UNIT-IV

Graphs-Basic terminology, Representation of Graphs: Sequential and Linked representation.

Graph Traversals-Breadth First Search, Depth First Search with algorithms and implementation.

Spanning Trees - Definition and properties, Minimum Spanning Tree, Exploring Minimum Spanning Tree Algorithms: Implementation of Prim's and Kruskal's.

Implementation of Dijkstra Algorithms for finding shortest path in graphs..

****

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. A graph is defined as follows...

Graph is a collection of vertices and arcs in which vertices are connected with arcs

Graph is a collection of nodes and edges in which nodes are connected with edges

Generally, a graph **G** is represented as **G = (V , E)**,

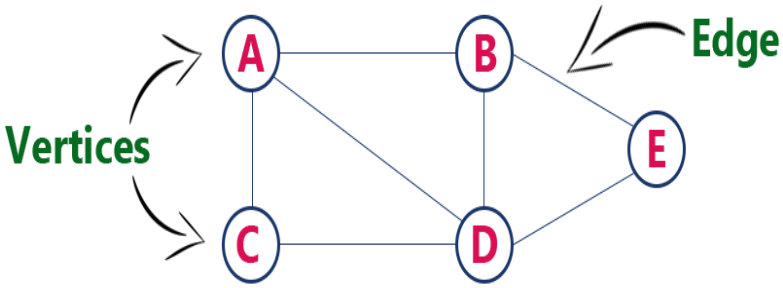
where **V is set of vertices** and **E is set of edges**.

Example:1

The following is a graph with 5 vertices and 6 edges.

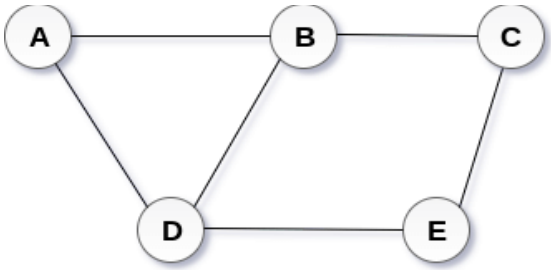
This graph G can be defined as G = (V , E)

Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Example:2

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



Graph Terminology

We use the following terms in graph data structure...

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

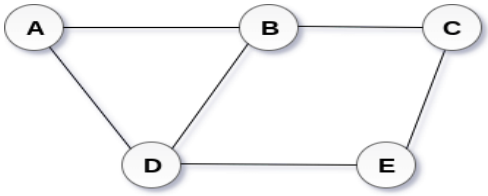
An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (starting Vertex, ending Vertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example:1 graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

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.

Undirected Graph:

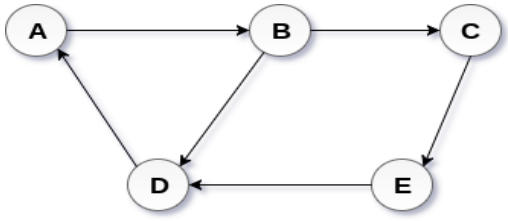
A graph with only undirected edges is said to be undirected graph.



Undirected Graph

Directed Graph:

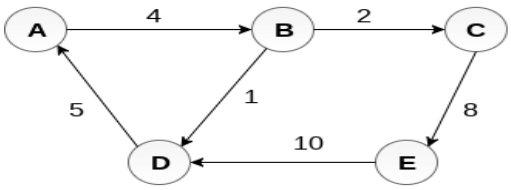
A graph with only directed edges is said to be directed graph.



Directed Graph

Weighted directed graph:

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.



Weighted Directed Graph

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.

- **Adjacent**
  - If there is an edge between vertices A and B then both A and B are said to be adjacent.

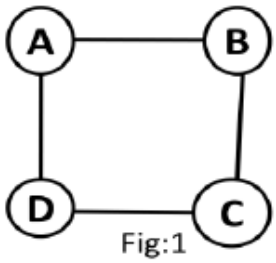


Fig:1

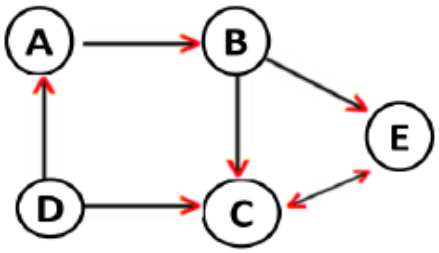


Fig:2

In fig:1 the adjacent list is

Vertex	Adjacent verices
A	B,C
B	A,C
C	B,D
D	A,C

In fig:2 the adjacent list is

Vertex	Adjacent verices
A	B
B	E,C
C	E
D	A,C
E	C

Incident

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

Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge

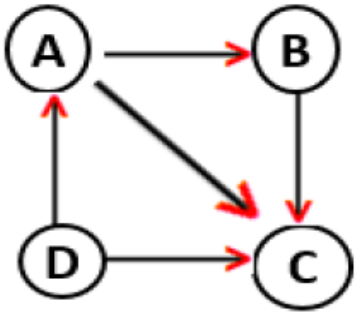
A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex. A node with degree 0 is called as isolated node.

- **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.

Example 1:



In given above given graph the degree are as follows

Vertex	Degree	Indegree	Outdegree
A	3	1	2
B	2	1	1
C	c	2	0
D	2	0	2

Self-loop

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

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.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain **n(n-1)/2** edges where n is the number of nodes in the graph.

Graph Representation

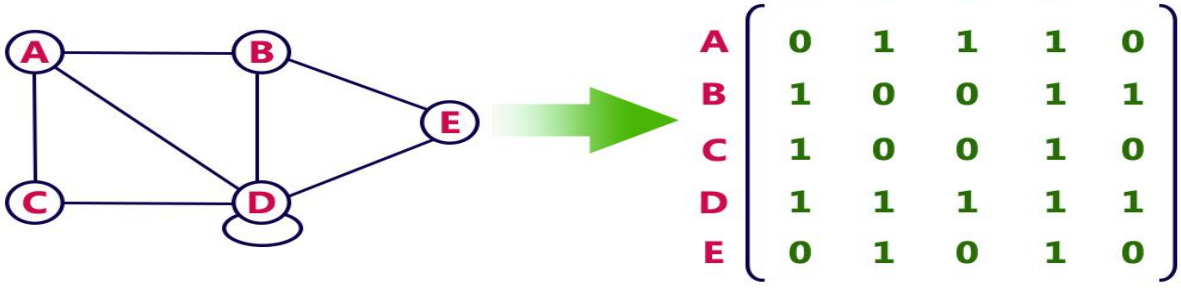
Graph data structure is represented using following representations...

1. **Adjacency Matrix (Sequential Representation)**
2. **Adjacency List (Linked Representation)**
3. **Incidence Matrix**

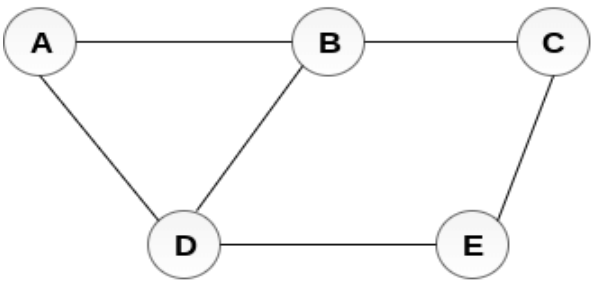
1. Adjacency Matrix (Sequential Representation):

- 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:1- consider the following **undirected graph representation**...



	A	B	C	D	E
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

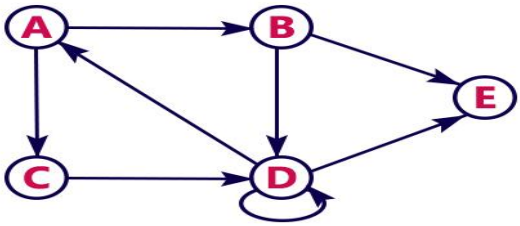


Undirected Graph

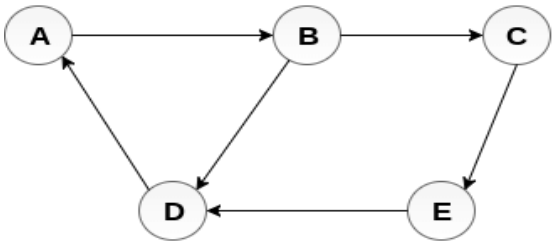
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

Example:2- consider the following **directed graph representation**...



	A	B	C	D	E
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

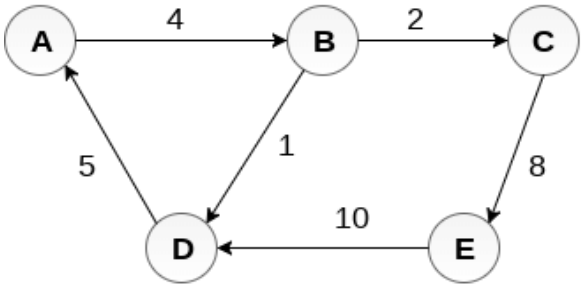


Directed Graph

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

Adjacency Matrix

Example:3- consider the following **weighted directed graph representation**...



Weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

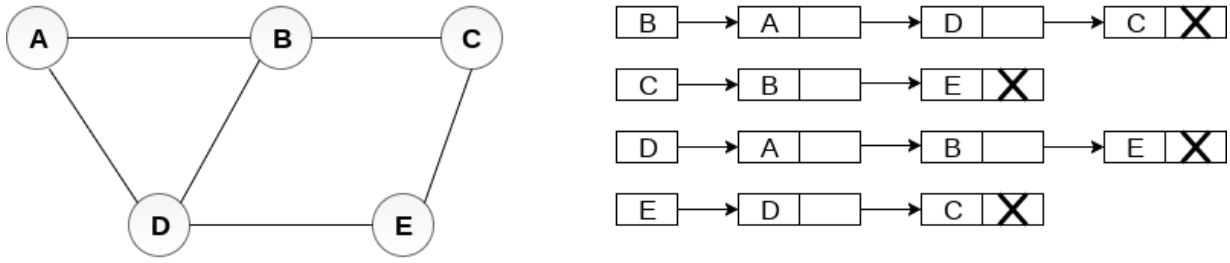
Adjacency Matrix

2. Adjacency List (Linked Representation):

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

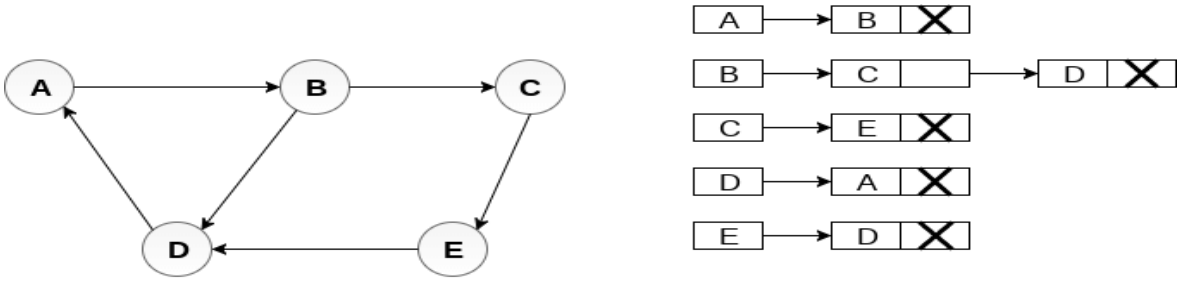
In this representation, every vertex of a graph contains list of its adjacent vertices.

Example:1- consider the following **undirected graph representation**...

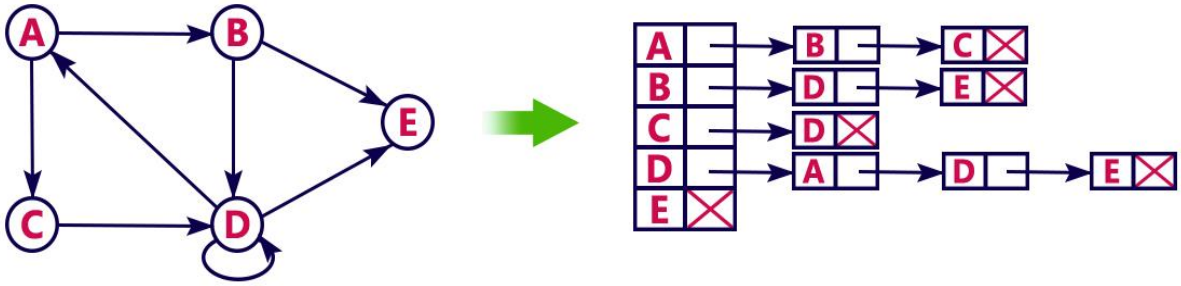


Undirected Graph **Adjacency List**

Example:2- consider the following **directed graph representation**...



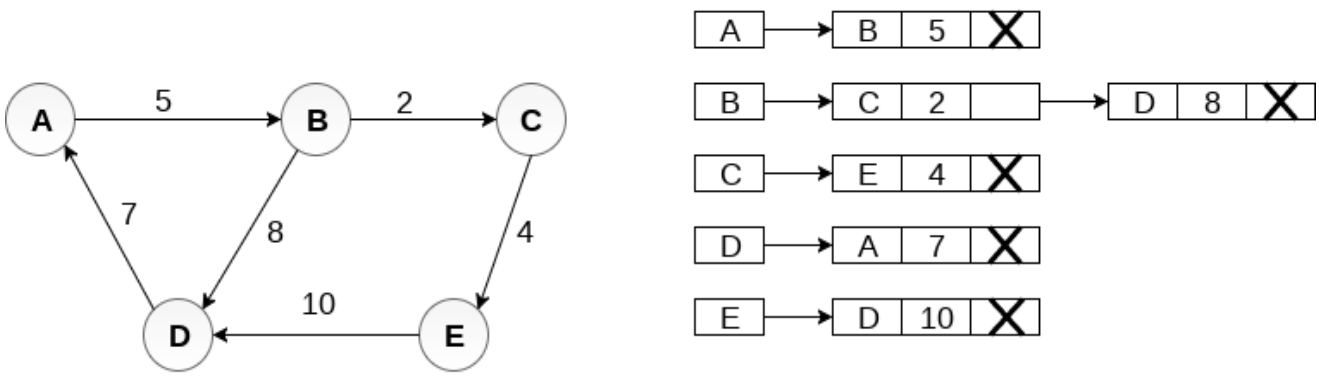
Directed Graph **Adjacency List**



In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.

Example:3- consider the following **weighted directed graph representation**

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.

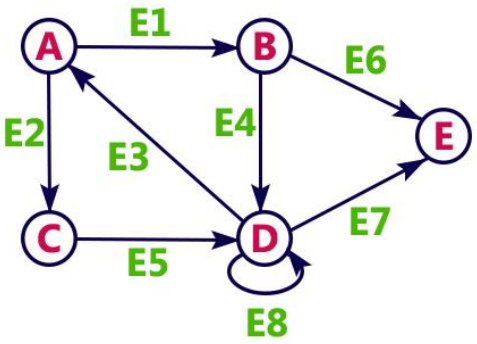


Weighted Directed Graph **Adjacency List**

3. Incidence Matrix:

- In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges.
- That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6.
- In this matrix, rows represent vertices and columns represent edges. This matrix is filled with 0 or 1 or -1.
- Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

Example: consider the following directed graph representation...



	E1	E2	E3	E4	E5	E6	E7	E8
A	1	1	-1	0	0	0	0	0
B	-1	0	0	1	0	1	0	0
C	0	-1	0	0	1	0	0	0
D	0	0	1	-1	-1	0	1	1
E	0	0	0	0	0	-1	-1	0

Graph Traversal Methods

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. That means using graph traversal we visit all the vertices of the graph without getting into looping path.
- The resultant graph for any traversal technique is Spanning tree

  There are two graph traversal techniques and they are as follows...

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

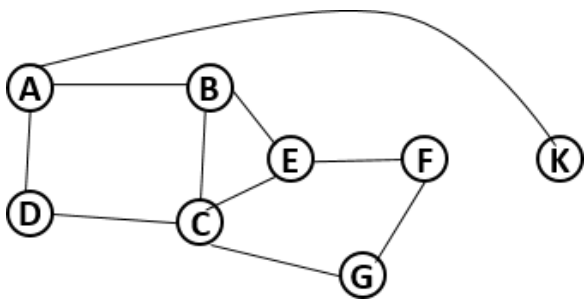
1. BFS (Breadth First Search) :

- 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

Example-1: Traverse the below graph using BFS

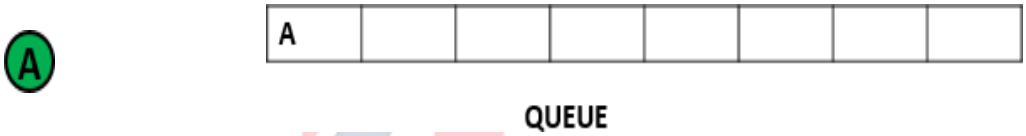


Solution: The adjacent list of every node is

Vertex	Adjacent nodes
A	B, D, K
B	A, C, E
C	B, D, E, G
D	A, C
E	B, C , F
F	E, G
G	C, F
K	A

Step 1:

- Create a Queue with size equal to number of vertices in a graph.
- Select '**A**' as starting vertex, mark as visited and insert into the Queue.



QUEUE

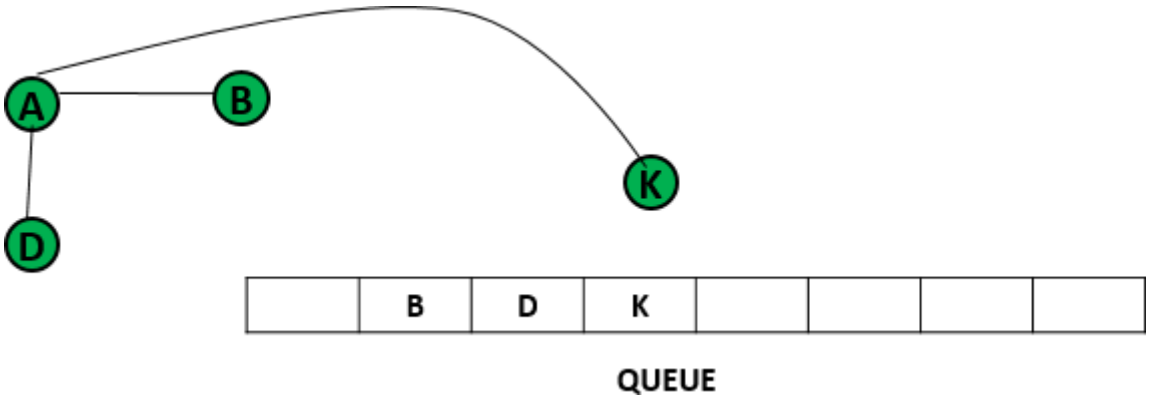
Step 2:

- Explore all unvisited adjacent node of '**A'**.
- Node **A as** three unvisited adjacent nodes i.e **B, D, K**.

  i.e mark '**B'** as visited and insert it onto the Queue than

  mark '**D'** as visited and insert it onto the Queue than

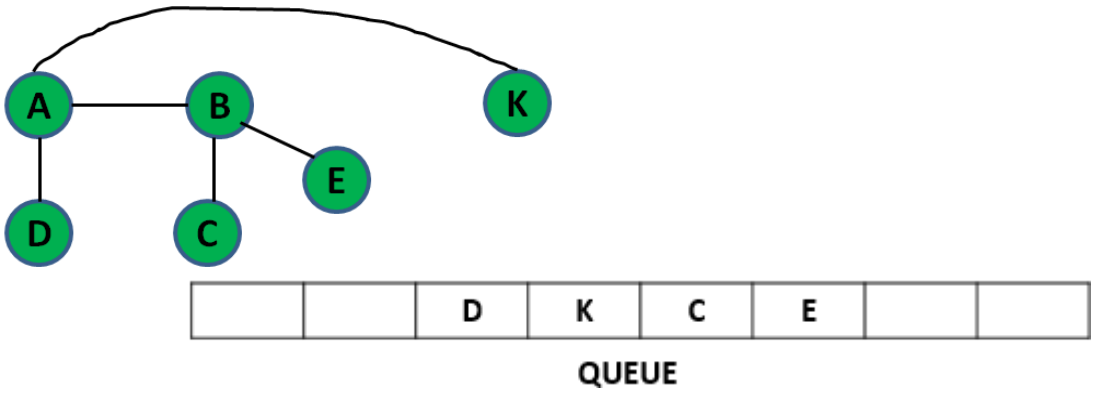
  mark '**K'** as visited and insert it onto the Queue.
- After visiting all adjacent nodes delete the vertex **A** from queue.



