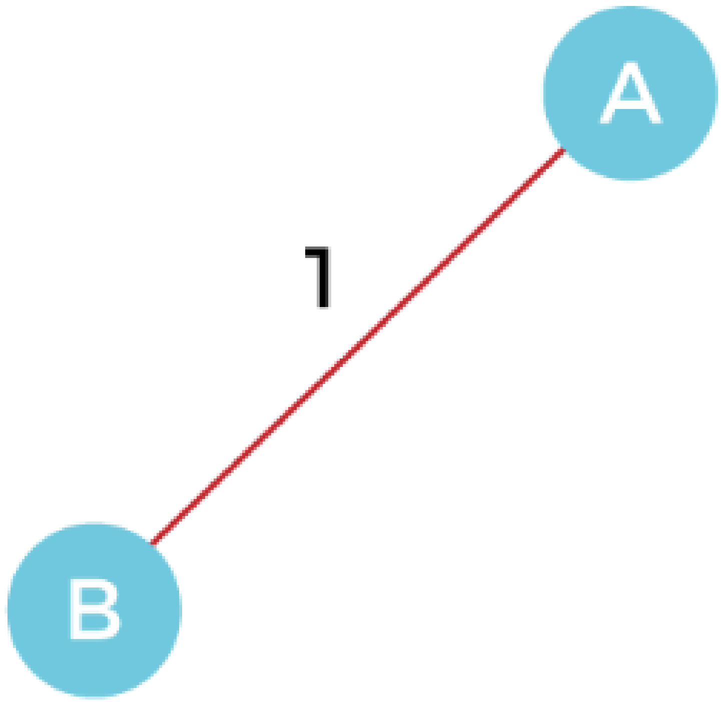


| Edge | AB | AC | AD | AE | BC | CD | DE |
|---|---|---|---|---|---|---|---|
| Weight | 1 | 7 | 10 | 5 | 3 | 4 | 2 |

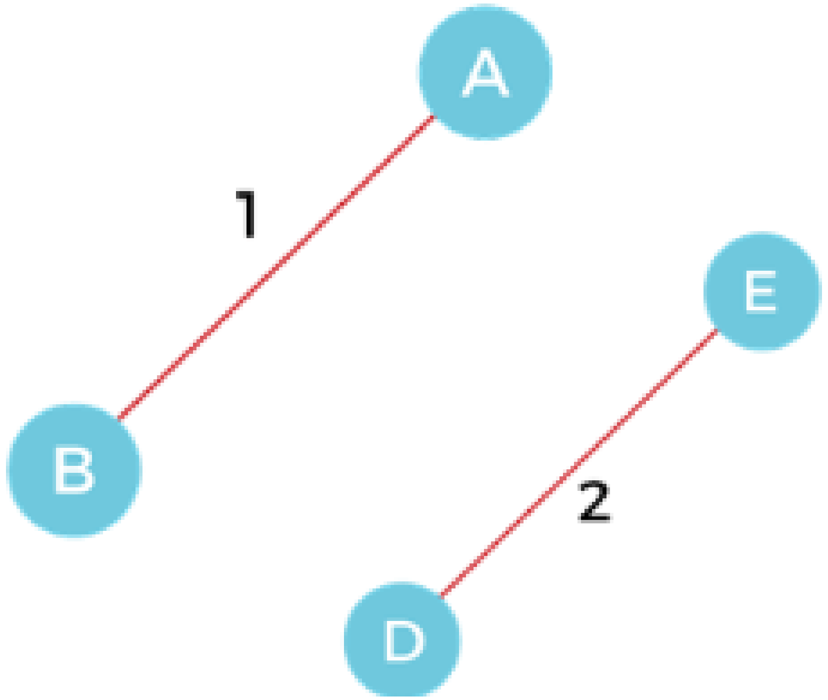


| Edge | AB | DE | BC | CD | AE | AC | AD |
|---|---|---|---|---|---|---|---|
| Weight | 1 | 2 | 3 | 4 | 5 | 7 | 10 |

