UNIT-5

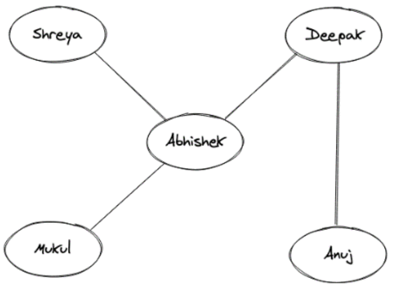
GRAPHS: Introduction, definitions and basic terminologies, representations of graphs, graph traversals, Minimum Spanning Trees and applications.

INTRODUCTION:

- Graphs are **non-linear data structures** comprising a finite set of nodes and edges. The nodes are the elements, and edges are ordered pairs of connections between the nodes.
- A **non-linear data structure** is one where the elements are not arranged in sequential order.
  - For example, an array is a linear data structure because it is arranged one after the other.
  - You can traverse all the elements of an array in a single run. Such is not the case with non-linear data structures.
  - The elements of a non-linear data structure are arranged in multiple levels like trees, graphs etc.

EXAMPLE:

- The best example of graphs in the real world is **Facebook or a network of socially connected people**. Each person on **Facebook** is a node and is connected through edges. Thus, a is a friend of B. B is a friend of C, and so on.



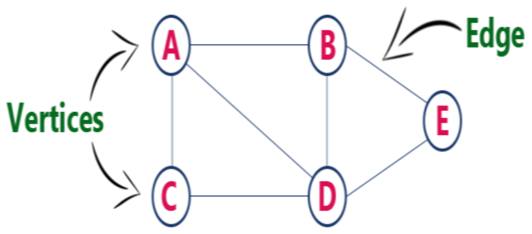
The names here are equivalent to the nodes of a graph and the lines that define the relationship of "knowing each other" is simply the equivalent of an edge of a graph : "Abhishek" knows "Mukul" and "Mukul" knows "Abhishek".

DEFINITION:

- "**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

The following is a graph with 5 vertices and 7 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)}.



BASIC TERMINOLOGY:

1. 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.

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

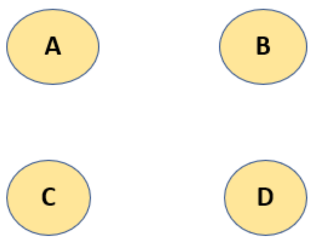
Edges are three types.

- **Undirected Edge -** An undirected edge is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
- **Directed Edge -** A directed edge 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).
- **Weighted Edge -** A weighted edge is a edge with value (cost) on it.

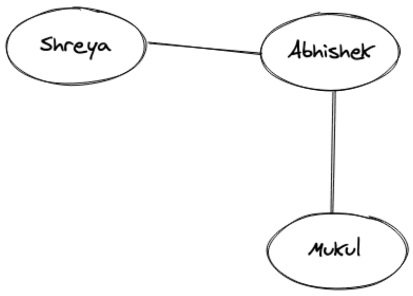
3. End vertices or Endpoints: The two vertices joined by edge are called end vertices (or endpoints) of that edge.
4. Origin: If a edge is directed, its first endpoint is said to be the origin of it.
5. Destination: If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.
6. Outgoing Edge: A directed edge is said to be outgoing edge on its origin vertex.
7.Incoming Edge: A directed edge is said to be incoming edge on its destination vertex.
8. Degree: Total number of edges connected to a vertex is said to be degree of that vertex.
9. Indegree: Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
10. Outdegree: Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.
11. Parallel edges or Multiple edges: If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.
12. Self-loop: Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.
13. Simple Graph: A graph is said to be simple if there are no parallel and self-loop edges.
14. 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.

TYPES OF GRAPHS:

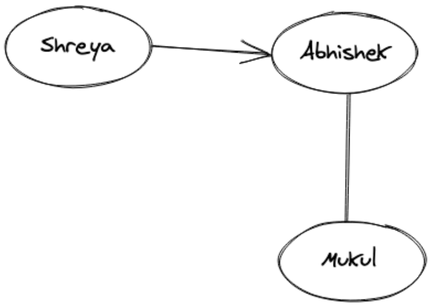
1. Null Graphs: A graph is said to be null if there are no edges in that graph. A pictorial representation of the null graph is given below:



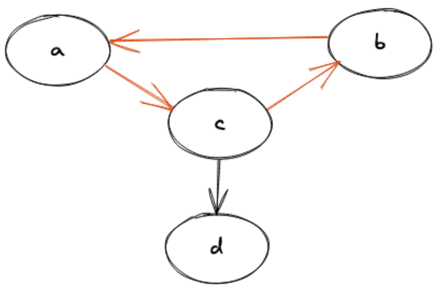
2. Undirected Graphs: These types of graphs where the relation is bi-directional or there is not a single direction, are known as Undirected graphs



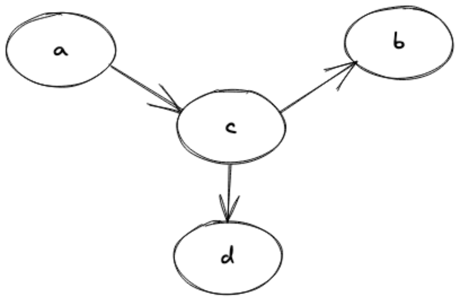
3. Directed Graphs: The edges with arrows basically denote the direction of the relationship and such graphs are known as directed graphs.