QUEUE

Step 3:

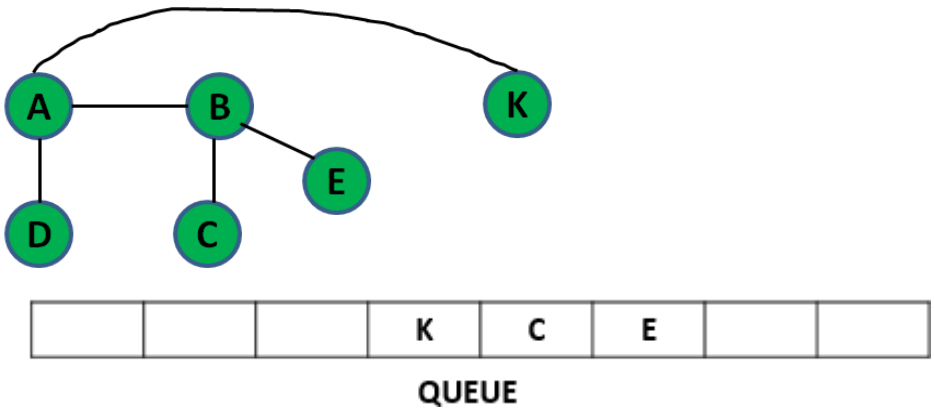
- Explore all unvisited adjacent node of '**B**'.
- Node '**B**' **as** three adjacent nodes i.e **A, C, E** but **A** is already visited. So don't visit it again.
- Visit all unvisited adjacent nodes and insert into Queue.
  i.e mark '**C**' as visited and insert it onto the Queue than mark '**E**' as visited and insert it onto the Queue.
- After visiting all adjacent nodes delete the vertex **B** from queue.



Visited order: A→B→D→K→C→E

Step 4:

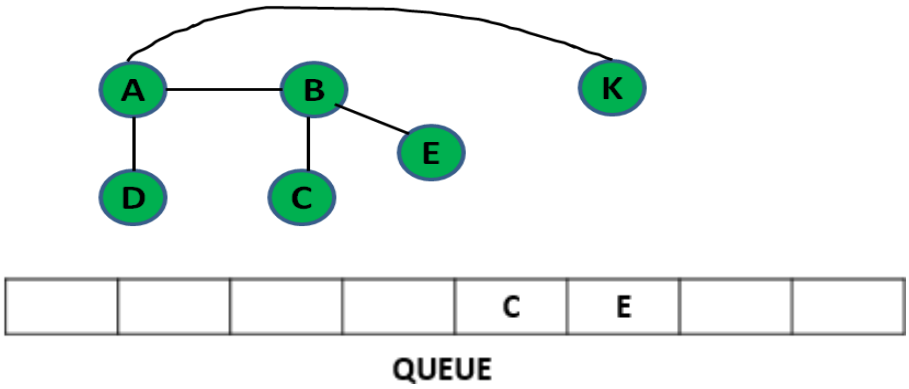
- Explore all unvisited adjacent node of '**D**'.
- Node '**D**' **as** two adjacent nodes i.e **A, C,** but **A and C** are already visited. So don't visit them again.
- As there are no unvisited adjacent nodes delete the vertex **D** from queue.



Visited order: A→B→D→K→C→E

Step 5:

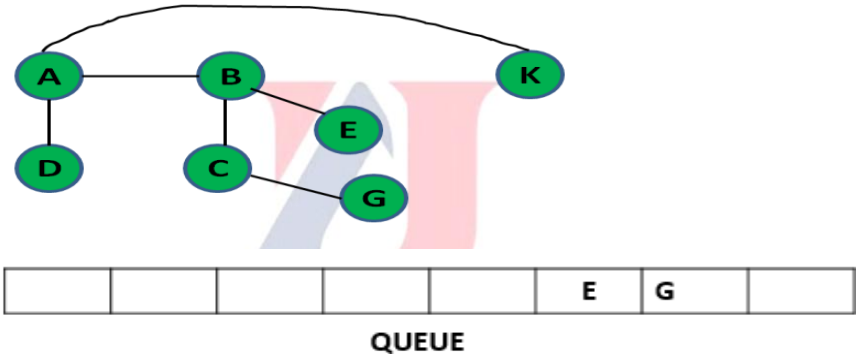
- Explore all unvisited adjacent node of '**K**'.
- Node '**K**' **as** one adjacent node i.e **A,** but already visited. So don't visit them again.
- As there are no unvisited adjacent nodes delete the vertex **K** from queue.



Visited order: A→B→D→K→C→E

Step 6:

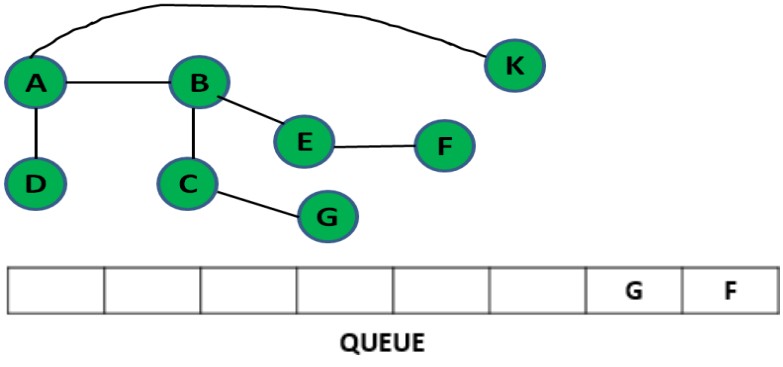
- Explore all unvisited adjacent node of '**C'**.
- Node '**C' as** four adjacent nodes i.e **B, D,E, G** but **B, D and E** are already visited. So don't visit them again.
- NOW mark '**G'** as visited and insert it onto the Queue than
- As there are no unvisited adjacent nodes delete the vertex **C** from queue.



Visited order: A→B→D→K→C→E→G

Step 7:

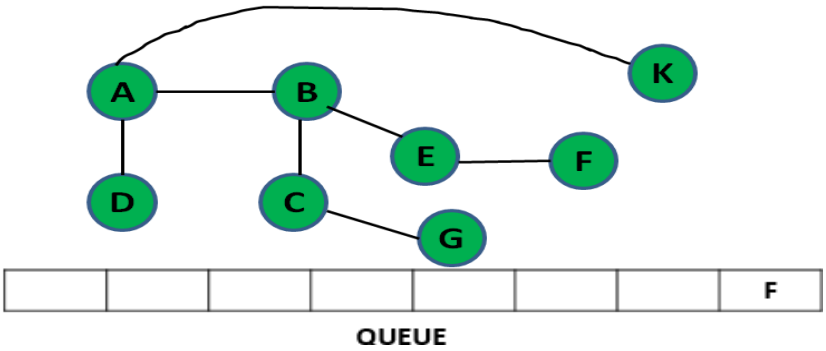
- Explore all unvisited adjacent node of '**E'**.
- Node '**E' as** four adjacent nodes i.e **B, C,F** but **B and C** are already visited. So don't visit them again.
- NOW Mark '**F'** as visited and insert it onto the Queue.
- As there are no unvisited adjacent nodes delete the vertex **E** from queue.



Visited order: A→B→D→K→C→E→G→F

Step 8:

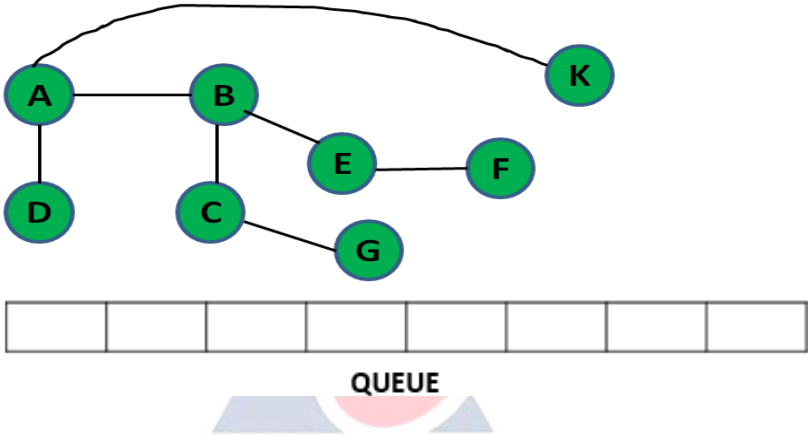
- Explore all unvisited adjacent node of '**G**'.
- As there are no unvisited adjacent nodes delete the vertex **G** from queue.



Visited order: A→B→D→K→C→E→G→F

Step 9:

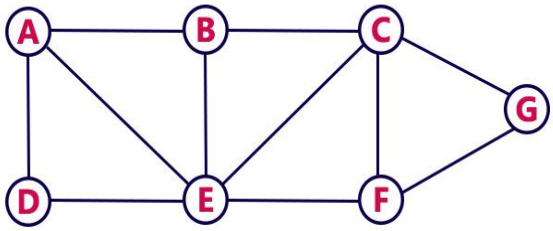
- Explore all unvisited adjacent node of '**F**'.
- As there are no unvisited adjacent nodes delete the vertex **F** from queue.



Visited order: A→B→D→K→C→E→G→F

Example-2:

Consider the following example graph to perform BFS traversal

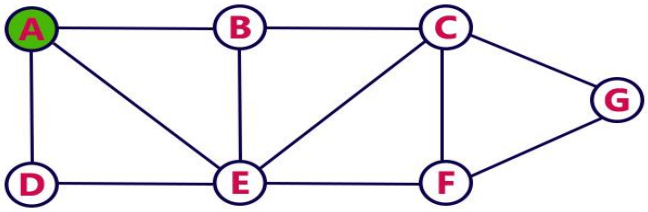


Adjacency List

A:	**D,E,B**
B:	**A,C,E**
C:	**B,E,F,G**
D:	**A,E**
E:	**A,B,D,C,F**
F:	**C,E,G**
G:	**C,F**

Step 1:

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

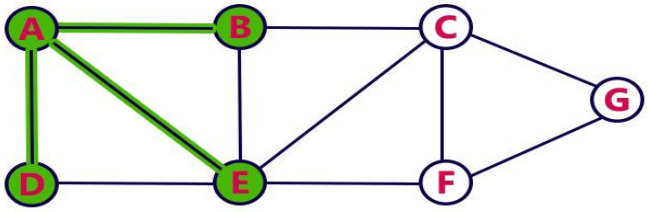


Queue

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..

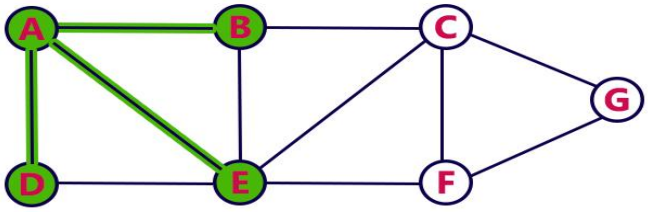


Queue

	D	E	B			

Step 3:

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

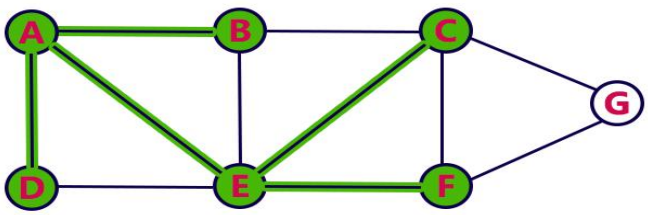


Queue

		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.

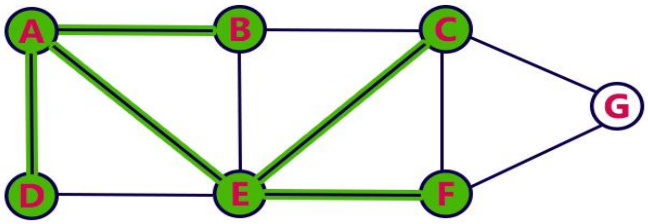


Queue

			B	C	F	

Step 5:

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

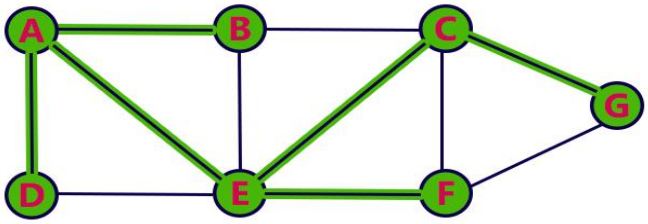


Queue

				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.

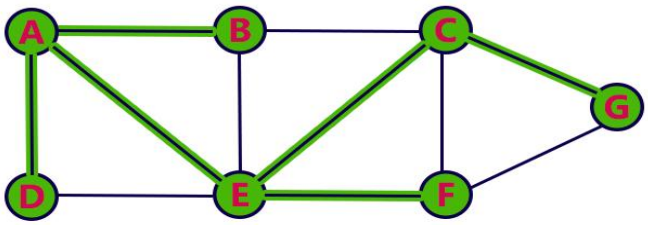


Queue

					F	G

Step 7:

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

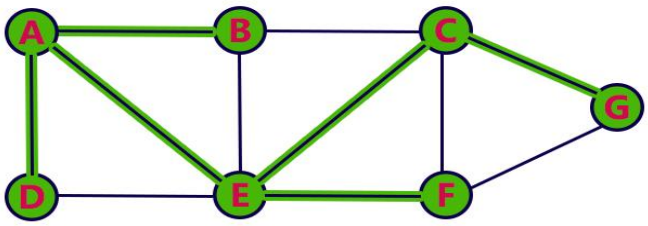


Queue

						G

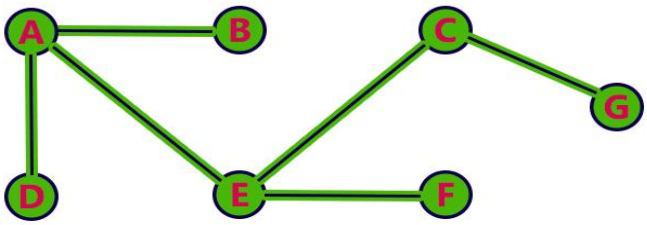
Step 8:

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



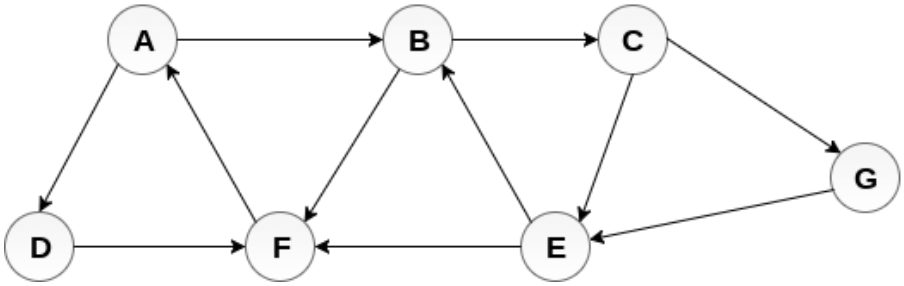
Queue

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



Example-3:

Consider the graph G shown in the following image; calculate the minimum path P from node A to node E. Given that each edge has a length of 1.



Adjacency Lists

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

Applications

- ✓ Finding the shortest path in an unweighted graph.
- ✓ Checking if a graph is connected.
- ✓ Web crawlers.
- ✓ Social networking friend suggestions.
- ✓ Solving puzzles like "minimum moves to reach a state".

1. Depth First Search (DFS) Algorithm:

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.

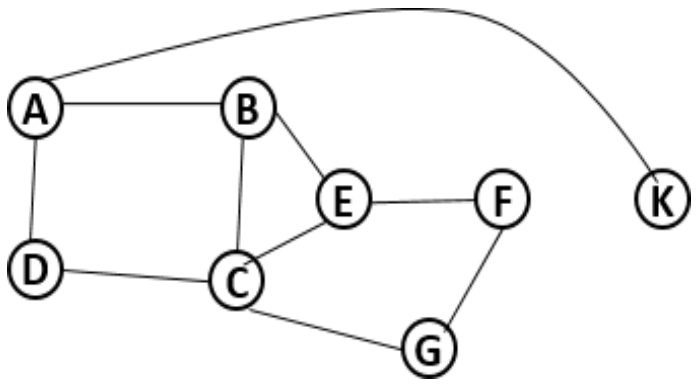
The data structure which is being used in DFS **is stack**. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

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.

Example-1: Traverse the below graph using DFS.



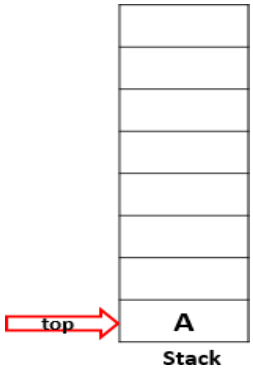
Solution: The adjacent list of every vertex is

Vertex	Adjacent nodes
A	B, D, K
B	A, C, E
C	B, D, E, G
D	A, C
E	B, C , F
F	E, G
G	C, F
K	A

Step 1:

- Create a stack with size equal to number of vertices in a graph.
- Select '**A**' as starting vertex, mark as visited and push onto stack.

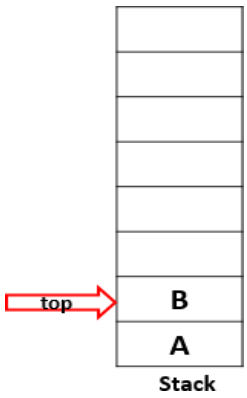
Visited order: A



Step 2:

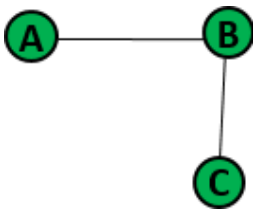
- Explore any unvisited adjacent node of '**A'**. We have three adjacent nodes i.e **B, D, K** and we can pick any one of them. Here we pick the node.
- So mark '**B'** as visited and push it onto the stack.

Visited order: A→ B

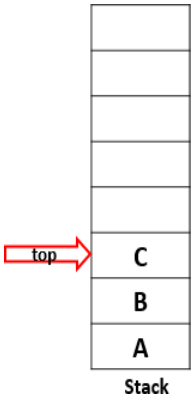


Step 3:

- Explore any unvisited adjacent node from '**B'**. We have three nodes i.e **A, C, E** but node '**A'** already visited so don't visit it again.
- So mark '**C'** as visited and push it onto the stack.

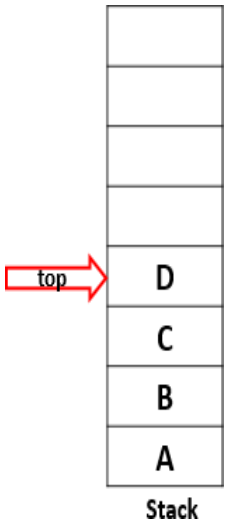


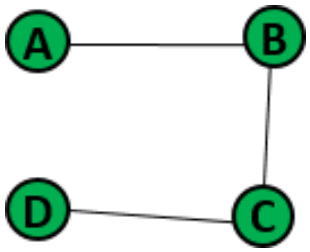
Visited order: A→B→C



Step 4:

- Explore any unvisited adjacent node from '**C**'. We have four nodes i.e **B, D, E, G** but node '**B**' already visited so don't visit it again.
- So mark '**D**' as visited and push it onto the stack.