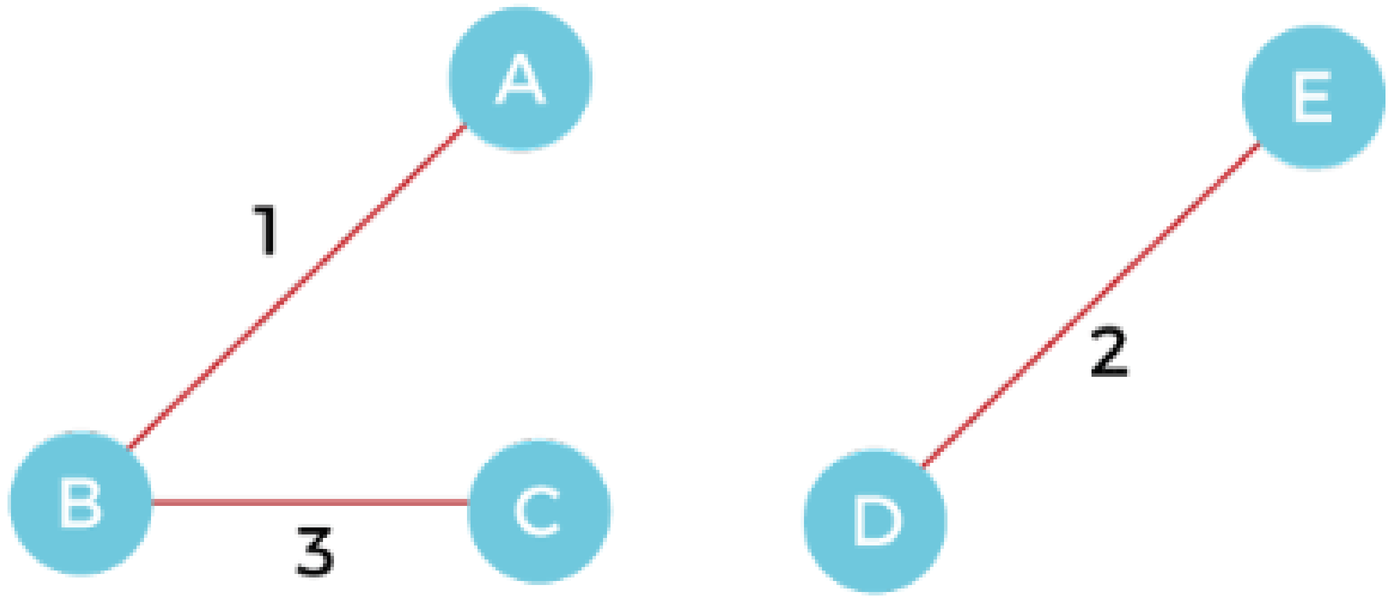
## Step 2



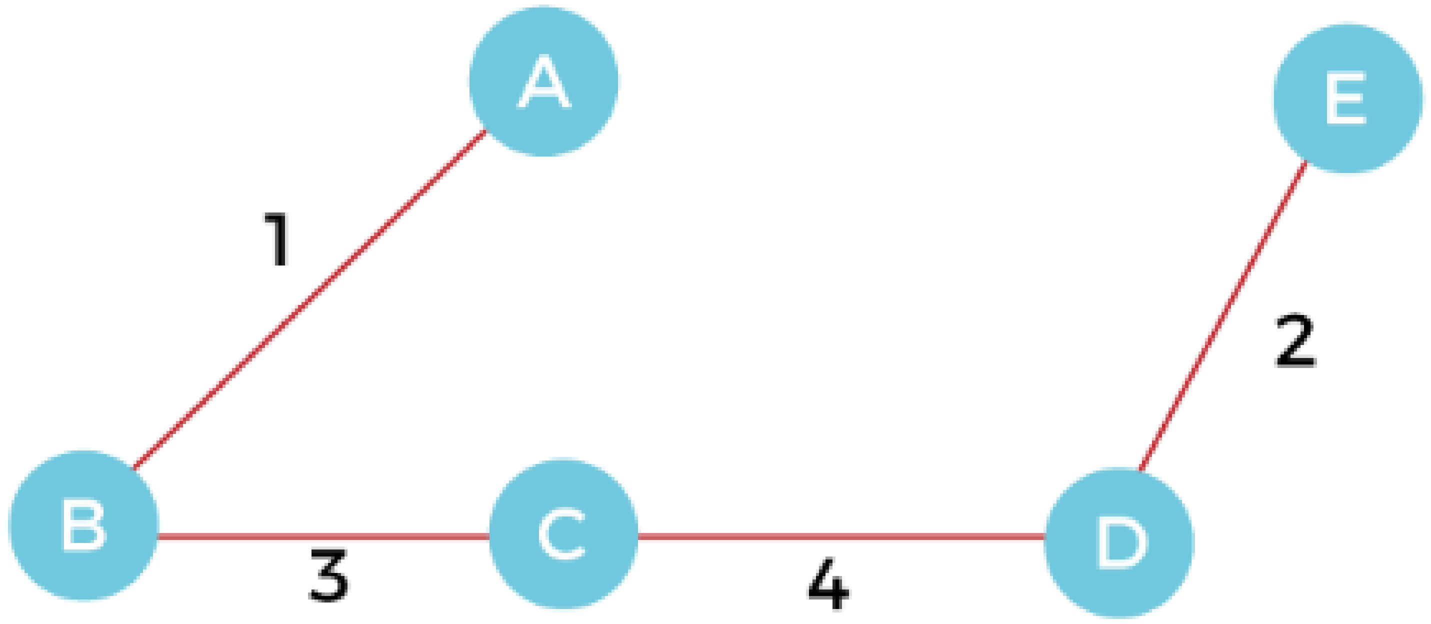
## Step 3



## Step 4



## Step 5



## Further

**Step 6 -** After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

**Step 7 -** Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

**Step 8 -** Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

## Cost

- The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

## Algorithm

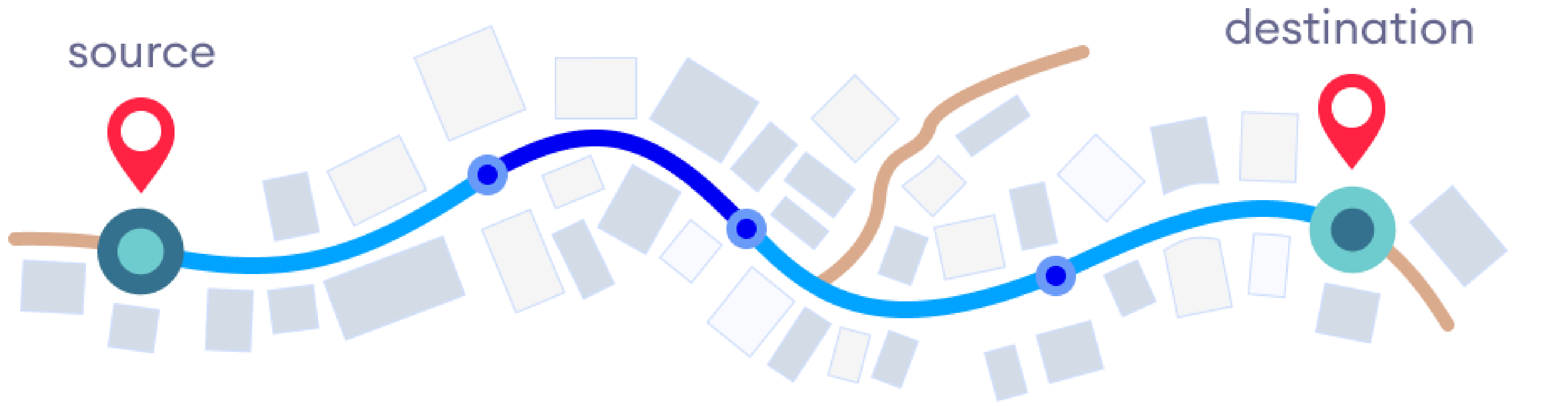
1. Step 1: Create a forest F in such a way that every vertex of the graph is a se parate tree.
2. Step 2: Create a set E that contains all the edges of the graph.