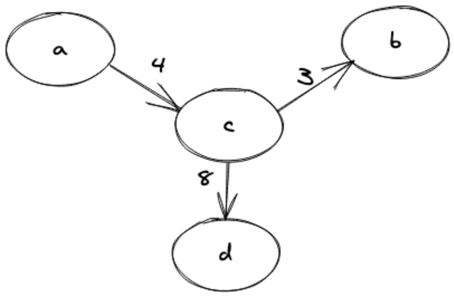
4. Cyclic Graphs: A graph that contains at least one node that traverses back to itself is known as a cyclic graph. In simple words, a graph should have at least one cycle formation for it to be called a cyclic graph.



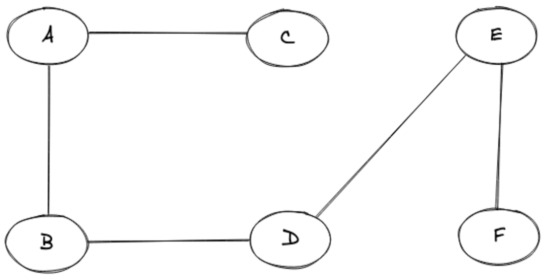
5. Acyclic Graph: A graph where there's no way we can start from one node and can traverse back to the same one, or simply doesn't have a single cycle is known as an acyclic graph.



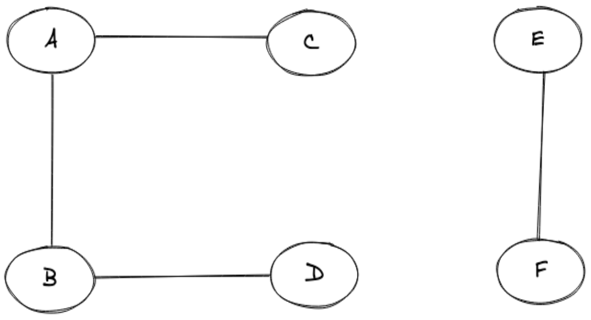
6. Weighted Graph: When the edge in a graph has some weight associated with it, we call that graph as a weighted graph. The weight is generally a number that could mean anything, totally dependent on the relationship between the nodes of that graph.



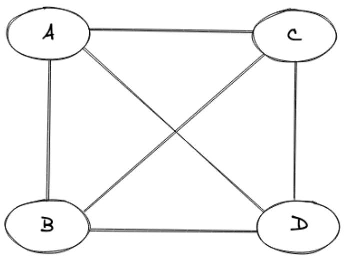
7. Connected Graph: A graph where we have a path between every two nodes of the graph is known as a connected graph. A path here means that we are able to traverse from a node "A" to say any node "B“.



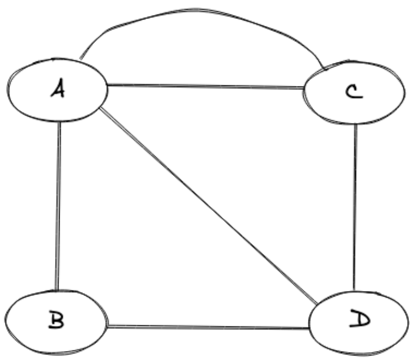
8. Disconnected Graph: A graph that is not connected is simply known as a disconnected graph. In a disconnected graph, we will not be able to find a path from between every two nodes of the graph.



9. Complete Graph: A graph is said to be a complete graph if there exists an edge for every pair of vertices(nodes) of that graph



10. Multigraph: A graph is said to be a multigraph if there exist two or more than two edges between any pair of nodes in the graph.

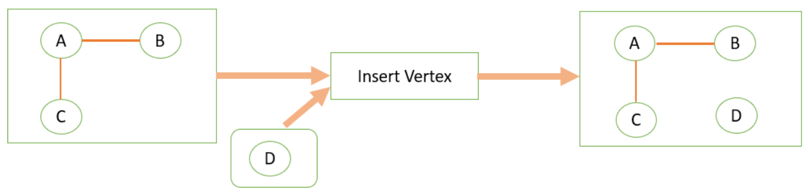


OPERATIONS ON GRAPH:

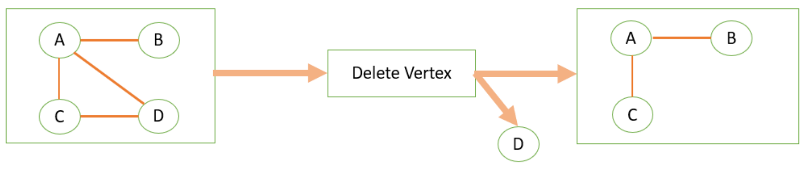
There are 5 major primitive graph operations that provide the basic modules needed to maintain a graph.