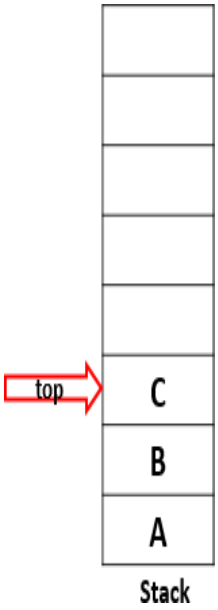


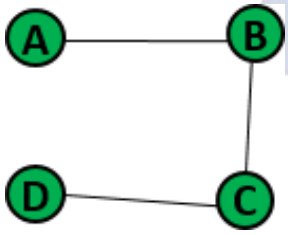


Visited order: A→B→C→D

Step 5:

- Explore any unvisited adjacent node from '**D**'. We have two nodes i.e **A, C** but adjacent nodes are already visited so don't visit them again.
- As '**D**' does not have any unvisited adjacent node. So, pop '**D**' from the stack.

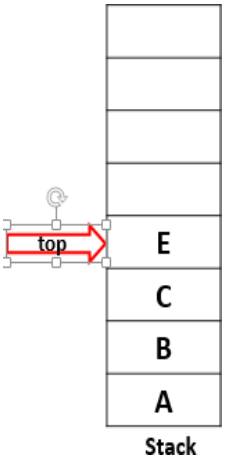


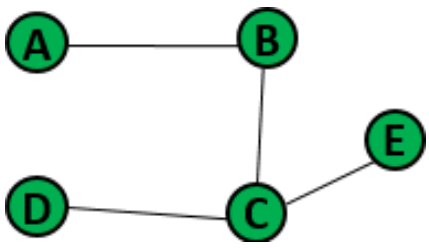


Visited order: A→B→C→D

Step 6:

- Explore any unvisited adjacent node from '**C**'. We have three nodes i.e **B, D, E** but node '**B**' **and 'D'** already visited so don't visit them again.
- So mark '**E**' as visited and push it onto the stack.

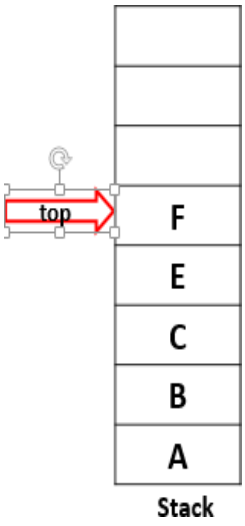


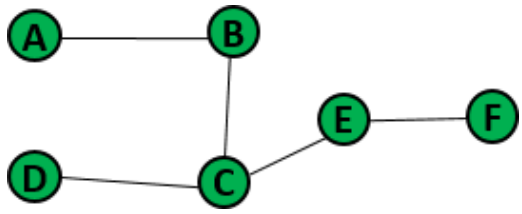


Visited order: A→B→C→D→E

Step 7:

- Explore any unvisited adjacent node from '**E'**. We have three nodes i.e **B, C, F** but node '**B' and 'C'** already visited so don't visit them again.
- So mark '**F'** as visited and push it onto the stack.

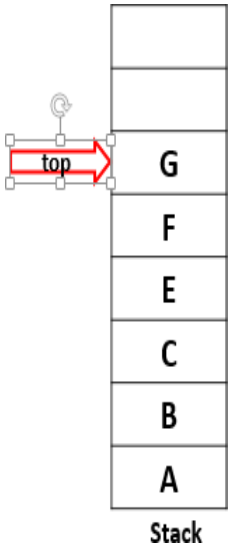


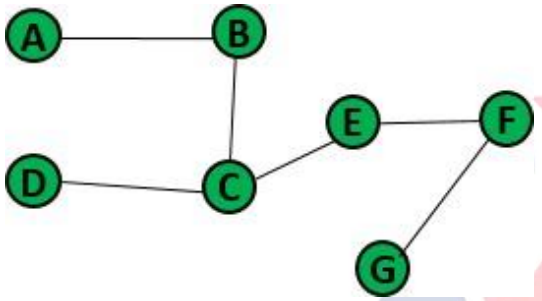


Visited order: A→B→C→D→E→F

Step 8:

- Explore any unvisited adjacent node from '**F'**. We have two nodes i.e **E, G** but node '**E'** already visited so don't visit it again.
- So mark '**G'** as visited and push it onto the stack.

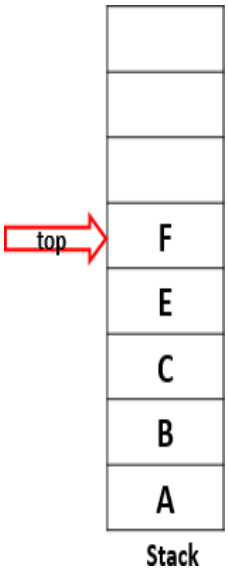


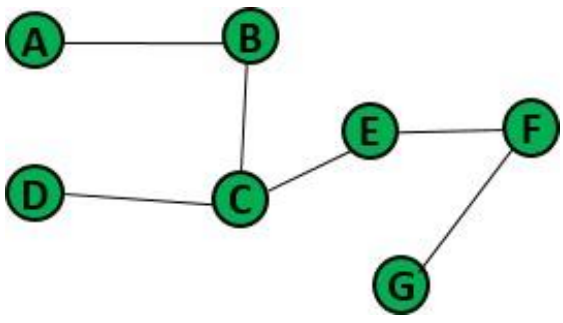


Visited order: A→B→C→D→E→F→G

Step 9:

- Explore any unvisited adjacent node from '**G'**. We have two nodes i.e **C, F** but adjacent nodes are already visited so don't visit them again.
- As '**G**' does not have any unvisited adjacent node. So, pop '**G'** from the stack

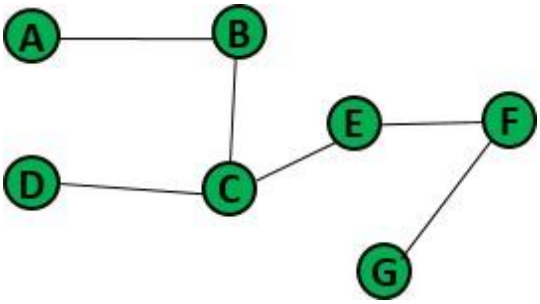


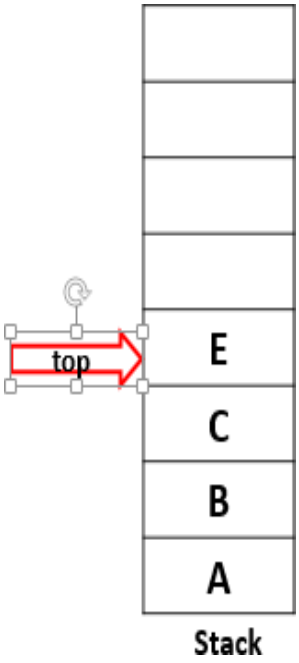


Visited order: A→B→C→D→E→F→G

Step 10:

- Explore any unvisited adjacent node from '**F**'. We have two nodes i.e **E, G** but adjacent nodes are already visited so don't visit them again.
- As '**F**' does not have any unvisited adjacent node. So, pop '**F**' from the stack

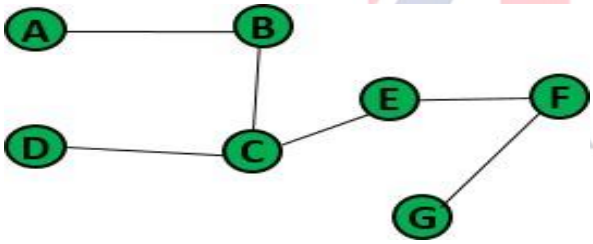


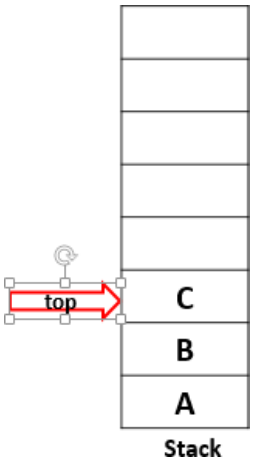


Visited order: A→B→C→D→E→F→G

Step 11:

- Explore any unvisited adjacent node from '**E**'. We have three nodes i.e **B, C, F** but adjacent nodes are already visited so don't visit them again.
- As '**E**' does not have any unvisited adjacent node. So, pop '**E**' from the stack

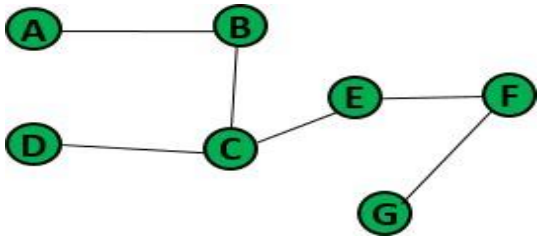


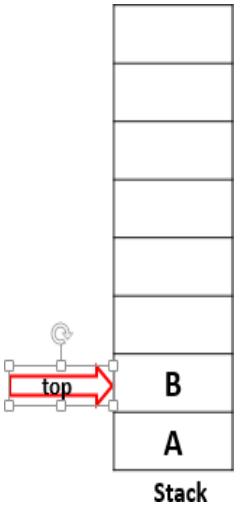


Visited order: A→B→C→D→E→F→G

Step 12:

- Explore any unvisited adjacent node from '**C**'. We have four nodes i.e **B, D, E, G** but adjacent nodes are already visited so don't visit them again.
- As '**C**' does not have any unvisited adjacent node. So, pop '**C**' from the stack

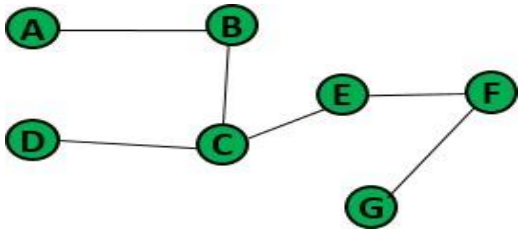




Visited order: A→B→C→D→E→F→G

Step 13:

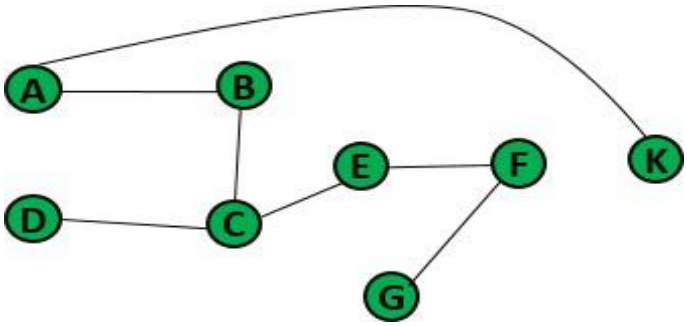
- Explore any unvisited adjacent node from '**B'**. We have three nodes i.e **A**,**C, E** but adjacent nodes are already visited so don't visit them again.
- As '**B'** does not have any unvisited adjacent node. So, pop '**B'** from the stack

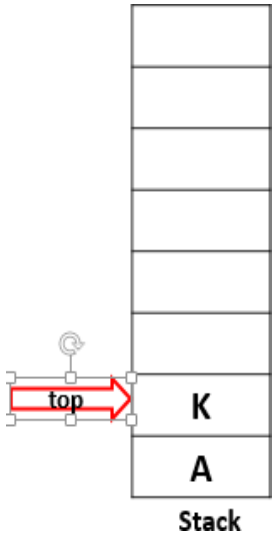


Visited order: A→B→C→D→E→F→G

Step 14:

- Explore any unvisited adjacent node from '**A'**. We have three nodes i.e **B**,**D, K** but adjacent nodes **B and D** already visited so don't visit them again.
- So mark '**K'** as visited and push it onto the stack.

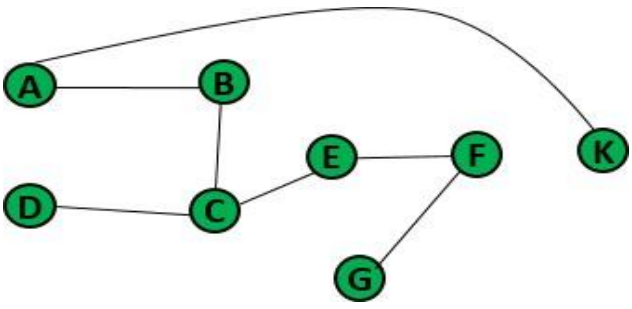




Visited order: A→B→C→D→E→F→G→K

Step 15:

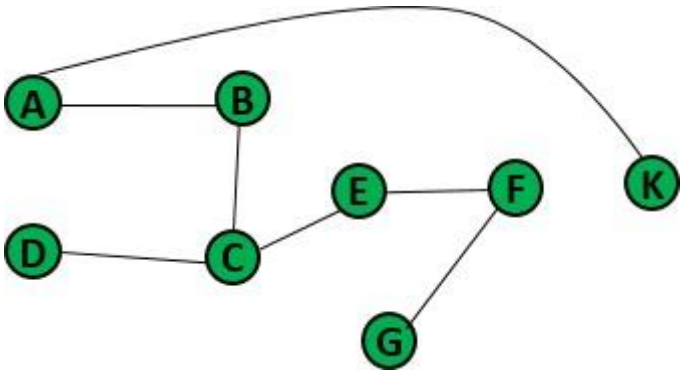
- Explore any unvisited adjacent node from '**K'**. We have only one node i.e **A**, but adjacent node is already visited so don't visit it again.
- As '**K'** does not have any unvisited adjacent node. So, pop '**K'** from the stack



Visited order: A→B→C→D→E→F→G→K

Step 16:

- Explore any unvisited adjacent node from '**A**'. We have three nodes i.e **B, D,K,** but adjacent nodes are already visited so don't visit them again.
- As '**A**' does not have any unvisited adjacent node. So, pop '**A**' from the stack

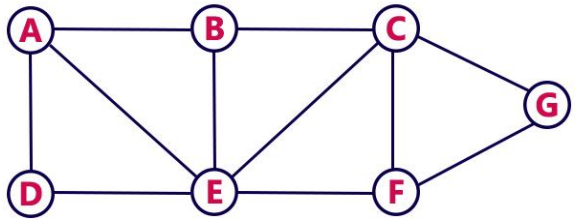


Visited order: A→B→C→D→E→F→G→

As stack is empty the traversing is done successfully

Example-2:

Consider the following example graph to perform DFS traversal

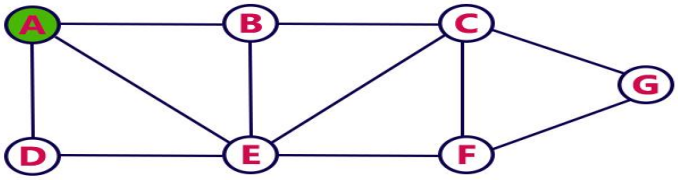


Adjacency List

A:	**D,E,B**
B:	**A,C,E**
C:	**B,E,F,G**
D:	**A,E**
E:	**A,B,D,C,F**
F:	**C,E,G**
G:	**C,F**

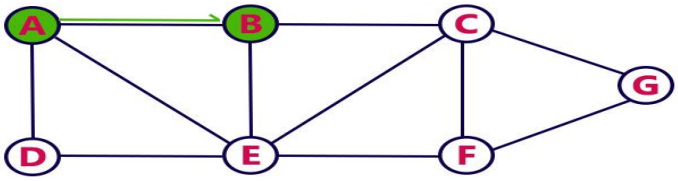
Step 1:

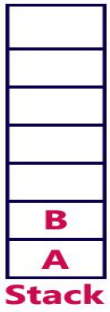
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



Step 2:

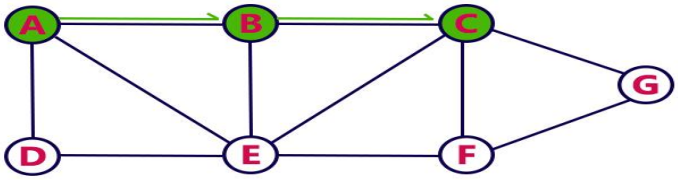
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

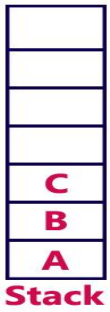




Step 3:

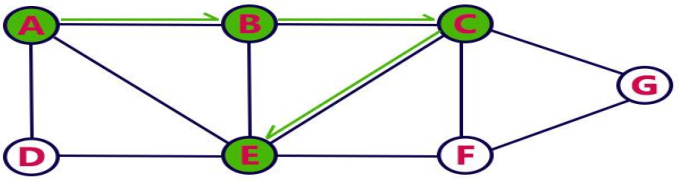
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

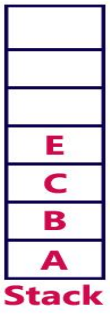




Step 4:

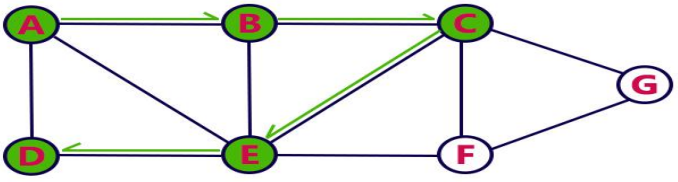
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack





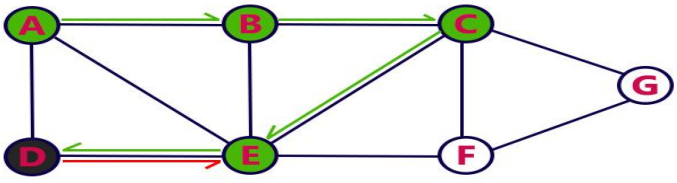
Step 5:

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



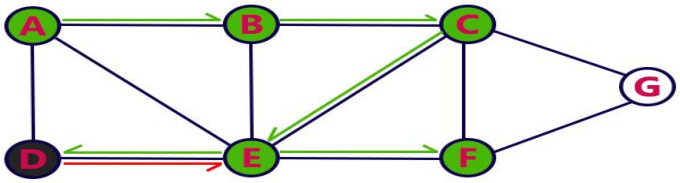
Step 6:

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



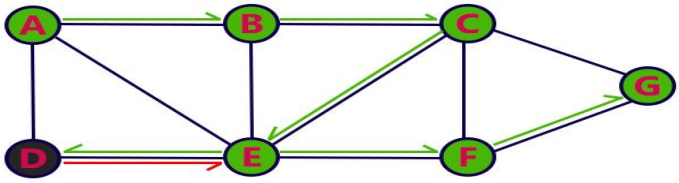
Step 7:

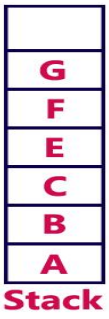
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



Step 8:

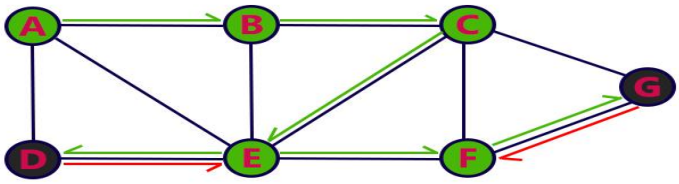
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

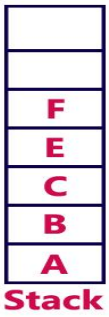




Step 9:

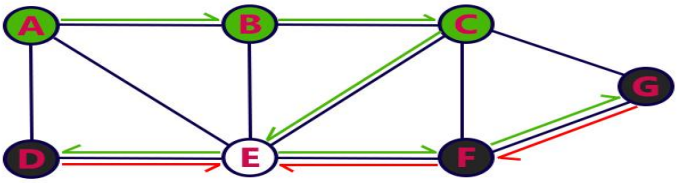
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

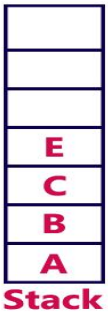




Step 10:

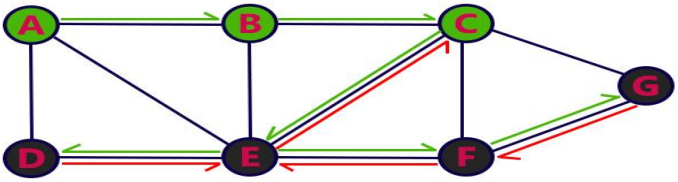
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

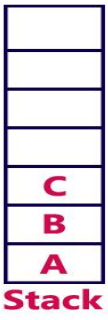




Step 11:

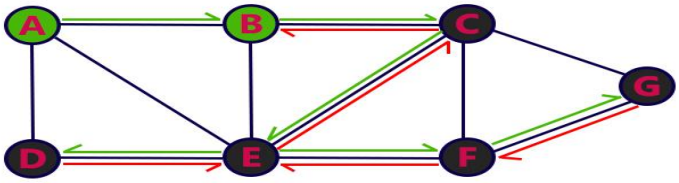
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.