3. Step 3: Repeat Steps 4 and 5 **while** E is NOT EMPTY and F is not spanning
4. Step 4: Remove an edge from E with minimum weight
5. Step 5: IF the edge obtained in Step 4 connects two different trees, then ad d it to the forest F
6. (**for** combining two trees into one tree).
7. ELSE
8. Discard the edge
9. Step 6: END

## Dijkstra’s Algorithm

- Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.
- It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

## Central Idea

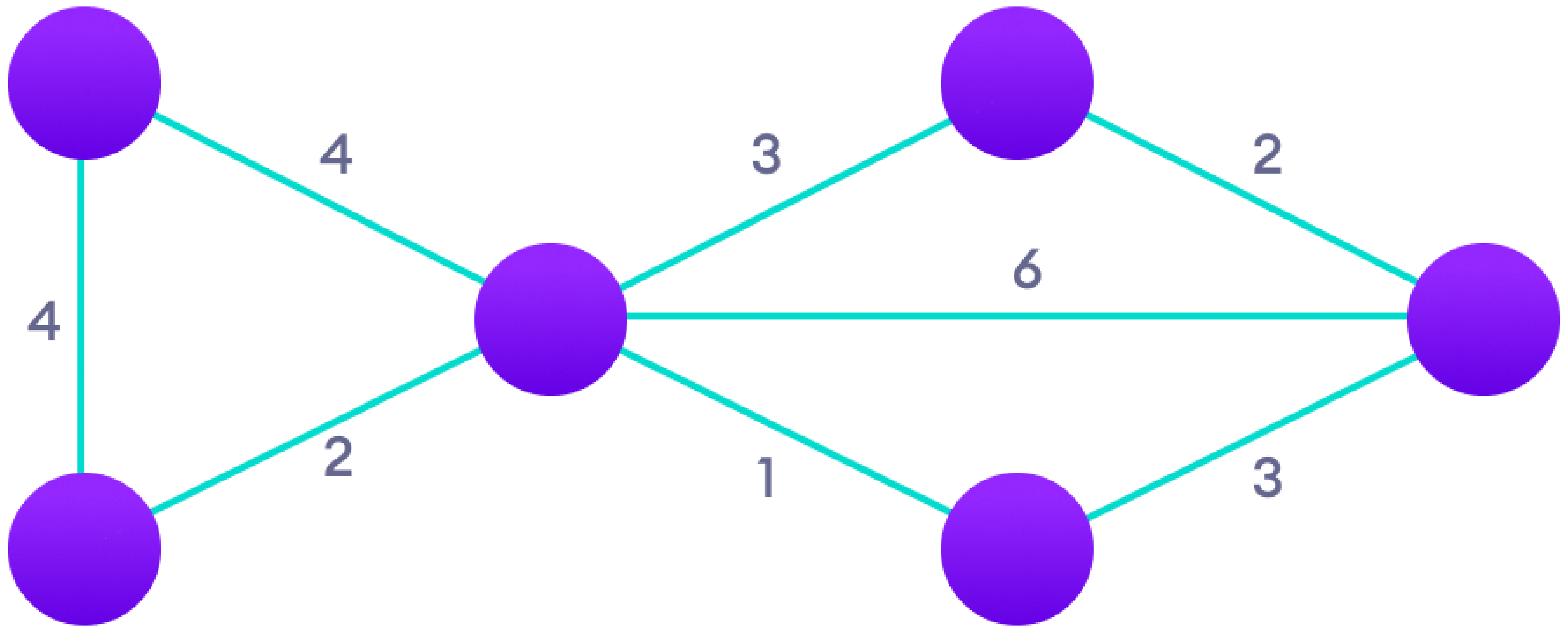
- Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.