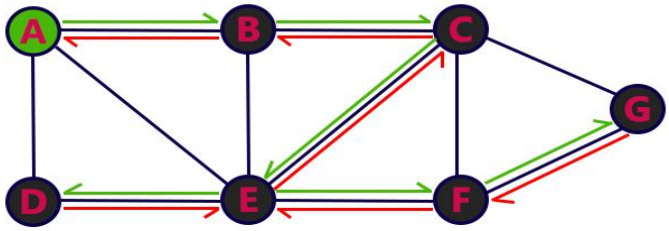
Step 12:

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



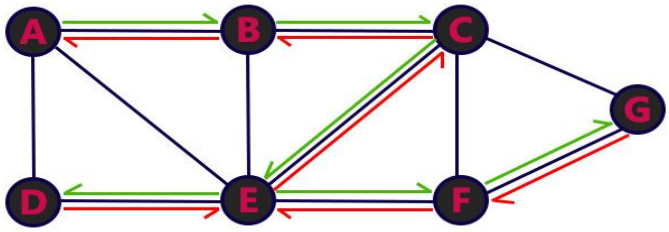
Step 13:

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

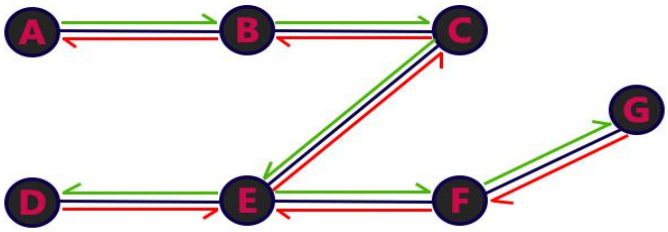


Step 14:

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

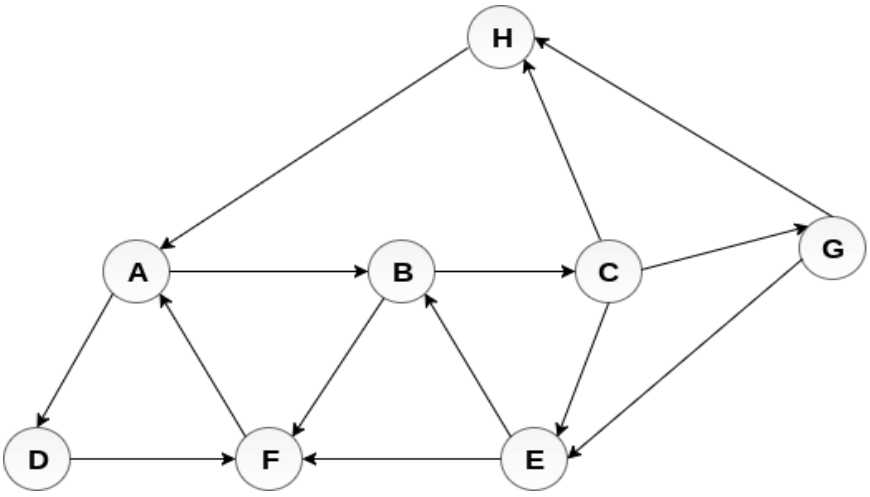


- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Example-3 :

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:

STEP 1:

- ✓ Select the vertex H as starting point (visit H)
- ✓ Push H on to the stack.

Print DFS

H							

H

STEP 2:

- ✓ Visit any adjacent vertex of H which is not visited (A)
- ✓ Push newly visited vertex A on to the stack

Print DFS

H	A						

A
H

STEP 3:

- ✓ Visit any adjacent vertex of A which is not visited (B)
- ✓ Push newly visited vertex B on to the stack

Print DFS

H	A	B					

B
A
H

STEP 4:

- ✓ Visit any adjacent vertex of B which is not visited (C)
- ✓ Push newly visited vertex C on to the stack

Print DFS

H	A	B	C				

C
B
A
H

STEP 5:

- ✓ Visit any adjacent vertex of C which is not visited (G)
- ✓ Push newly visited vertex G on to the stack

Print DFS

H	A	B	C	G			

G
C
B
A
H

STEP 6:

- ✓ Visit any adjacent vertex of G which is not visited (E)
- ✓ Push newly visited vertex E on to the stack

Print DFS	H	A	B	C	G	E		

E
G
C
B
A
H

STEP 7:

- ✓ Visit any adjacent vertex of E which is not visited (F)
- ✓ Push newly visited vertex F on to the stack

Print DFS	H	A	B	C	G	E	F	

F
E
G
C
B
A
H

STEP 8:

- ✓ There are no new vertices to be visited from F. so use back track
- ✓ Pop F from stack

E
G
C
B
A
H

STEP 9:

- ✓ There are no new vertices to be visited from E. so use back track
- ✓ Pop E from stack

G
C
B
A
H

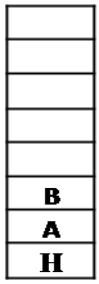
STEP 10:

- ✓ There are no new vertices to be visited from G. so use back track
- ✓ Pop G from stack

C
B
A
H

STEP 11:

- ✓ There are no new vertices to be visited from C. so use back track
- ✓ Pop C from stack

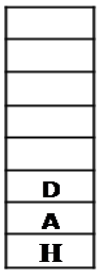


STEP 12:

- ✓ There are no new vertices to be visited from B. so use back track
- ✓ Pop B from stack

STEP 13:

- ✓ Visit any adjacent vertex of A which is not visited (D)
- ✓ Push newly visited vertex D on to the street



Print DFS

H	A	B	C	G	E	F	D

STEP 14:

- ✓ There are no new vertices to be visited from D. so use back track
- ✓ Pop D from stack

STEP 15:

- ✓ There are no new vertices to be visited from A. so use back track
- ✓ Pop A from stack

STEP 16:

- ✓ There are no new vertices to be visited from H. so use back track
- ✓ Pop H from stack

Stack becomes empty. So stop DFS Traversal

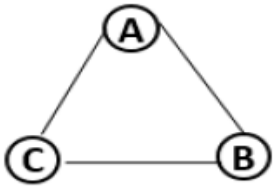
Print DFS

H	A	B	C	G	E	F	D

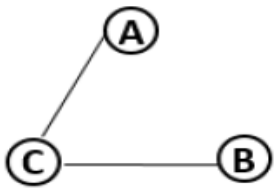
Spanning Trees

- 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..
- By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.
- A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.
- A complete undirected graph can have $\mathbf{n^{n-2}}$ number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be $\mathbf{5^{5-2} = 125.}$

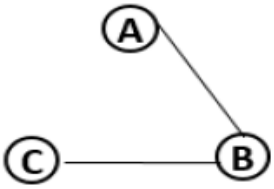
Example:1



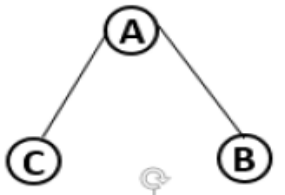
The maximum number of spanning trees for above graph is $3^{3-2} = 3^1 = 3$
The spanning trees are shown below



Spanning tree 1



Spanning tree 2



Spanning tree 3

Applications of the Spanning Tree:

A spanning tree is preferred to discover the shortest path to link all nodes of the graph. In this section, we will discuss some of the most famous and common applications of the spanning tree:

- A spanning tree is very helpful in the civil network zone and planning.
- It is used in network routing protocol.

Properties of spanning-tree:

Some of the properties of the spanning tree are given as follows -

- 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.

So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

Minimum spanning tree (MST) :

A minimum spanning tree (MST) is a subset of the edges of a connected, undirected graph that connects all the vertices with the most negligible possible total weight of the edges. A minimum spanning tree has precisely n-1 edges, where n is the number of vertices in the graph.

OR

Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

1. **Kruskal's Algorithm**:

In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution

Algorithm:

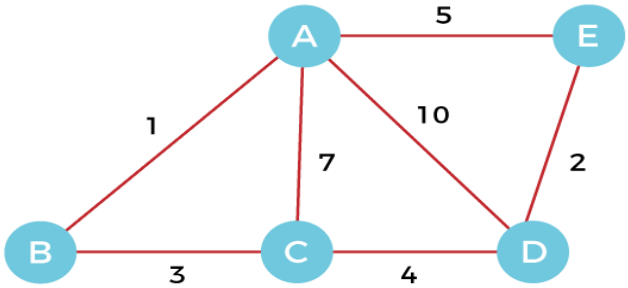
- Step 1: Sort all edges in increasing order of their edge weights.
- Step 2:
  - Take the edge with the lowest weight and use it to connect the vertices of graph.
  - If adding an edge creates a cycle, then reject that edge and go for the next least weight edge
- Step 3: Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

Thumb Rule to Remember

The above steps may be reduced to the following thumb rule-

- ✓ Simply draw all the vertices on the paper.
- ✓ Connect these vertices using edges with minimum weights such that no cycle gets formed.

Example-1: Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm.



The weight of the edges of the above graph is given in the below table -

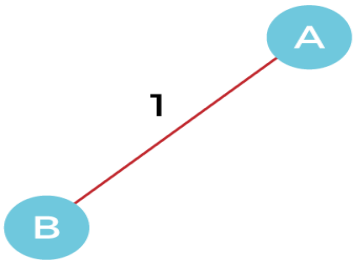
Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

Now, sort the edges given above in the ascending order of their weights.

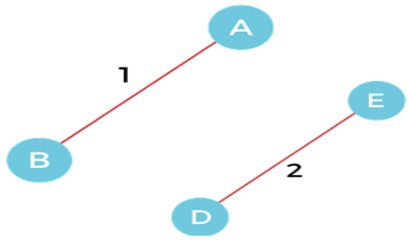
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Now, let's start constructing the minimum spanning tree.

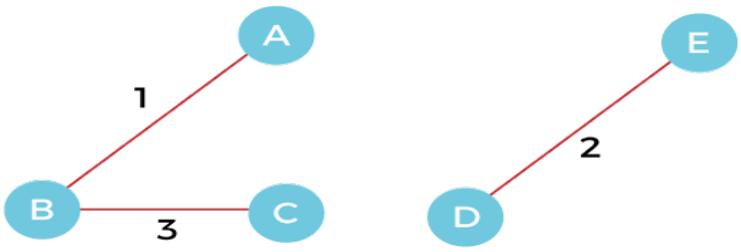
Step 1 - First, add the edge **AB** with weight **1** to the MST.



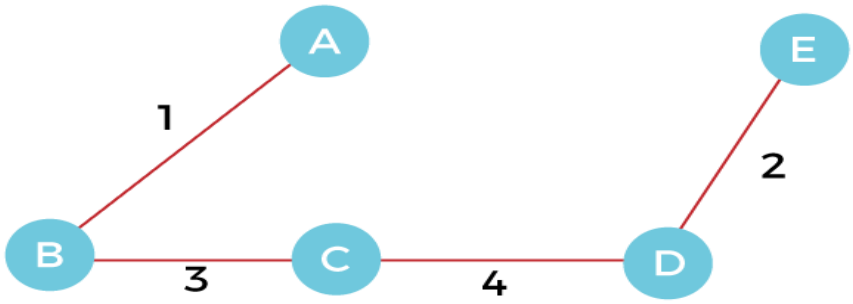
Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



Step 3 - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.



Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.

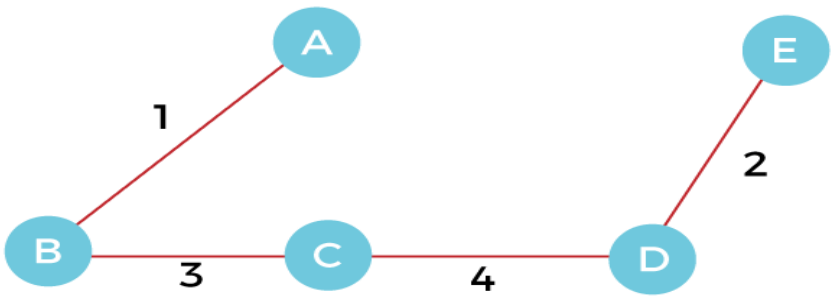


Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

Step 6 - Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

Step 7 - Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

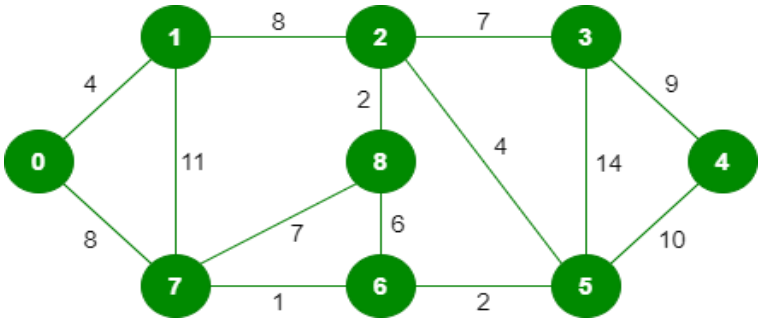
So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



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

Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

Example-2: Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm.



- The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.
- After sorting:

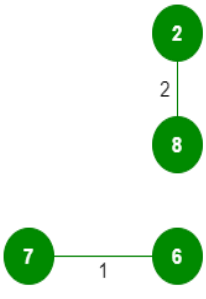
Weight	Source	Destination
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from the sorted list of edges

Step 1: Pick edge 7-6. No cycle is formed, include it.

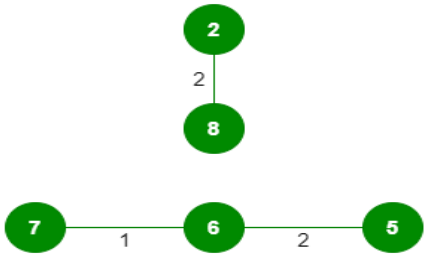
Add edge 7-6 in the MST

Step 2: Pick edge 8-2. No cycle is formed, include it.



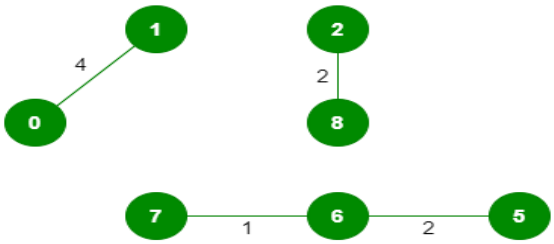
Add edge 8-2 in the MST

Step 3: Pick edge 6-5. No cycle is formed, include it.



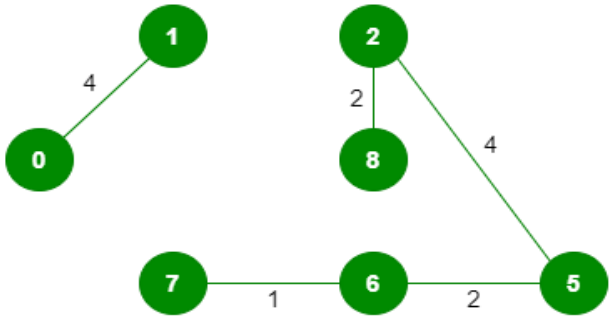
Add edge 6-5 in the MST

Step 4: Pick edge 0-1. No cycle is formed, include it.



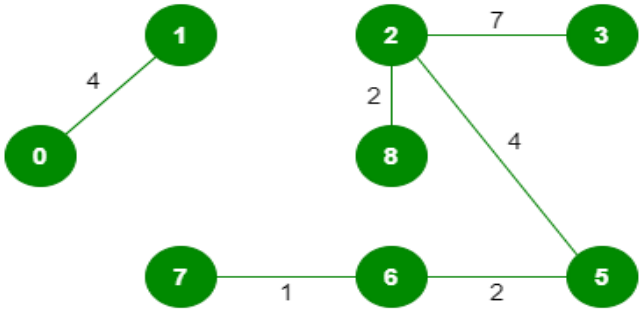
Add edge 0-1 in the MST

Step 5: Pick edge 2-5. No cycle is formed, include it.



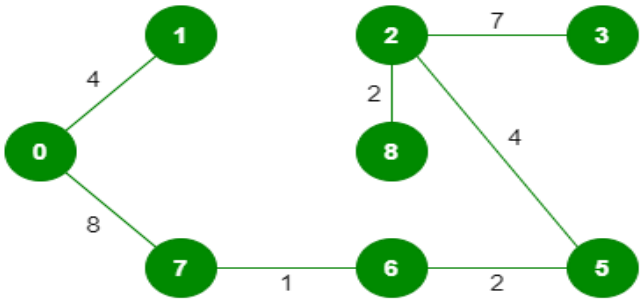
Add edge 2-5 in the MST

Step 6: Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it.



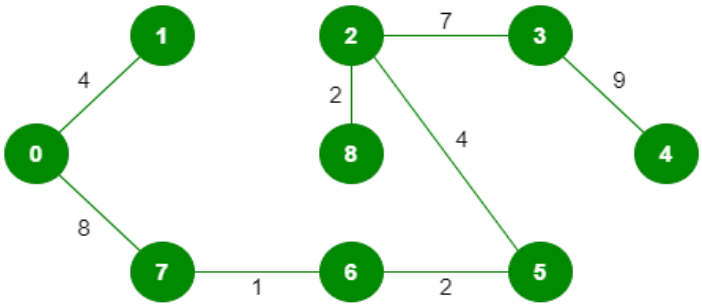
Add edge 2-3 in the MST

Step 7: Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is formed, include it.



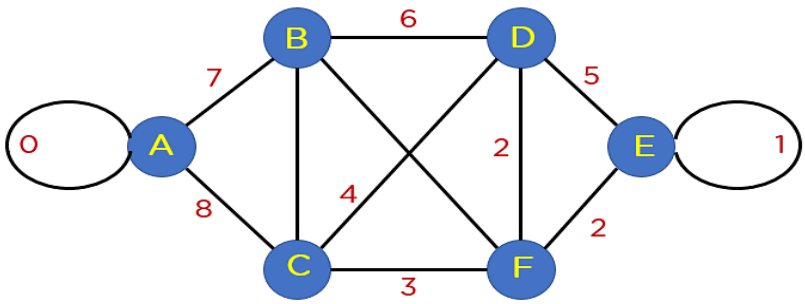
Add edge 0-7 in MST

Step 8: Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle is formed, include it.