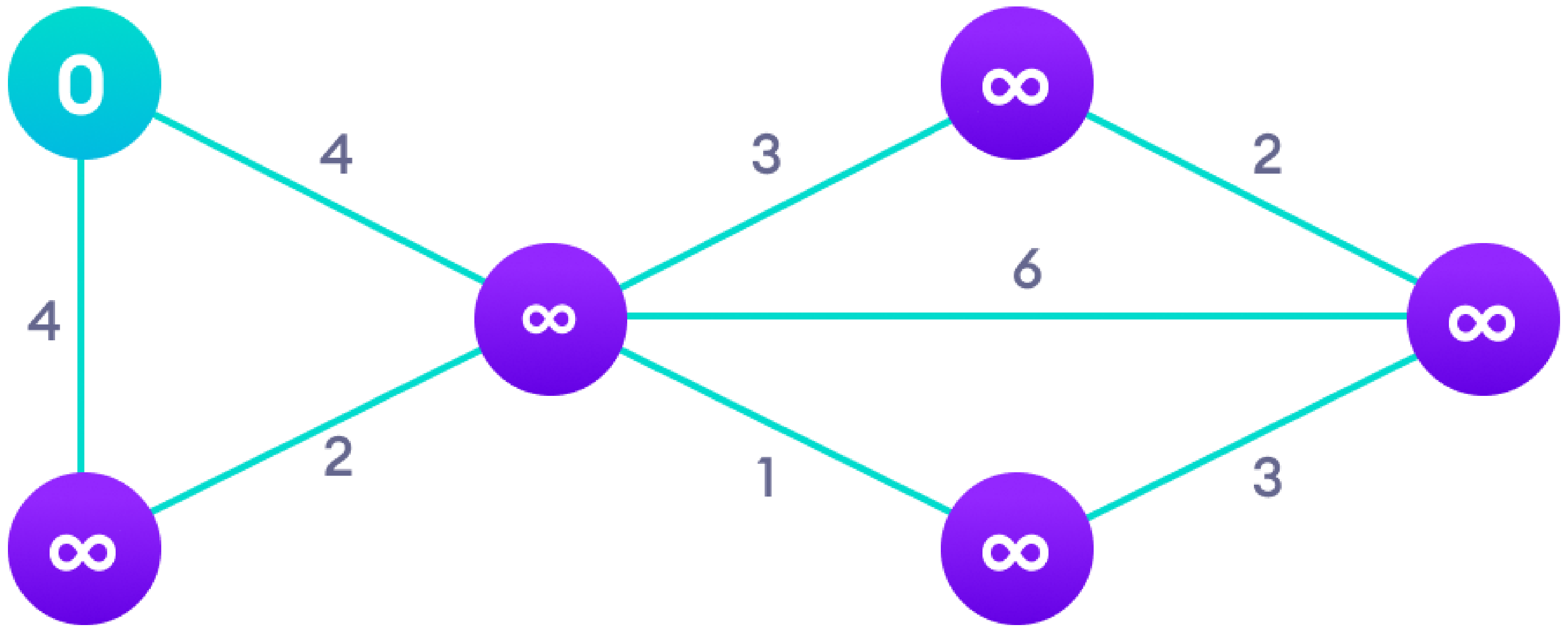
🟦 the shortest path between the source and destination