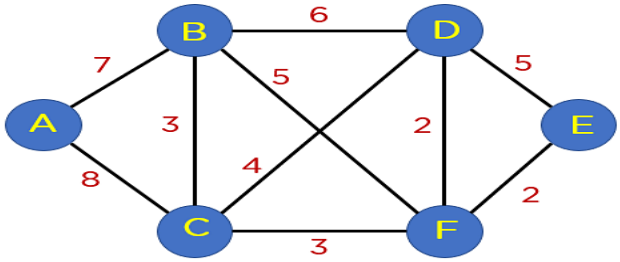


Note: Since the number of edges included in the MST equals to (V – 1), so the algorithm stops here

Example-3: Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm



- If you observe this graph, you'll find two looping edges connecting the same node to itself again. And you know that the tree structure can never include a loop or parallel edge.
- Hence, primarily you will need to remove these edges from the graph structure

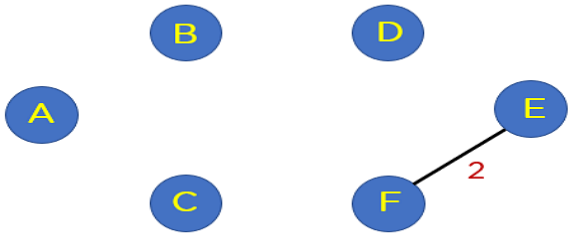


Removing parallel edges or loops from the graph.

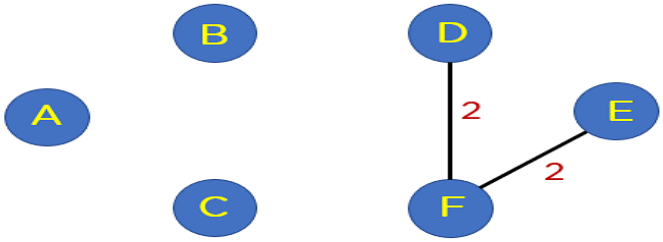
The next step that you will proceed with is arranging all edges in a sorted list by their edge weights.

The Edges of the Graph	Edge Weight	
Source Vertex	Destination Vertex	
E	F	2
F	D	2
B	C	3
C	F	3
C	D	4
B	F	5
B	D	6
A	B	7
A	C	8

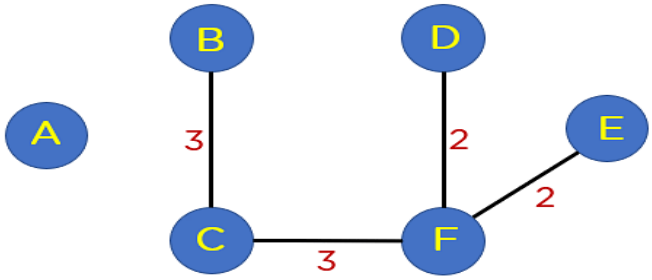
After this step, you will include edges in the MST such that the included edge would not form a cycle in your tree structure. The first edge that you will pick is edge EF, as it has a minimum edge weight that is 2.



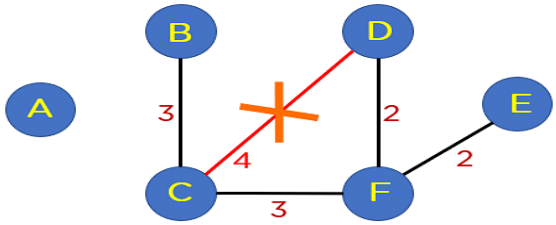
Add edge FD to the spanning tree



Add edge BC and edge CF to the spanning tree as it does not generate any loop.

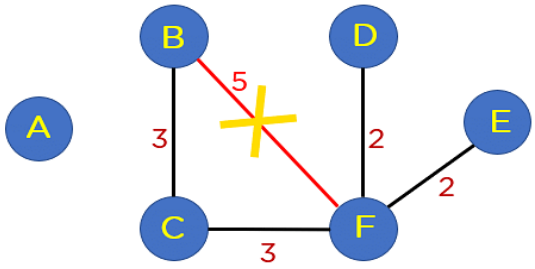


Next up is edge CD. This edge generates the loop in Your tree structure. Thus, you will discard this edge.



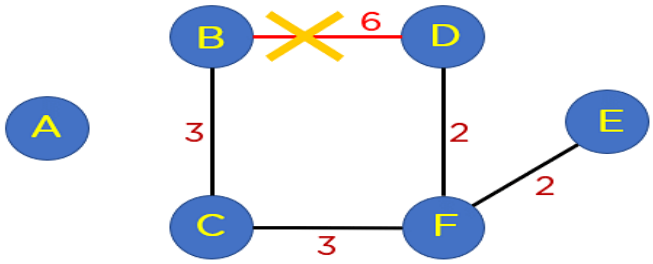
Edge CD should be discarded, as it creates loop.

Following edge CD, you have edge BF. This edge also creates the loop; hence you will discard it.



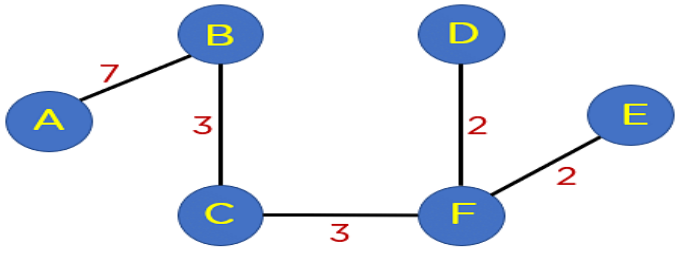
Edge BF should be discarded.

Next up is edge BD. This edge also formulates a loop, so you will discard it as well.



Edge BD should be discarded.

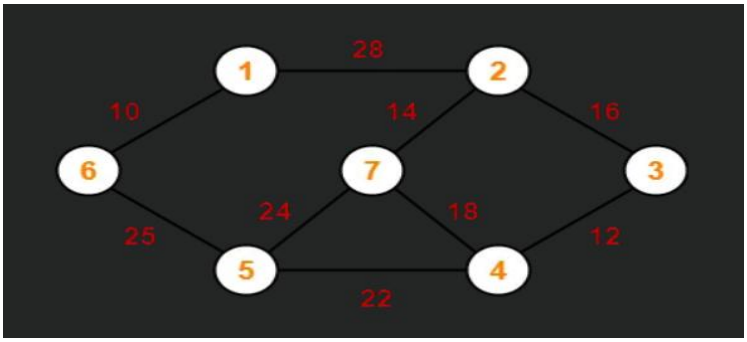
Next on your sorted list is edge AB. This edge does not generate any cycle, so you need not include it in the MST structure. By including this node, it will include 5 edges in the MST, so you don't have to traverse any further in the sorted list. The final structure of your MST is represented in the image below:



Minimum Spanning Tree.

The summation of all the edge weights in MST T(V', E') is equal to 17, which is the least possible edge weight for any possible spanning tree structure for this particular graph..

Example-4: Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm.

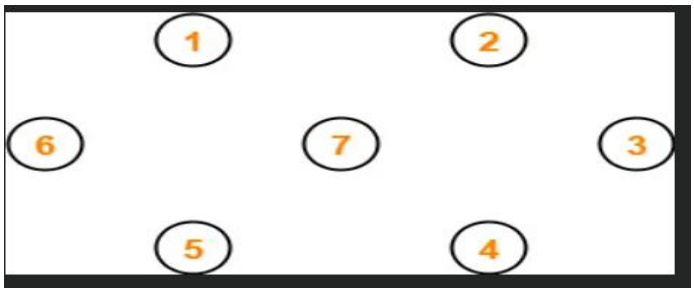


Solution-

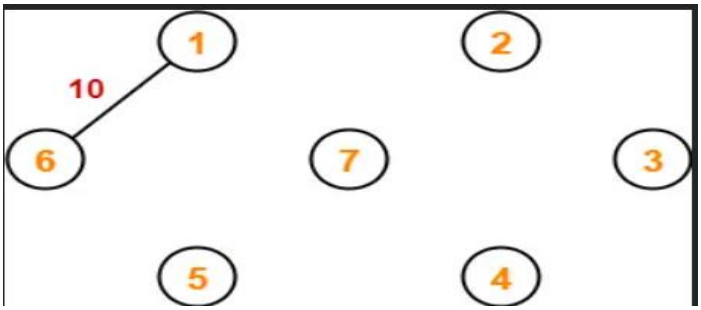
To construct MST using Kruskal's Algorithm,

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

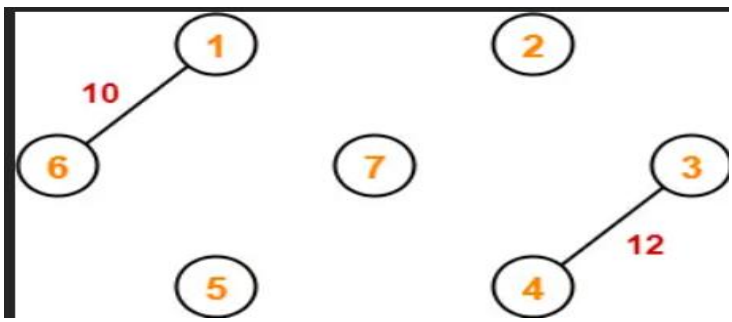
Step-01:



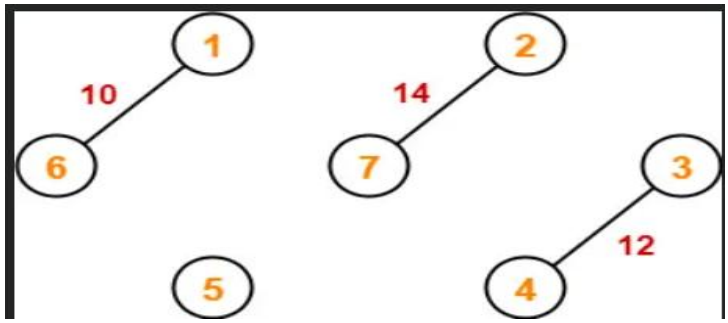
Step-02:



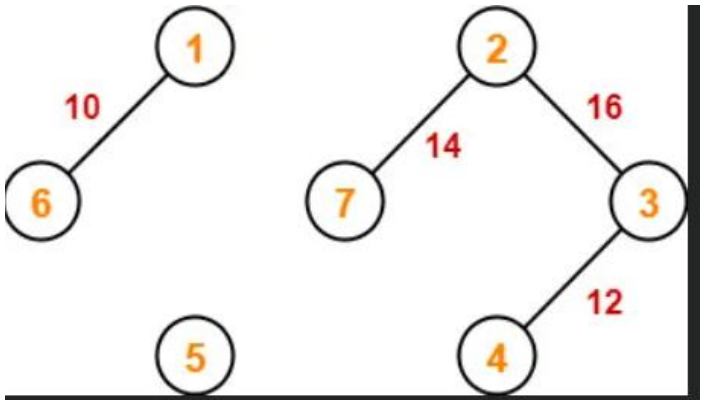
Step-03:



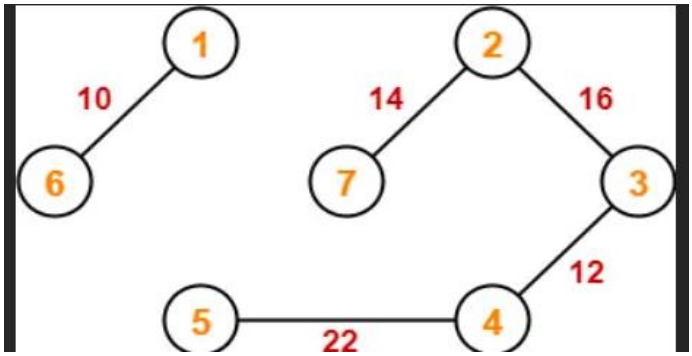
Step-04:



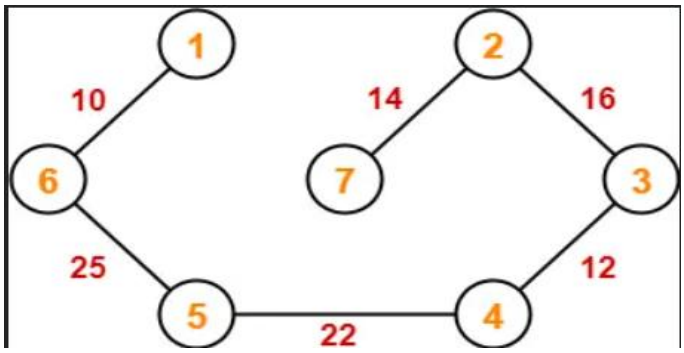
Step-05:



Step-06:



Step-07:



Since all the vertices have been connected / included in the MST, so we stop.

Weight of the MST

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

2. 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.
- Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.
- Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

Algorithm:

Step-01:

- ✓ Randomly choose any vertex.
- ✓ The vertex connecting to the edge having least weight is usually selected.

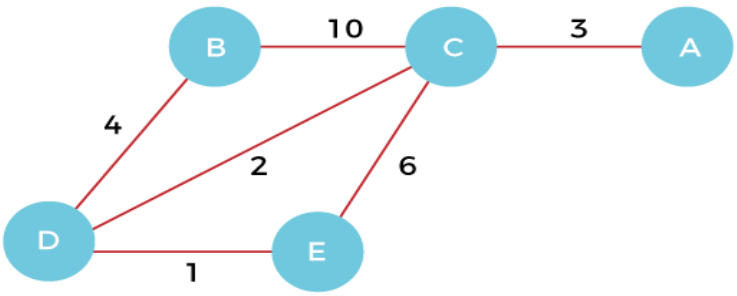
Step-02:

- ✓ Find all the edges that connect the tree to new vertices.
- ✓ Find the least weight edge among those edges and include it in the existing tree.
- ✓ If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

Step-03:

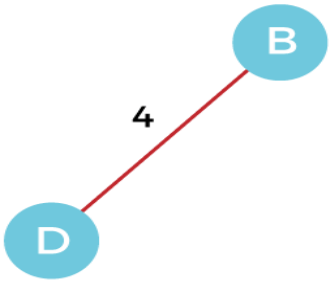
- ✓ Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.

Example:01- Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-

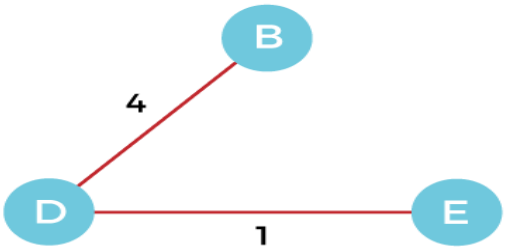


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

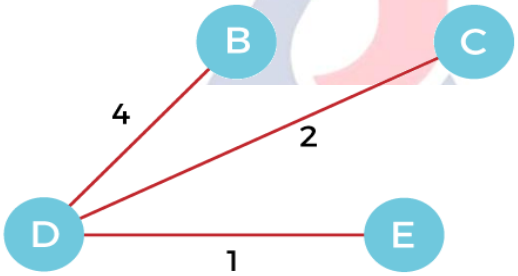
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



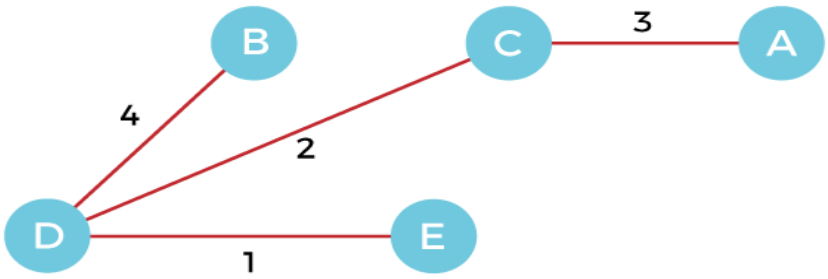
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



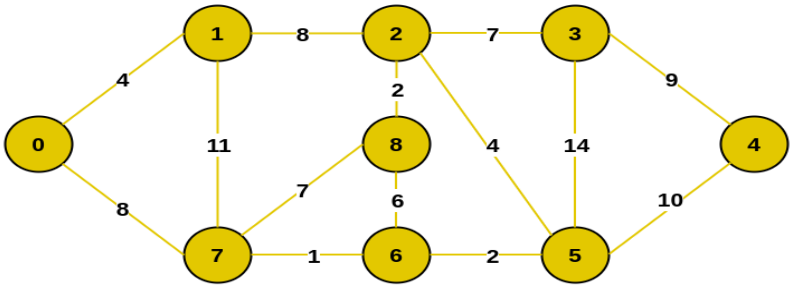
Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



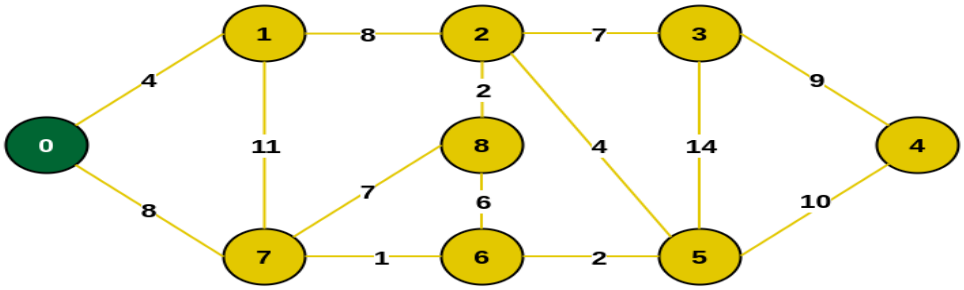
So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

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

Example:02- Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-



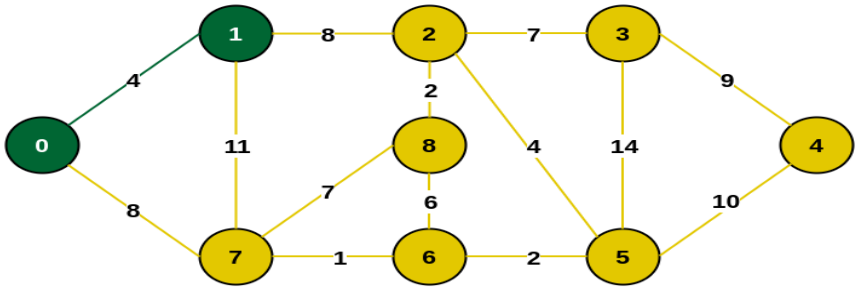
Step 1: Firstly, we select an arbitrary vertex that acts as the starting vertex of the Minimum Spanning Tree. Here we have selected vertex **0** as the starting vertex.



Select an arbitrary starting vertex. Here we have selected 0

0 is selected as starting vertex

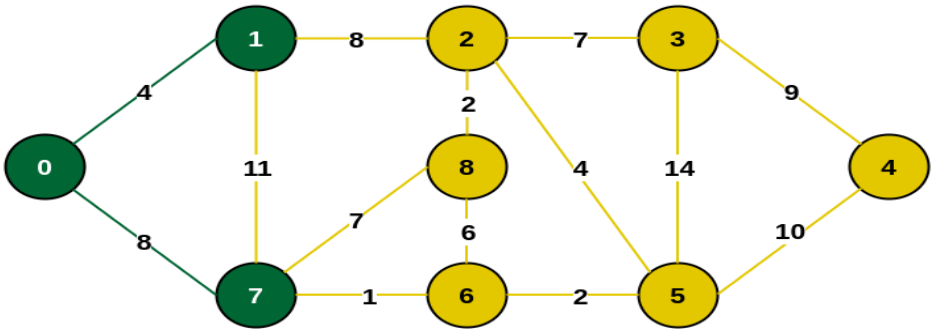
Step 2: All the edges connecting the incomplete MST and other vertices are the edges {0, 1} and {0, 7}. Between these two the edge with minimum weight is {0, 1}. So include the edge and vertex 1 in the MST.