🟦 a subpath which is also the shortest path between its source and destination

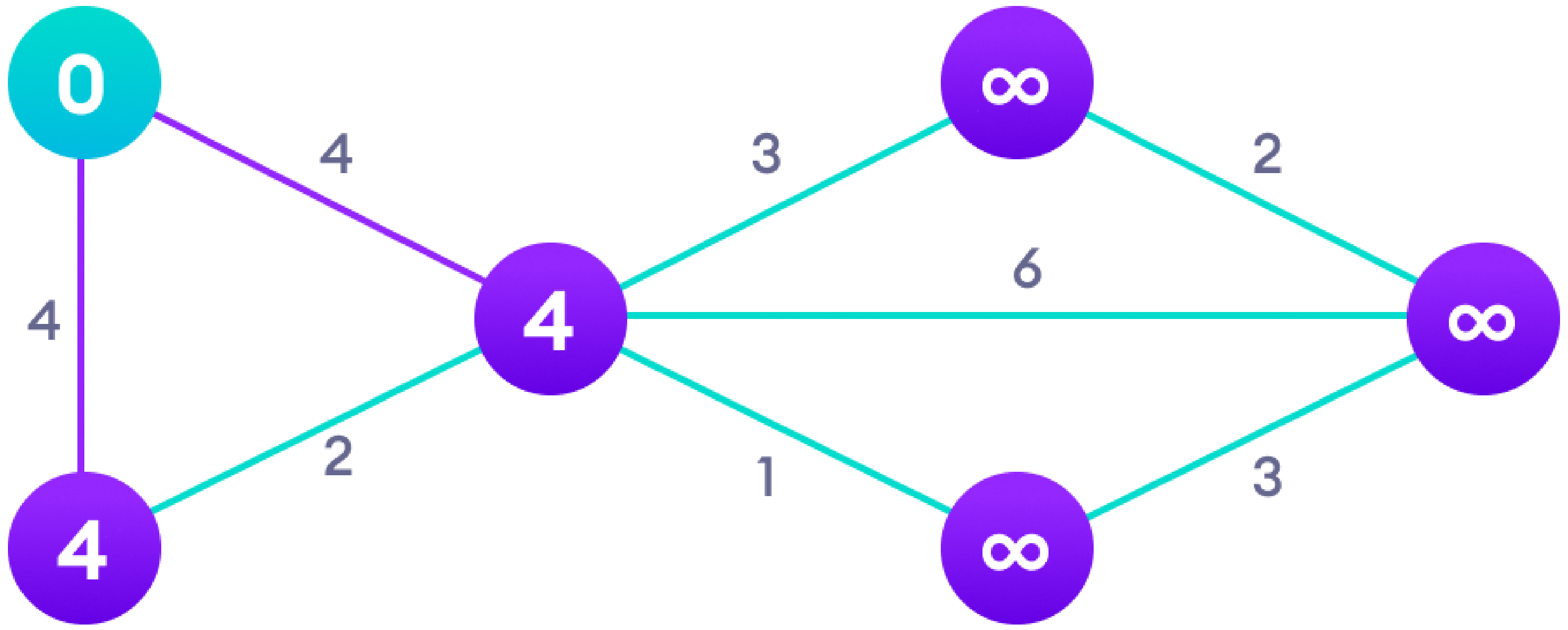