Minimum weighted edge from MST to other vertices is 0-1 with weight 4

1 is added to the MST

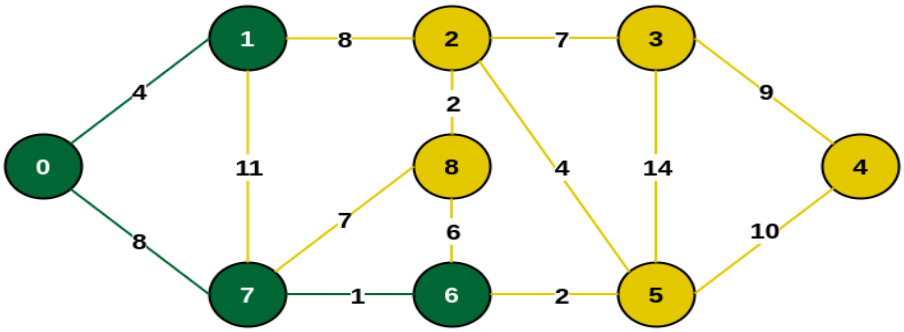
Step 3: The edges connecting the incomplete MST to other vertices are {0, 7}, {1, 7} and {1, 2}. Among these edges the minimum weight is 8 which is of the edges {0, 7} and {1, 2}. Let us here include the edge {0, 7} and the vertex 7 in the MST. [We could have also included edge {1, 2} and vertex 2 in the MST].



Minimum weighted edge from MST to other vertices is 0-7 with weight 8

7 is added in the MST

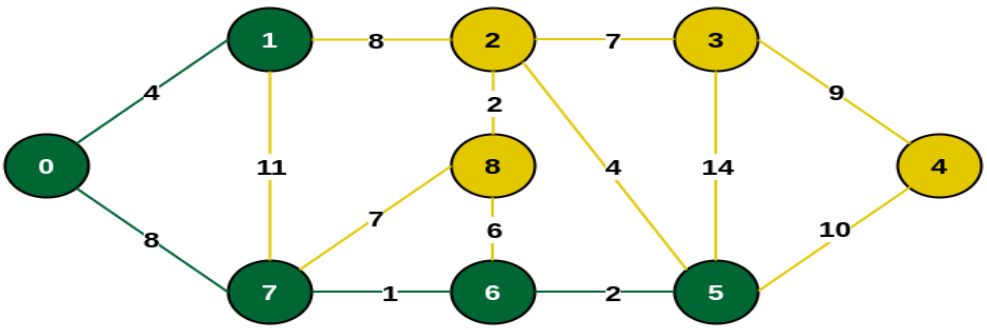
Step 4: The edges that connect the incomplete MST with the fringe vertices are {1, 2}, {7, 6} and {7, 8}. Add the edge {7, 6} and the vertex 6 in the MST as it has the least weight (i.e., 1).



Minimum weighted edge from MST to other vertices is 7-6 with weight 1

6 is added in the MST

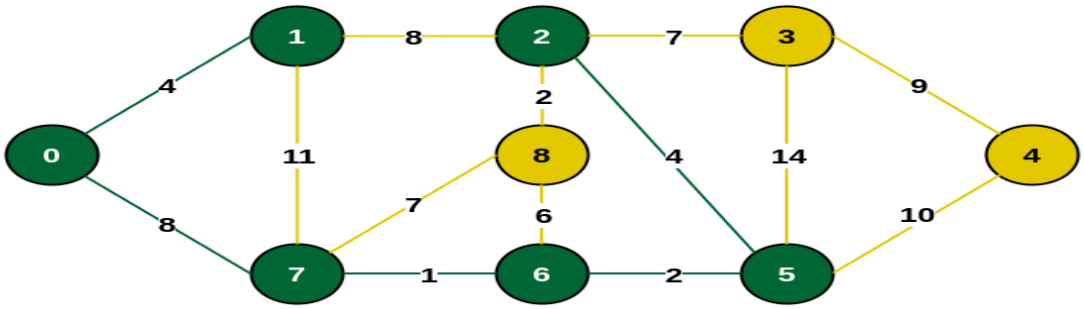
Step 5: The connecting edges now are {7, 8}, {1, 2}, {6, 8} and {6, 5}. Include edge {6, 5} and vertex 5 in the MST as the edge has the minimum weight (i.e., 2) among them.



Minimum weighted edge from MST to other vertices is 6-5 with weight 2

Include vertex 5 in the MST

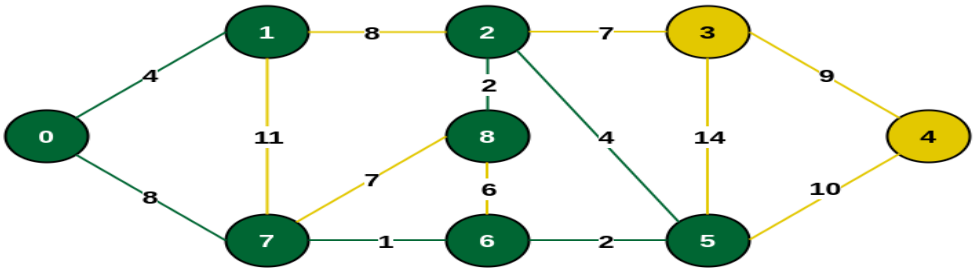
Step 6: Among the current connecting edges, the edge {5, 2} has the minimum weight. So include that edge and the vertex 2 in the MST.



Minimum weighted edge from MST to other vertices is 5-2 with weight 4

Include vertex 2 in the MST

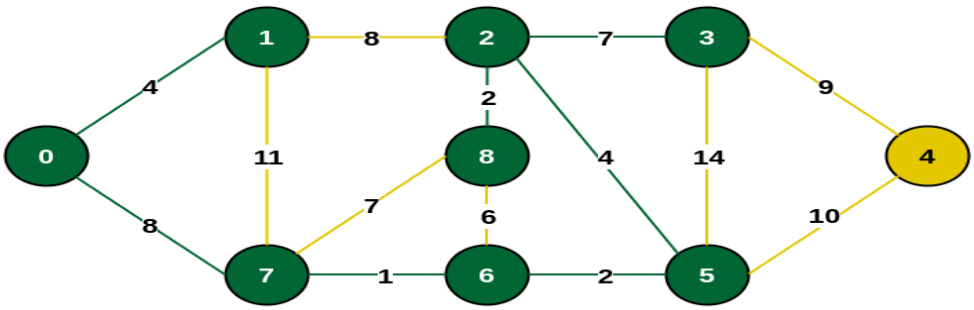
Step 7: The connecting edges between the incomplete MST and the other edges are {2, 8}, {2, 3}, {5, 3} and {5, 4}. The edge with minimum weight is edge {2, 8} which has weight 2. So include this edge and the vertex 8 in the MST.



Minimum weighted edge from MST to other vertices is 2-8 with weight 2

Add vertex 8 in the MST

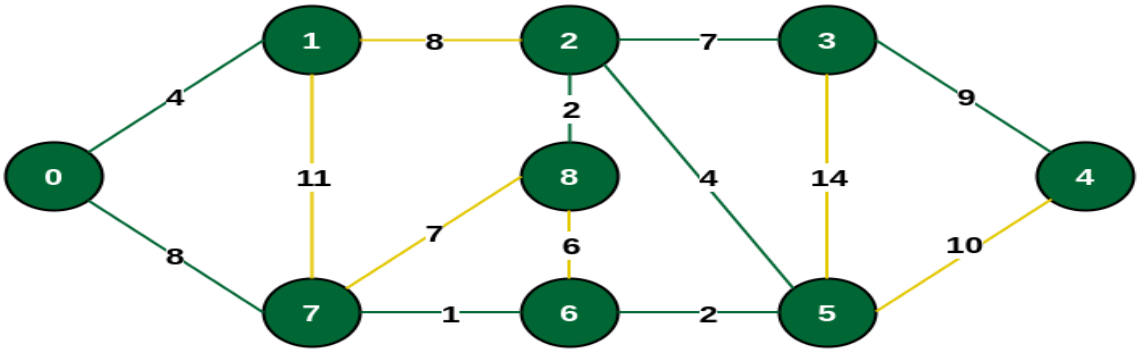
Step 8: See here that the edges {7, 8} and {2, 3} both have same weight which are minimum. But 7 is already part of MST. So we will consider the edge {2, 3} and include that edge and vertex 3 in the MST.



Minimum weighted edge from MST to other vertices is 2-3 with weight 7

Include vertex 3 in MST

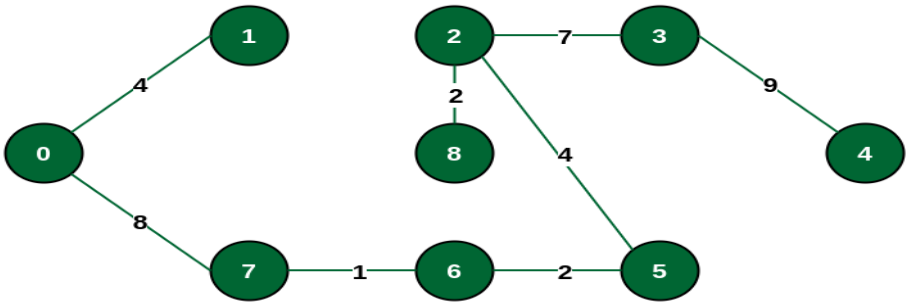
Step 9: Only the vertex 4 remains to be included. The minimum weighted edge from the incomplete MST to 4 is {3, 4}.



Minimum weighted edge from MST to other vertices is 3-4 with weight 9

Include vertex 4 in the MST

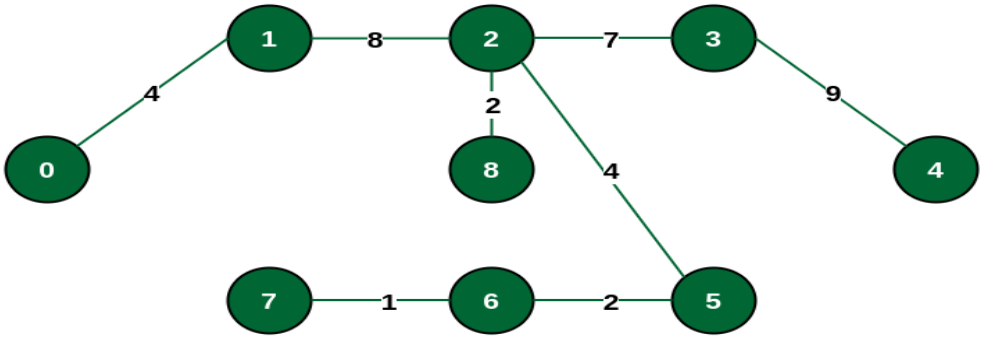
The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = **37**.



The final structure of MST

The structure of the MST formed using the above method

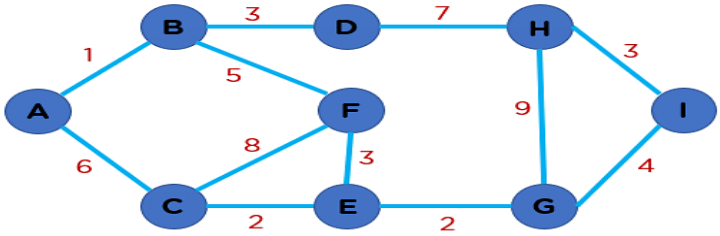
Note: If we had selected the edge {1, 2} in the third step then the MST would look like the following.



Alternative MST structure

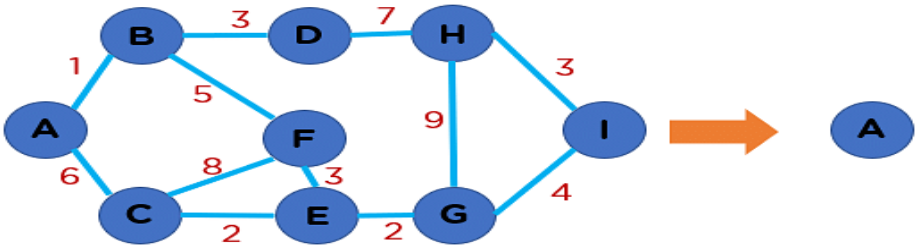
Structure of the alternate MST if we had selected edge {1, 2} in the MST

Example:03- Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-



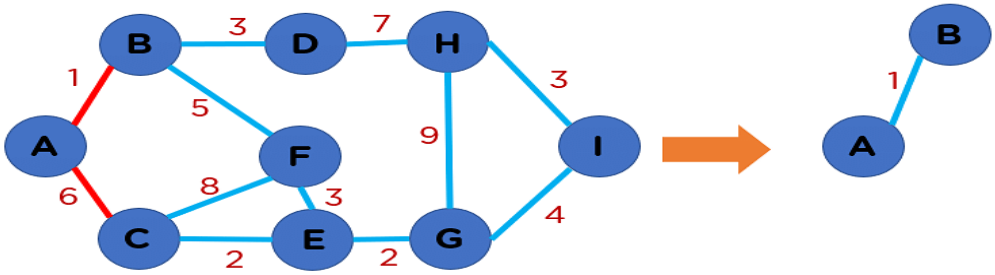
Graph G(V, E)

- Primarily, to begin with the creation of MST, you will choose an arbitrary starting vertex. Let's say node **A** is your starting vertex. This means it will be included first in your tree structure.



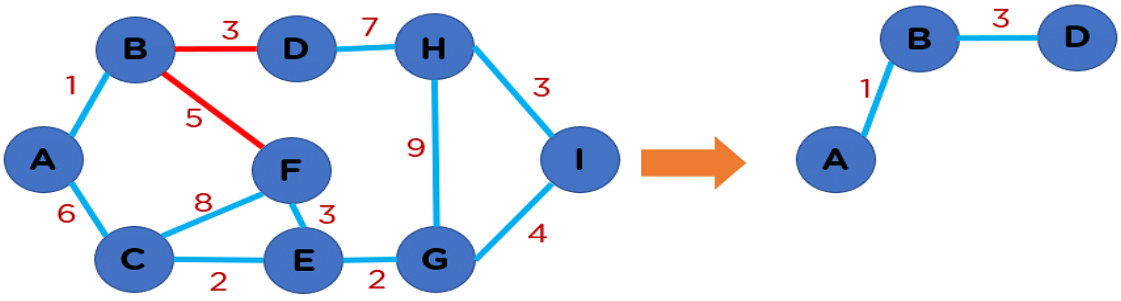
Choose an arbitrary starting vertex.

- After the inclusion of node A, you will look into the connected edges going outward from node A and you will pick the one with a minimum edge weight to include it in your T(V', E') structure.



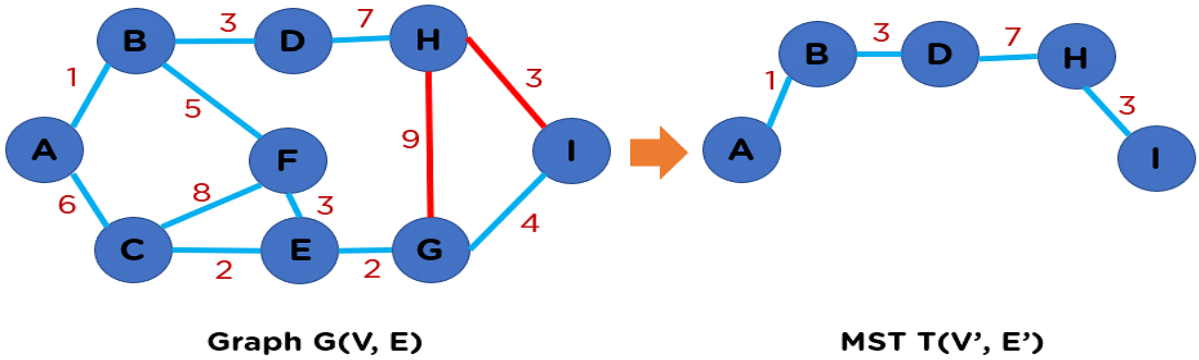
Select the edge with minimal value to include it in MST.

- Now, you have reached node B. From node B, there are two possible edges out of which edge BD has the least edge weight value. So, you will include it in your MST.



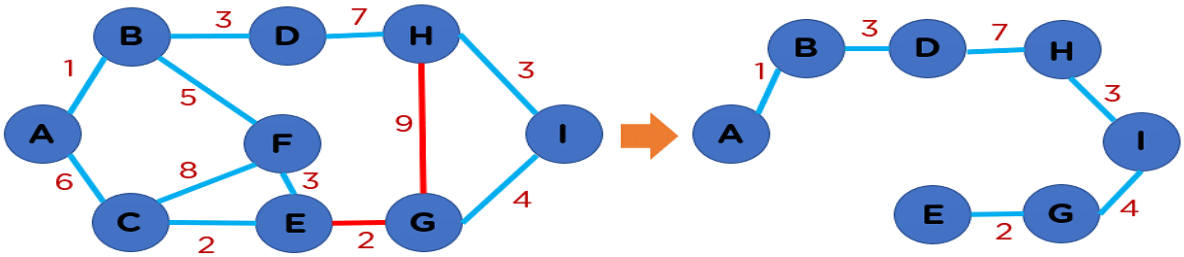
Graph G(V, E) MST T(V', E')

- From node D, you only have one edge. So, you will include it in your MST. Further, you have node H, for which you have two incident edges. Out of those two edges, edge HI has the least cost, so you will include it in MST structure.



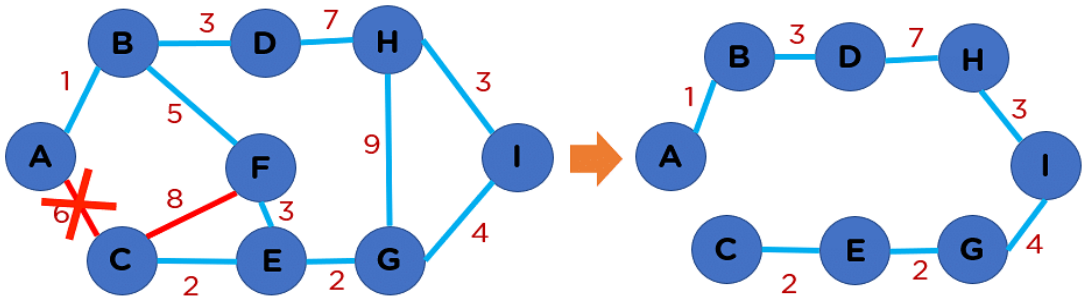
Graph G(V, E) MST T(V', E')

- Similarly, the inclusion of nodes G and E will happen in MST.



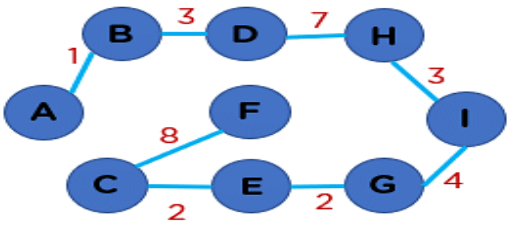
Graph G(V, E) MST T(V', E')

- After that, nodes E and C will get included. Now, from node C, you have two incident edges. Edge CA has the tiniest edge weight. But its inclusion will create a cycle in a tree structure, which you cannot allow. Thus, we will discard edge CA as shown in the image below.



Graph G(V, E) MST T(V', E')

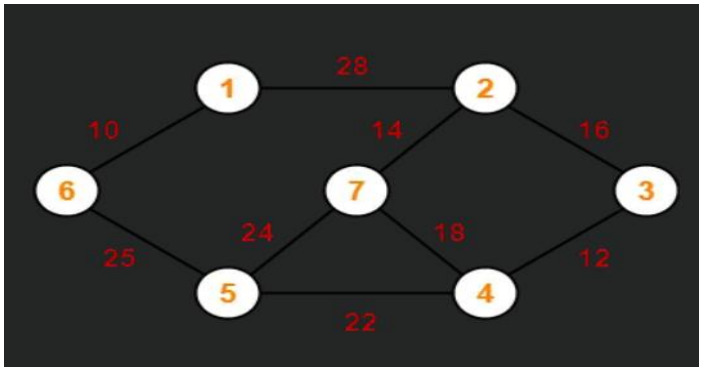
- And we will include edge CF in this minimum spanning tree structure.

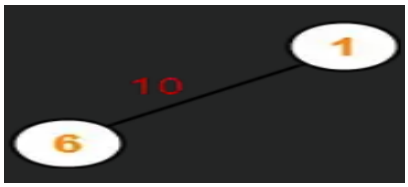


MST T(V', E')

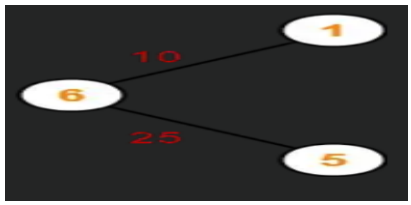
- The summation of all the edge weights in MST T(V', E') is equal to 30, which is the least possible edge weight for any possible spanning tree structure for this particular graph.

Example-4: Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm

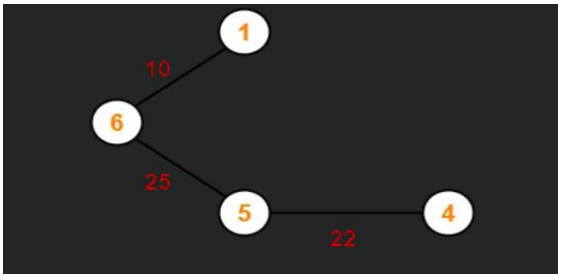




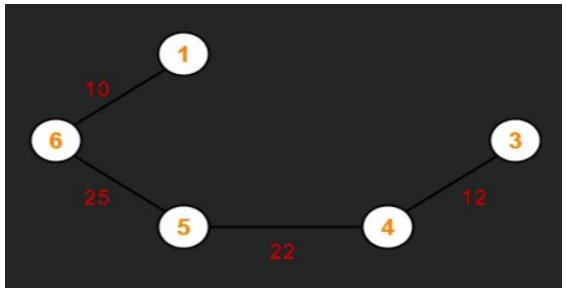
Step:01



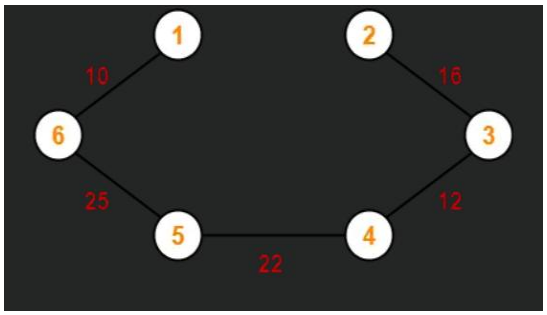
Step:02



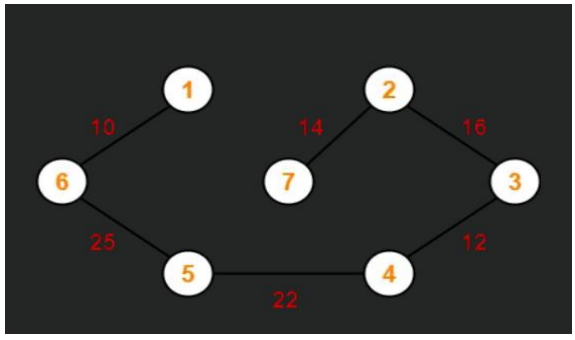
Step:03



Step:04



Step:05



Step:06

Since all the vertices have been included in the MST, so we stop.

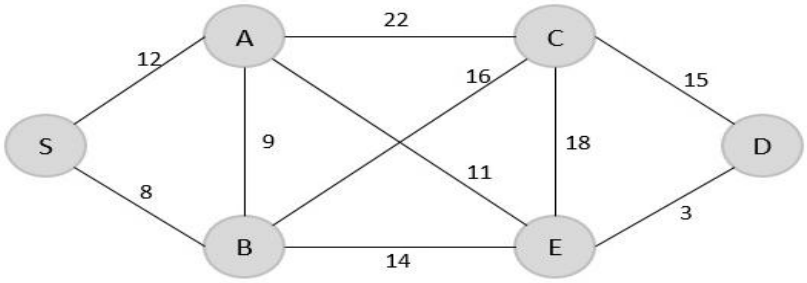
Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

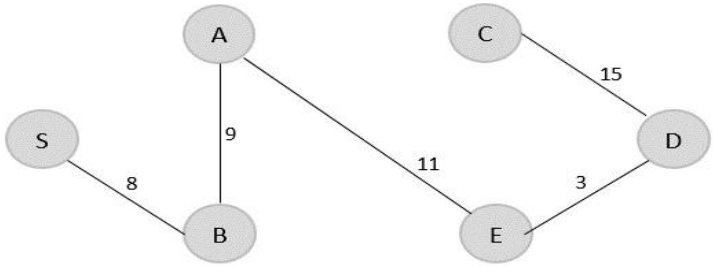
= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

Example-5: Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm



Find the minimum spanning tree using prim's method (greedy approach) for the graph given below with **S** as the arbitrary root.



The minimum spanning tree is obtained with the minimum cost = 46

Kruskal's Algorithm has several advantages:

- It is simple to understand and implement.
- It works well with sparse graphs since it focuses on edges rather than vertices.
- The algorithm can handle disconnected components and find the MST for each connected component.
- It uses efficient data structures like union-find for cycle detection, making it computationally efficient with the time complexity of O(ElogE).

Difference between Prim's Algorithm and Kruskal's Algorithm:

Prim's Algorithm	Kruskal's Algorithm
The tree that we are making or growing always remains connected.	The tree that we are making or growing usually remains disconnected.
Prim's Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest.
Prim's Algorithm is faster for dense graphs.	Kruskal's Algorithm is faster for sparse graphs.

Kruskal's Algorithm Applications:

- Network Design: Kruskal's algorithm can be used to design networks with the least cost. It can be used to find the least expensive network connections that can connect all the nodes in the network.
- Approximation Algorithms: Kruskal's algorithm can be used to find approximate solutions to several complex optimization problems. It can also solve the traveling salesman problem, the knapsack problem, and other NP-hard optimization problems.
- Image Segmentation: Image segmentation is the process of partitioning an image into multiple segments. Kruskal's algorithm can be used to break down an image into its constituent parts in an efficient manner.
- Clustering: Clustering is the process of grouping data points based on their similarity.
- .

Advantages and disadvantages of prim's algorithm:

Advantages of Prim's Algorithm	Disadvantages of Prim's Algorithm
It guarantees to find the minimum spanning tree for any connected, weighted, undirected graph.	It may not work efficiently on sparse graphs, where the number of edges is much smaller than the number of vertices, because of the use of priority queue.
It is a simple algorithm to understand and implement.	It is not suitable for dynamic graphs where the edges can be added or removed frequently, as it requires the entire graph to be present upfront.
It works well on dense graphs, where the number of edges is close to the number of vertices.	It requires the graph to be connected. If the graph is not connected, the algorithm will only find the minimum spanning tree of one of its connected components.

The applications of prim's algorithm are :

- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables

Dijkstra Algorithm:

- Dijkstra's shortest path algorithm is similar to that of Prim's algorithm as they both rely on finding the shortest path locally to achieve the global solution.
- However, unlike prim's algorithm, the dijkstra's algorithm does not find the minimum spanning tree; it is designed to find the shortest path in the graph from one vertex to other remaining vertices in the graph.
- Dijkstra's algorithm can be performed on both directed and undirected graphs.
- Dijkstra algorithm works only for connected graphs.
- Dijkstra algorithm works only for those graphs that do not contain any negative weight edge.
- Since the shortest path can be calculated from single source vertex to all the other vertices in the graph, Dijkstra's algorithm is also called **single-source shortest path algorithm**.

Fundamentals of Dijkstra's Algorithm:

The following are the basic concepts of Dijkstra's Algorithm:

1. Dijkstra's Algorithm begins at the node we select (the source node), and it examines the graph to find the shortest path between that node and all the other nodes in the graph.
2. The Algorithm keeps records of the presently acknowledged shortest distance from each node to the source node, and it updates these values if it finds any shorter path.
3. Once the Algorithm has retrieved the shortest path between the source and another node, that node is marked as 'visited' and included in the path.
4. The procedure continues until all the nodes in the graph have been included in the path. In this manner, we have a path connecting the source node to all other nodes, following the shortest possible path to reach each node.

Algorithm:

Step 1: First, we will mark the source node with a current distance of 0 and set the rest of the nodes to INFINITY.

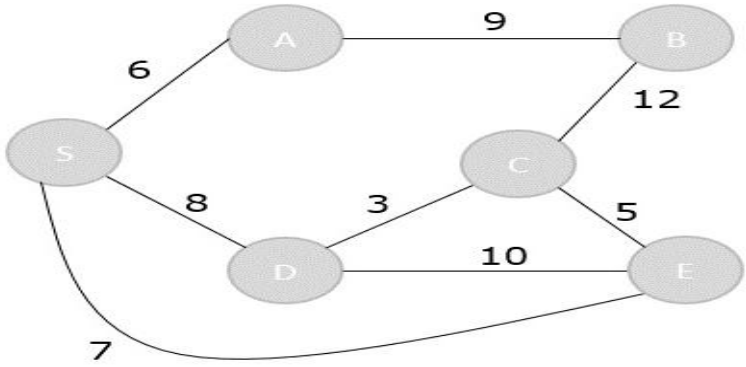
Step 2: We will then set the unvisited node with the smallest current distance as the current node, suppose X.

Step 3: For each neighbor N of the current node X: We will then add the current distance of X with the weight of the edge joining X-N. If it is smaller than the current distance of N, set it as the new current distance of N.

Step 4: We will then mark the current node X as visited.

Step 5: We will repeat the process from **'Step 2'** if there is any node unvisited left in the graph.

Example-1: Using Dijkstra's Algorithm, find the shortest distance from source vertex 'S' to remaining vertices in the following graph-



Step 1:

Initialize the distances of all the vertices as ∞, except the source node S.

Vertex	S	A	B	C	D	E
Distance	0	∞	∞	∞	∞	∞

Now that the source vertex S is visited, add it into the visited array.

visited = {S}

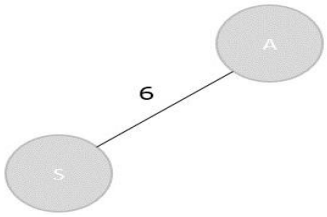
Step 2:

The vertex S has three adjacent vertices with various distances and the vertex with minimum distance among them all is A. Hence, A is visited and the dist[A] is changed from ∞ to 6.

S → A = 6
S → D = 8
S → E = 7

Vertex	S	A	B	C	D	E
Distance	0	6	∞	∞	8	7

Visited = {S, A}



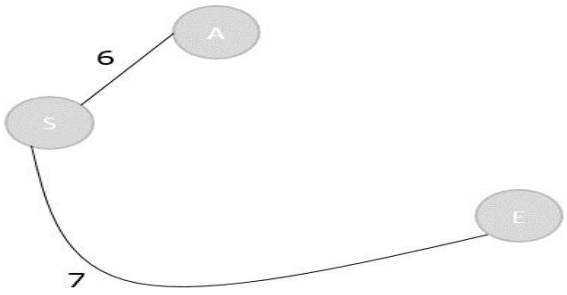
Step 3:

- There are two vertices visited in the visited array, therefore, the **adjacent vertices must be checked for both the visited vertices.**
- Vertex S has two more adjacent vertices to be visited yet: D and E. Vertex A has one adjacent vertex B.
- Calculate the distances from S to D, E, B and select the minimum distance –

S → D = 8 and S → E = 7.
S → B = S → A + A → B = 6 + 9 = 15

Vertex	S	A	B	C	D	E
Distance	0	6	15	∞	8	7

Visited = {S, A, E}



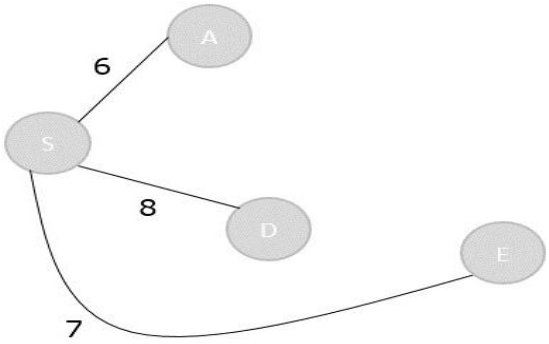
Step 4:

Calculate the distances of the adjacent vertices – S, A, E – of all the visited arrays and select the vertex with minimum distance.

S → D = 8
S → B = 15
S → C = S → E + E → C = 7 + 5 = 12

Vertex	S	A	B	C	D	E
Distance	0	6	15	12	8	7

Visited = {S, A, E, D}



Step 5:

Recalculate the distances of unvisited vertices and if the distances minimum than existing distance is found, replace the value in the distance array.

S → C = S → E + E → C = 7 + 5 = 12
S → C = S → D + D → C = 8 + 3 = 11

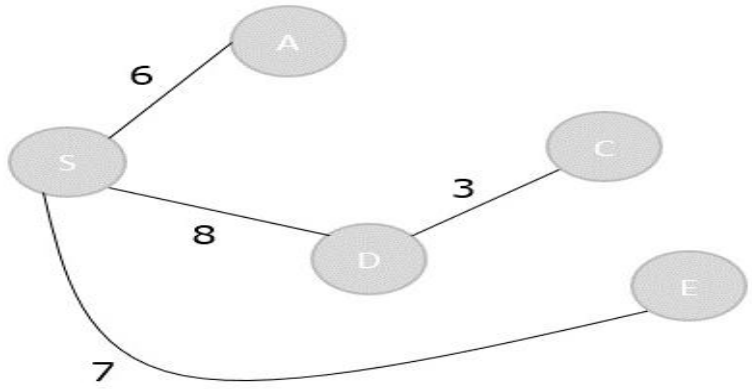
dist[C] = minimum (12, 11) = 11

S → B = S → A + A → B = 6 + 9 = 15
S → B = S → D + D → C + C → B = 8 + 3 + 12 = 23

dist[B] = minimum (15,23) = 15

Vertex	S	A	B	C	D	E
Distance	0	6	15	11	8	7

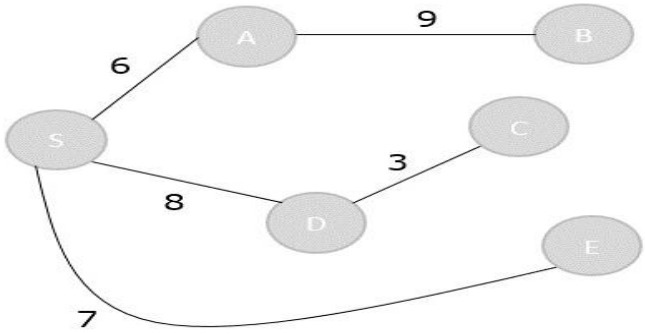
Visited = { S, A, E, D, C}



Step 6:

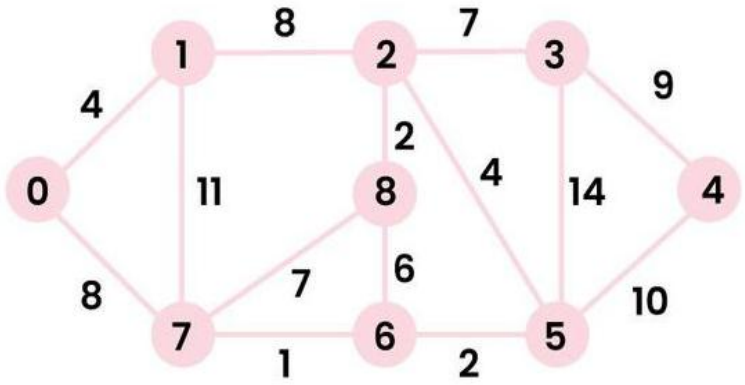
The remaining unvisited vertex in the graph is B with the minimum distance 15, is added to the output spanning tree.

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



The shortest path spanning tree is obtained as an output using the dijkstra's algorithm.

Example-2: Using Dijkstra's Algorithm, find the shortest distance from source vertex '**0**' to all nodes.:

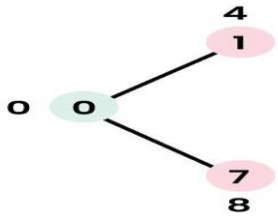


Step 1:

Initialize the distances of all the vertices as ∞, except the source node 0.

Vertex	0	1	2	3	4	5	6	7	8
Distance	0	∞	∞	∞	∞	∞	∞	∞	∞

Now that the source vertex 0 is visited, add it into the visited array.

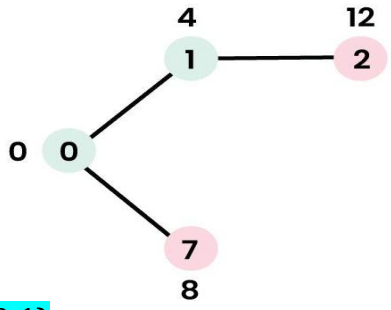


visited = {0}

Step 2:

The vertex 0 has two adjacent vertices with various distances and the vertex with minimum distance among them all is 1. Hence, 1 is visited and the dist[1] is changed from ∞ to 4.

Vertex	0	1	2	3	4	5	6	7	8
Distance	0	4	12	∞	∞	∞	∞	8	∞



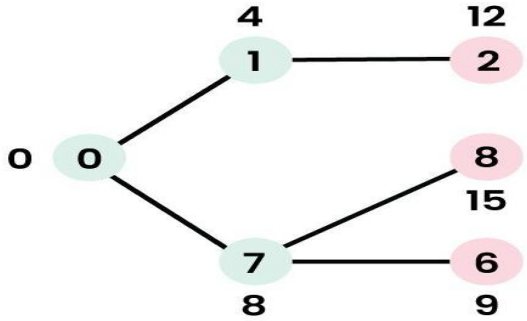
visited = {0,1}

Step 3:

- There are two vertices visited in the visited array, therefore, the adjacent vertices must be checked for **both the visited vertices**.
- Vertex 0 has one more adjacent vertex to be visited yet: 7.
- Vertex 1 has two adjacent vertex 2 and 7.
- Calculate the minimum distance –

0 → 7 =8
0 → 1→7 =15
0 →1→2=12

Vertex	0	1	2	3	4	5	6	7	8
Distance	0	4	12	∞	∞	∞	∞	8	∞

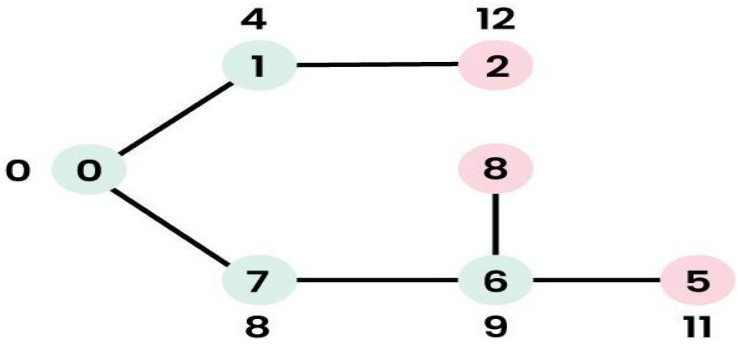


Visited = {0, 1,7}

Step 4:

- There are three vertices visited in the visited array, therefore, the adjacent vertices must be checked for **all visited vertices**.
- Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively)

Vertex	0	1	2	3	4	5	6	7	8
Distance	0	4	12	∞	∞	∞	9	8	15



Visited = {0, 1,7,6}

Step 5:

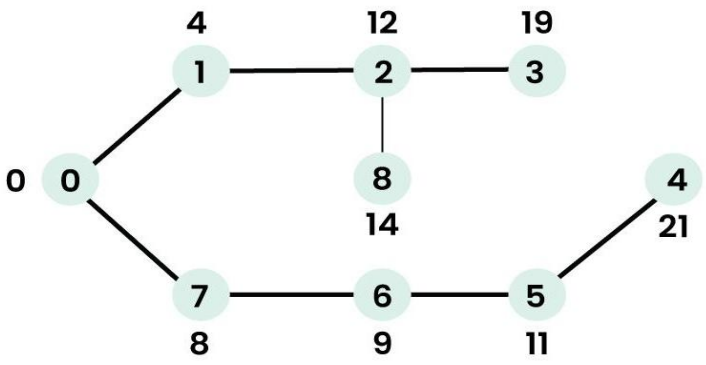
- There are Four vertices visited in the visited array, therefore, the adjacent vertices must be checked for **all visited vertices**.