## Example



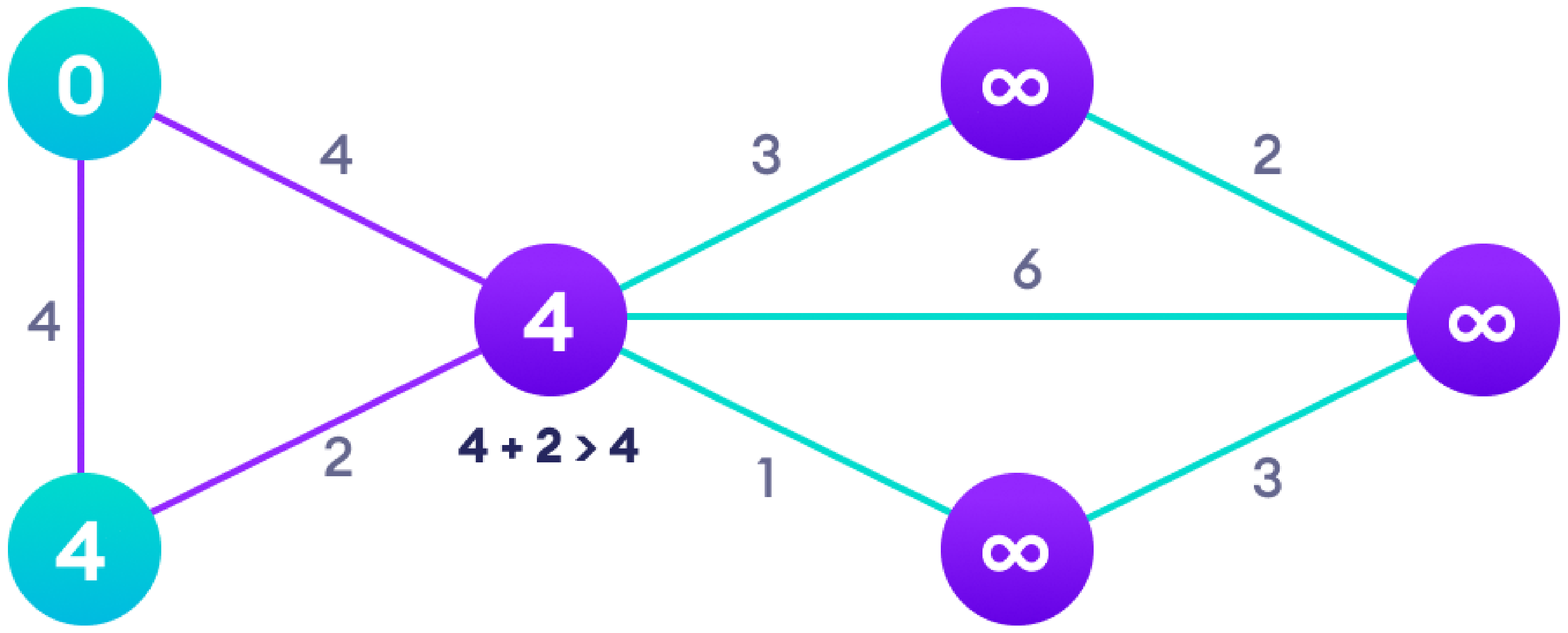
Step: 1



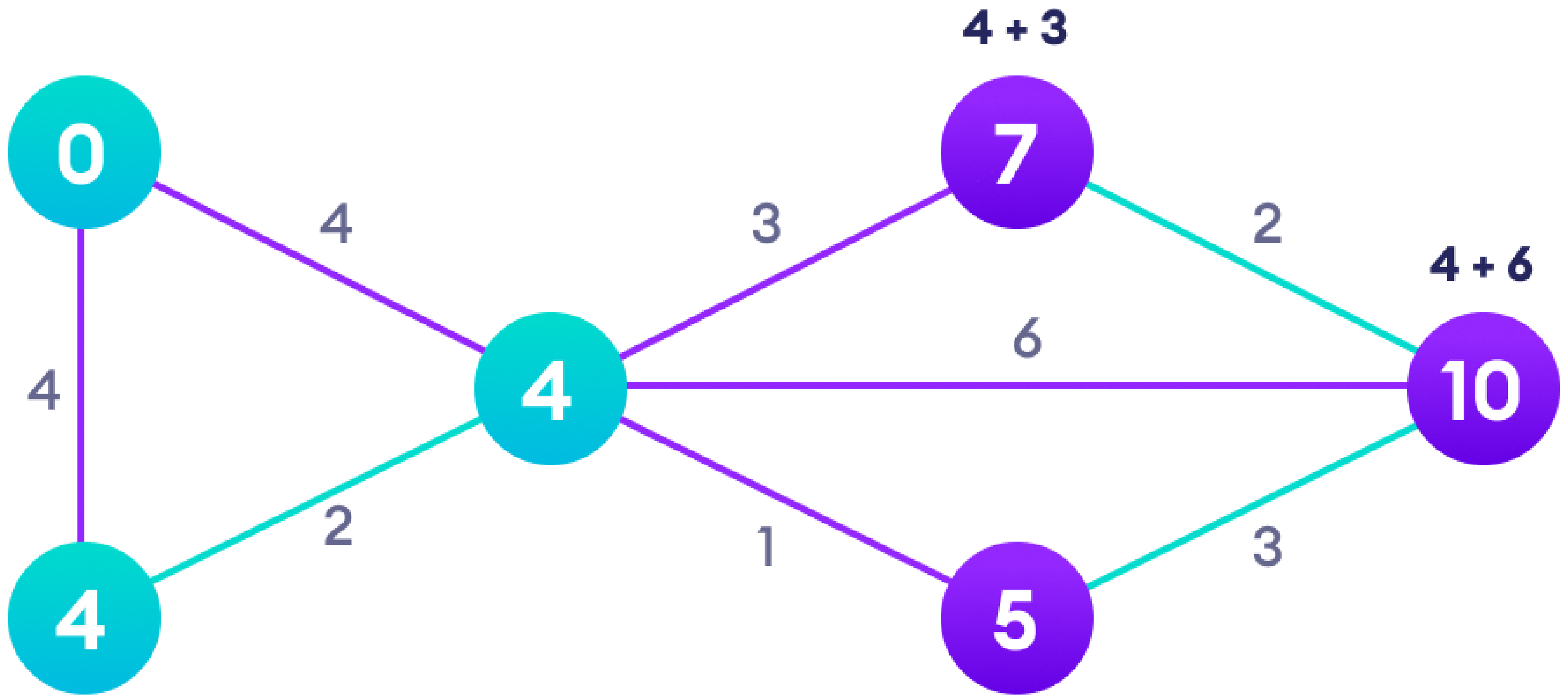