- 0 → 7→8 =15
- 0 → 1 →2=12
- 0 →7→6→8=15

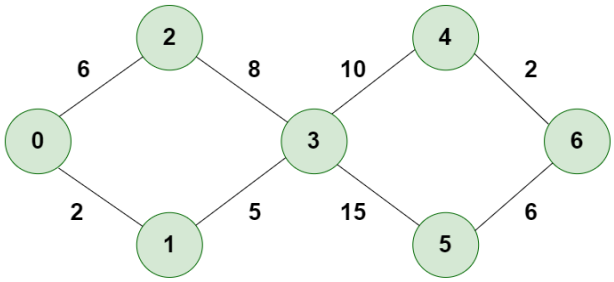
Vertex	0	1	2	3	4	5	6	7	8
Distance	0	4	12	∞	∞	11	9	8	15

Visited = {0, 1,7,6,2}

We repeat the above steps until visited all vertices of the given graph. Finally, we get the following Shortest Path Tree.



Example-3: Using Dijkstra's Algorithm, find the shortest distance from source vertex '**0**' to all nodes.:



Step 1:

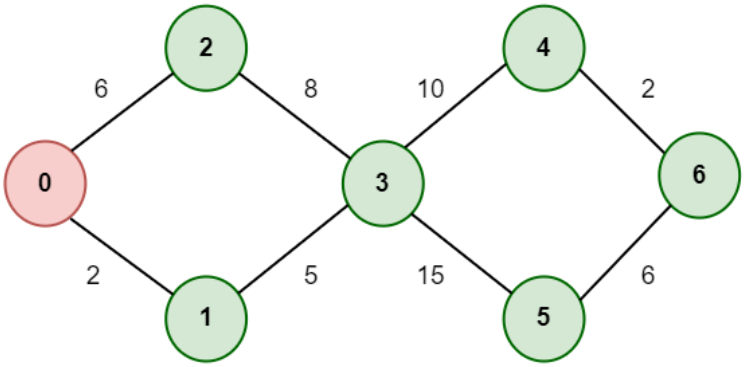
Initialize the distances of all the vertices as ∞, except the source node 0.

Start from Node 0 and mark Node as visited as you can check in below array visited Node is marked red.

Vertex	**0**	1	2	3	4	5	6
Distance	0	∞	∞	∞	∞	∞	∞

Now that the source vertex 0 is visited, add it into the visited array.

visited = {0}

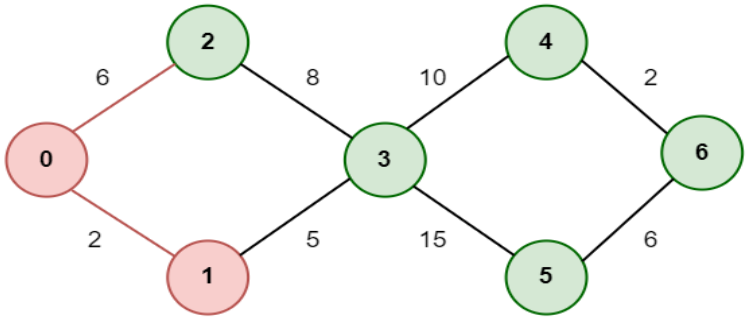


Step 2: Check for adjacent Nodes, Now we have to choices (Either choose Node1 with distance 2 or either choose Node 2 with distance 6) and choose Node with minimum distance.

In this step Node 1 is Minimum distance adjacent Node, so marked it as visited and add up the distance.

Distance: Node 0 -> Node 1 = 2

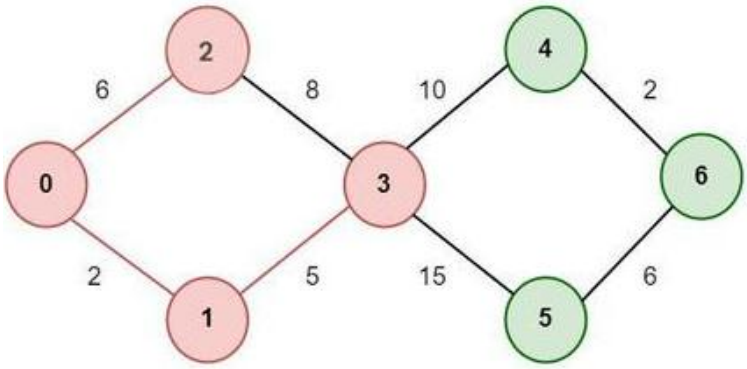
Vertex	**0**	**1**	2	3	4	5	6
Distance	0	2	∞	∞	∞	∞	∞



Step 3: Then Move Forward and check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:

Distance: Node 0 -> Node 1 -> Node 3 = 2 + 5 = 7

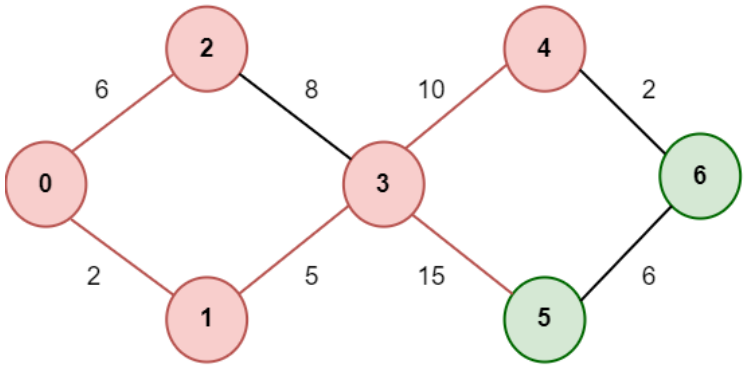
Vertex	**0**	**1**	**2**	**3**	4	5	6
Distance	0	2	6	7	∞	∞	∞



Step 4: Again we have two choices for adjacent Nodes (Either we can choose Node 4 with distance 10 or either we can choose Node 5 with distance 15) so choose Node with minimum distance. In this step **Node 4** is Minimum distance adjacent Node, so marked it as visited and add up the distance.

Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 = 2 + 5 + 10 = 17

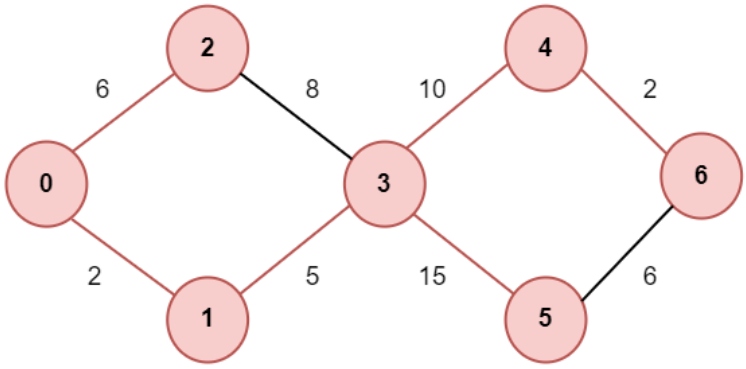
Vertex	**0**	**1**	**2**	**3**	**4**	5	6
Distance	0	2	6	7	17	22	∞



Step 5: Again, Move Forward and check for adjacent Node which is **Node 6**, so marked it as visited and add up the distance, Now the distance will be:

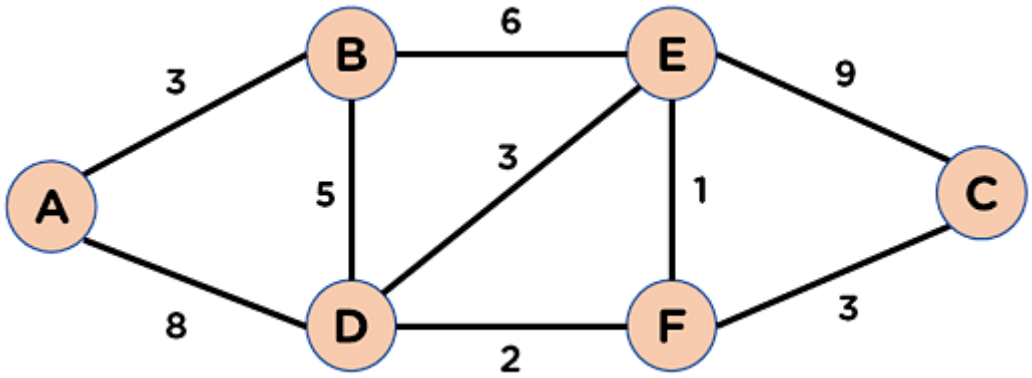
Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 -> Node 6 = 2 + 5 + 10 + 2 = 19

Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Distance	0	2	6	7	17	22	19



So, the Shortest Distance from the Source Vertex is 19 which is optimal one

Example-4: Using Dijkstra's Algorithm, find the shortest distance from source vertex '**A**' to 'C'



Applications of Dijkstra's Algorithm:

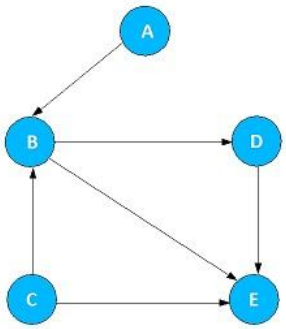
- **Google maps** uses Dijkstra algorithm to show shortest distance between source and destination.
- In **computer networking**, Dijkstra's algorithm forms the basis for various routing protocols, such as OSPF (Open Shortest Path First) and IS-IS (Intermediate System to Intermediate System).
- Transportation and traffic management systems use Dijkstra's algorithm to optimize traffic flow, minimize congestion, and plan the most efficient routes for vehicles.
- Airlines use Dijkstra's algorithm to plan flight paths that minimize fuel consumption, reduce travel time.
- Dijkstra's algorithm is applied in electronic design automation for routing connections on integrated circuits and very-large-scale integration (VLSI) chips.

Difference between Minimum Spanning Tree and Shortest Path:

Basic Parameters	Minimum Spanning Tree	Shortest Path
Definition	A tree that spans all vertices, while minimizing total weight.	The path with the lowest accumulated weight or distance.
Main Objective	Connecting all nodes together with minimal total weight.	It finds the most efficient route between the specific routes.
Key Property	It contains no cycles or loops.	Directed or undirected edges
Algorithm Examples	Some of the examples are Kruskal's and Prim's.	Some examples are Dijkstra's and Bellman-Ford's.
Advantages	It is used for wide purposes like the network design for laying cables and also merges the data in unsupervised learning.	The connection between the two points can be chosen very efficiently by the shortest path and it also helps the people in navigating the end location with minimal distance.
Disadvantages	When the network design is large, it becomes too tedious.	It finds the shortest path, but it may not be a unique one.

ASSIGNMENT QUESTIONS

1. Suppose you want to store a graph where edges frequently change (insertions/deletions). Which representation is better — adjacency matrix or adjacency list?
2. Consider the following directed graph:



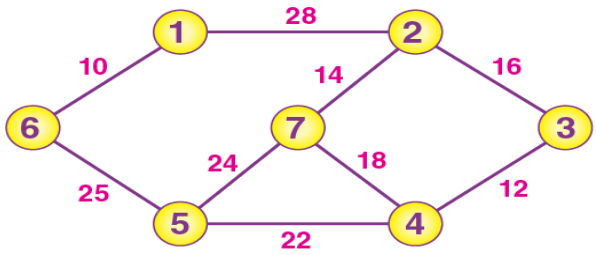
Find the **degree** of each vertex A, B, C, D.

3. In a **social media platform**, the graph is represented as a **directed graph**:
   - An edge A → B means *"A follows B."*

     If a user **X** has:
     - **In-degree = 500**
     - **Out-degree = 10**

   What does this say about X's popularity and following behaviour?
4. A graph has 10 vertices and 15 edges. You are asked to build a spanning tree. How many edges must you remove at minimum to form a spanning tree?
5. In a road network of 12 cities, engineers first construct a spanning tree to ensure all cities are connected with the minimum number of roads. Later, they add one extra road between two cities that were already connected.

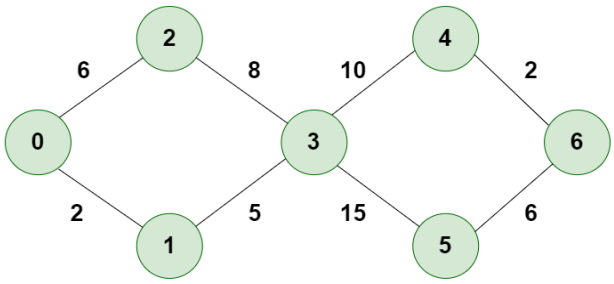
   What problem does this addition create in the spanning tree, and how can it be resolved while still maintaining the spanning tree property?
6. Consider the following weighted graph



Apply **Kruskal's algorithm and find the** total cost of the Minimum Spanning Tree (MST).

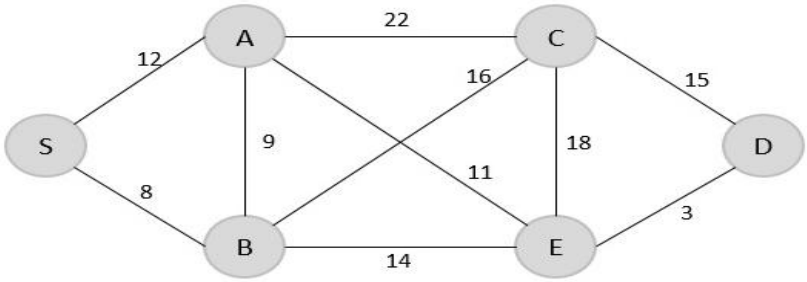
SECTION-B

1. A delivery service must send goods from **warehouse (0)** to all shops. The graph represents roads (weights = travel time in minutes):



   a) Use **Dijkstra's Algorithm** to compute the shortest path from warehouse 0 to all shops.
   b) What is the shortest time from 0 to 6?
   c) If edge (1-2) had weight -1, would Dijkstra's still work? Why/why not?

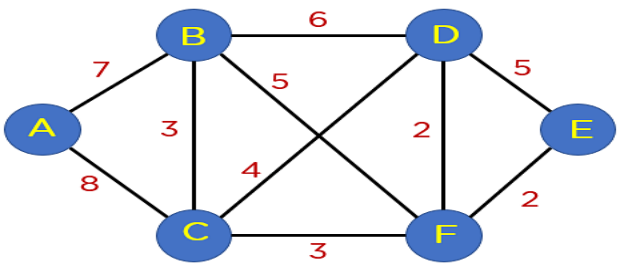
2. A company has **5 offices** that need to be connected with minimum cabling cost. The costs (in lakhs) are shown below:



   a) Use **Prim's Algorithm**, starting from node S, to find the **Minimum Spanning Tree (MST)** step by step.
   b) List the order in which edges are selected.
   c) What is the **total minimum cost** of connecting all offices?
   d) If an additional office (T) is added and connected to (E) with cost 4, how does MST change?

3. The government of a newly planned **Smart City** is tasked with connecting **6 major sectors** with roads. The goal is to make sure that **all sectors are connected with the minimum possible construction cost**.

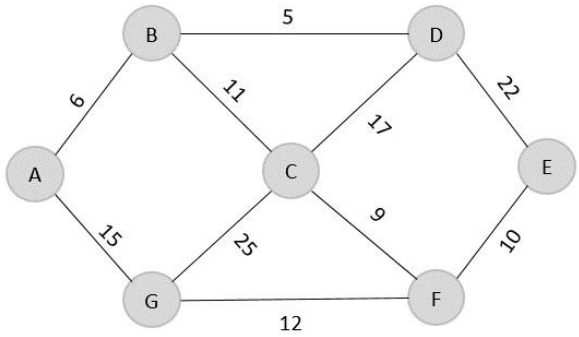
The planning team prepares the following cost map (in crores of rupees):



a) Apply **Kruskal's Algorithm** step by step starting from city (A).
b) Show the final **set of edges in MST** and calculate the **minimum road construction cost**.
c) After adding the metro rail line (B-E) with cost 2, how does the MST change?
d) Explain why Kruskal's Algorithm is better suited than Prim's in this case, given the graph's structure.

4. The Indian Railways wants to build a **new railway network** connecting **6 cities**. Each railway track has a construction cost (in crores of rupees). The goal is to minimize cost but still ensure **all cities are connected**.

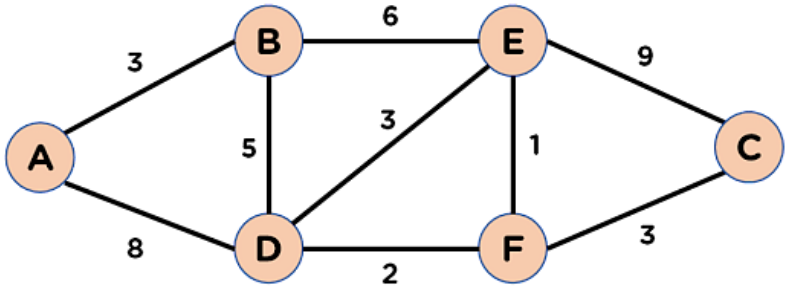
The railway department creates this cost graph:



a) Apply **Prim's Algorithm** step by step starting from city (A).
b) Apply **Kruskal's Algorithm** step by step.
c) Compare the MSTs obtained by both algorithms:
   - ✓ Are the set of edges same?
   - ✓ Is the total cost same?
d) If railway line (C-G) must be compulsorily included, how does that affect the MST?

5. A large **E-commerce company (AMAZON)** is planning its **last-mile delivery system** in a metropolitan city. The company has multiple warehouses, local hubs, and customer areas connected through city roads. Due to heavy traffic and variable road conditions, each road has a different **travel time (in minutes)**.

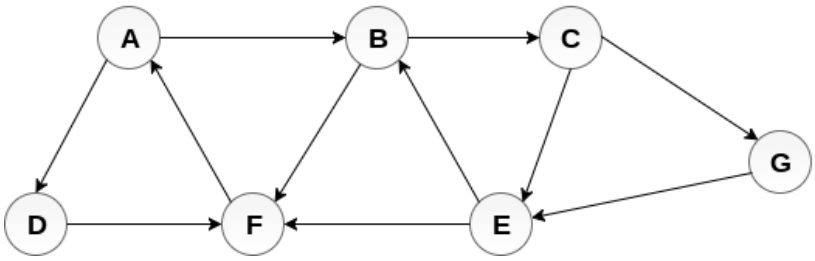
The city is represented as the following **weighted graph**:



a) Using **Dijkstra's Algorithm**, determine the **shortest path** from A to G.
b) What is the **minimum travel time** required?

c) If traffic suddenly increases on road **E–F**, explain in 2–3 lines how Dijkstra's Algorithm would help the system **recalculate and adapt routes in real time**.

6. A social media platform wants to analyse the influence of its users. Each user is represented as a vertex in a graph, and an edge between two users indicates a friendship. Some users are highly connected (influencers), while others have few connections. The platform wants to find **friends-of-friends up to 2 levels** for a new user to suggest connections.



a) Apply **BFS** starting from user **A**. Show step-by-step traversal order.
b) Identify all users at **distance 2** from A (friends-of-friends).
c) Explain how BFS helps in suggesting new connections efficiently.