Step: 2



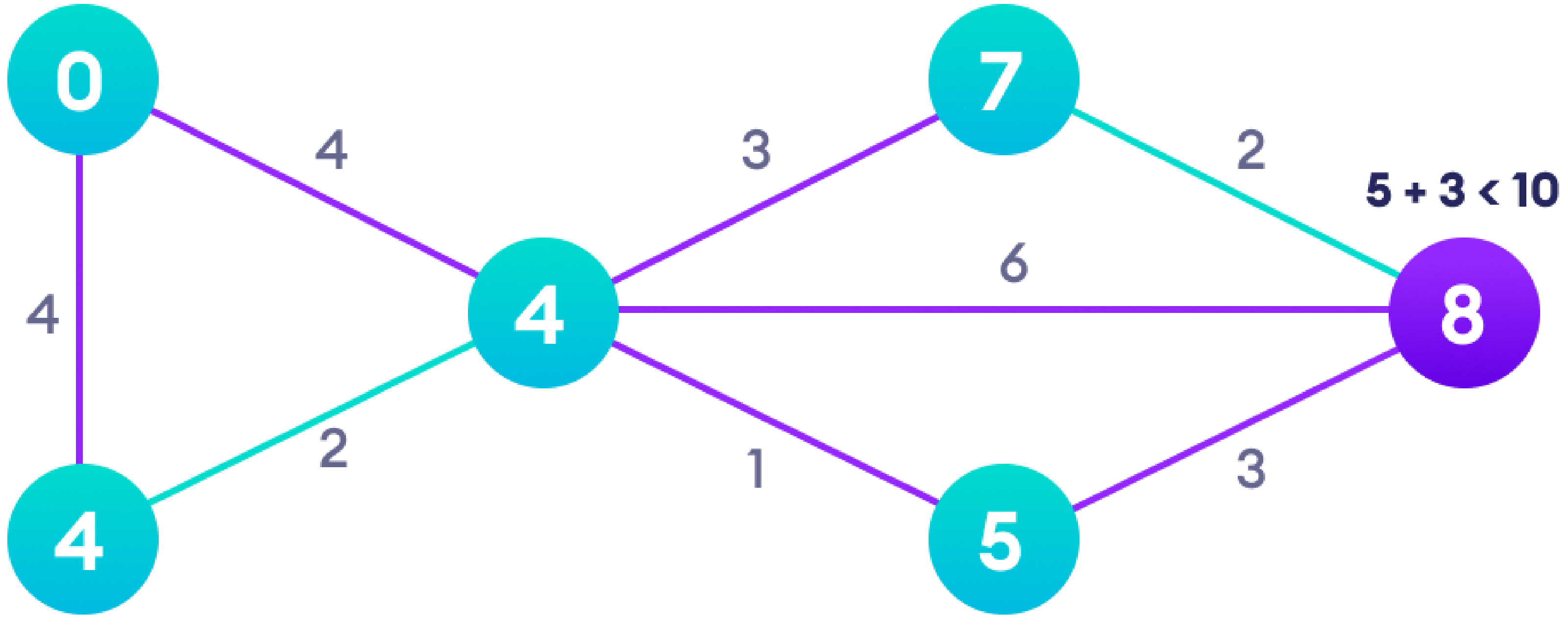
Step: 3



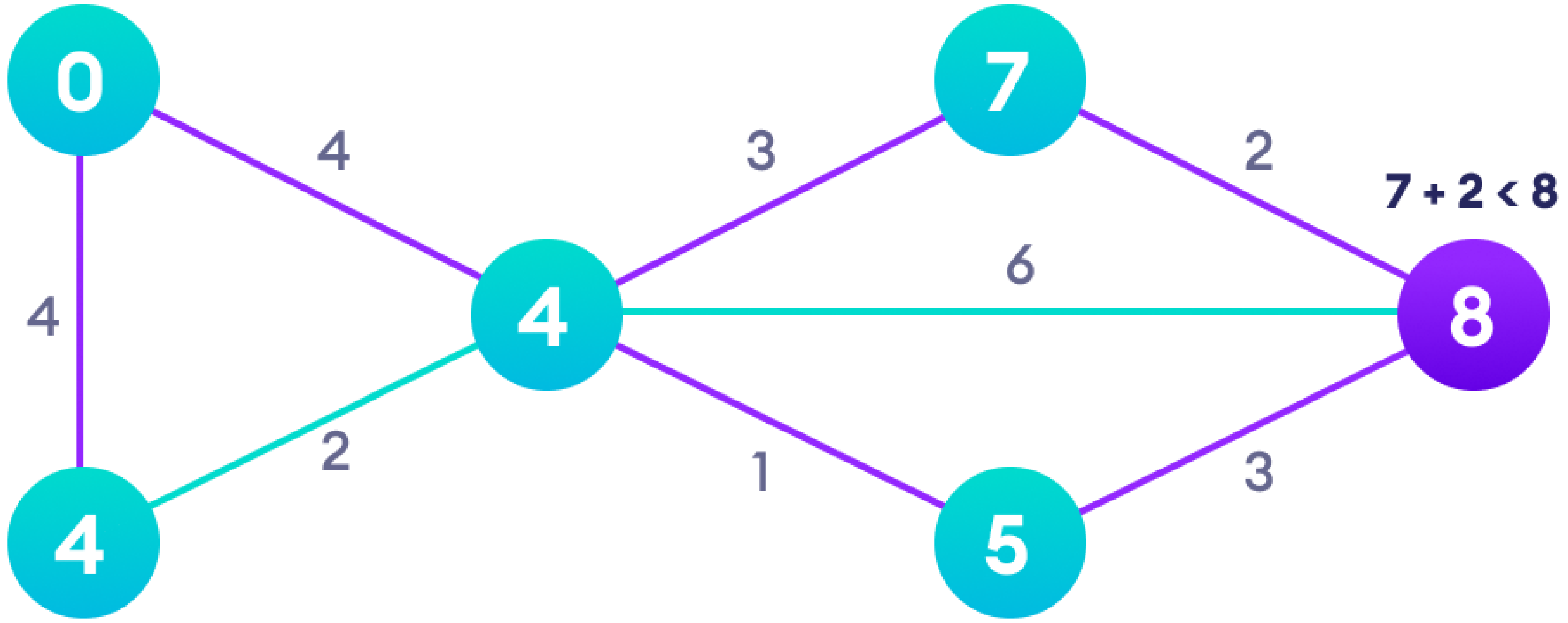
Step: 4



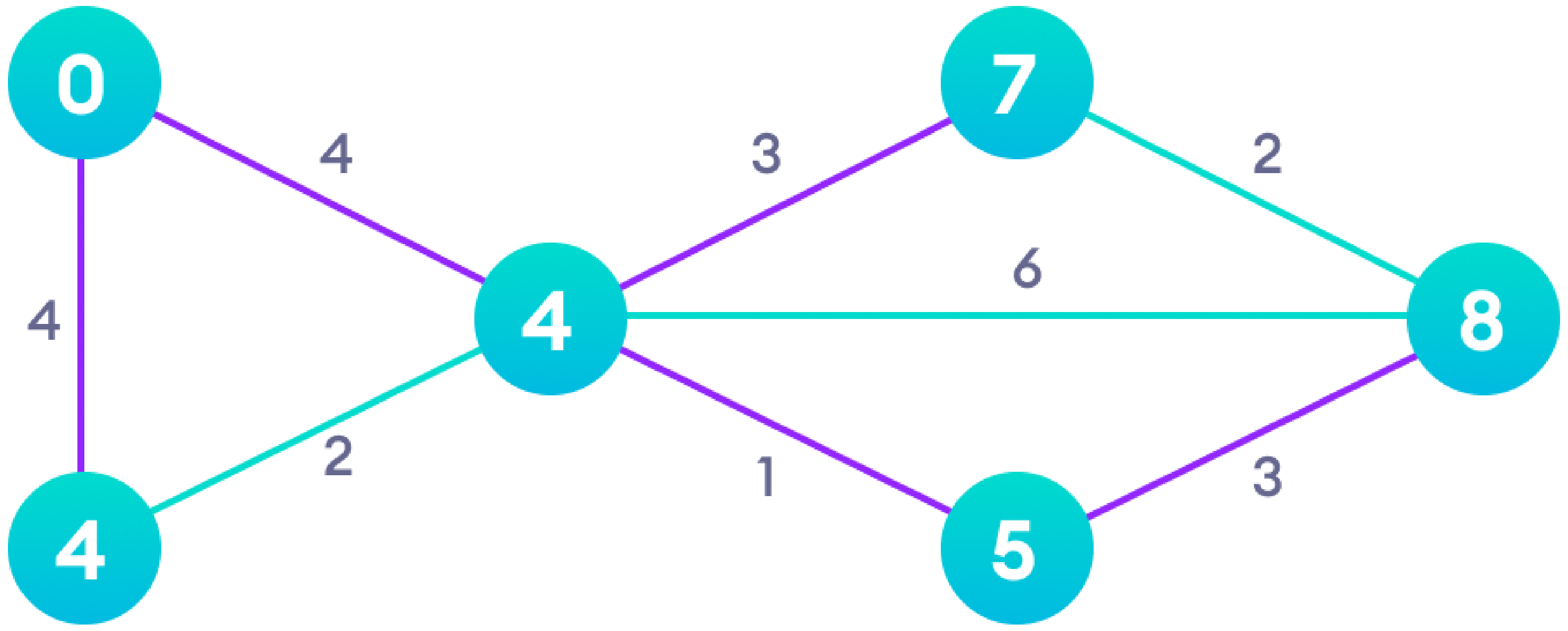
Step: 5



Step: 6



Step: 7



Step: 8