1. Insert a vertex
2. Delete a vertex
3. Insert an edge
4. Delete an edge
5. Traverse

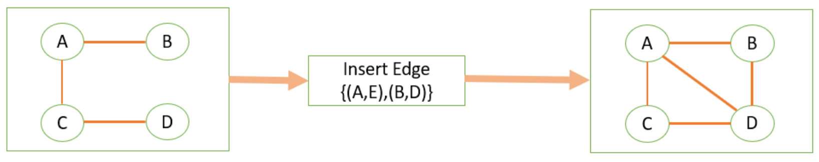
Insert Vertex



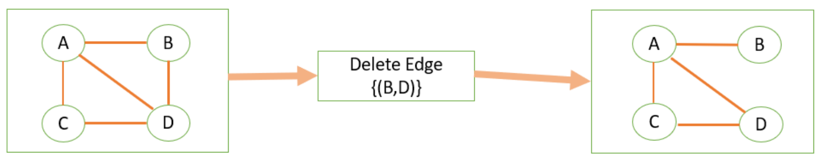
Delete Vertex



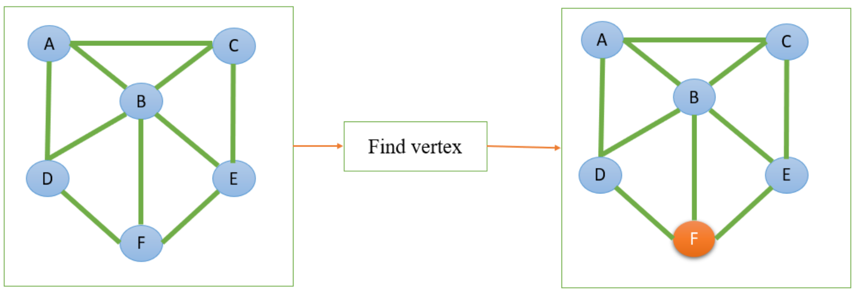
Insert Edge



Delete Edge



Traverse vertex

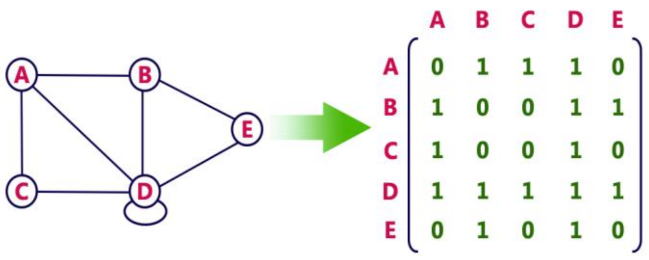


Representations of graphs:

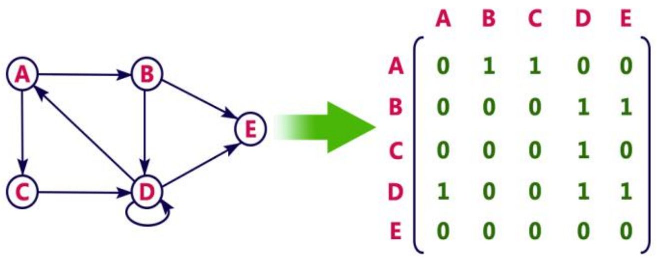
Graph data structure is represented using following representations...

1. **Adjacency Matrix:**

- In this representation, the graph is represented using a matrix of size total number of vertices by 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.



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

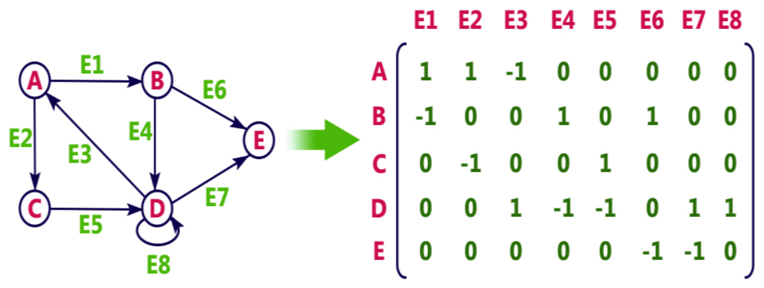


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

2. 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 represents edges. This matrix is filled with 0 or 1 or -1.

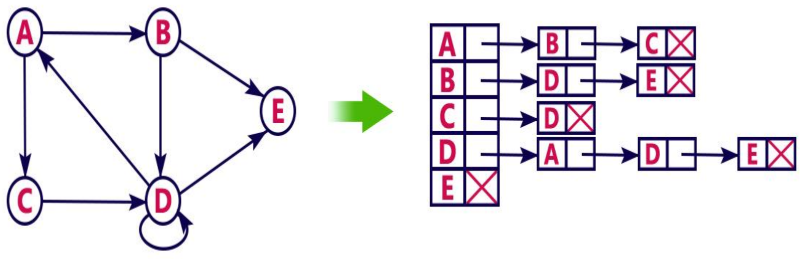
- Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

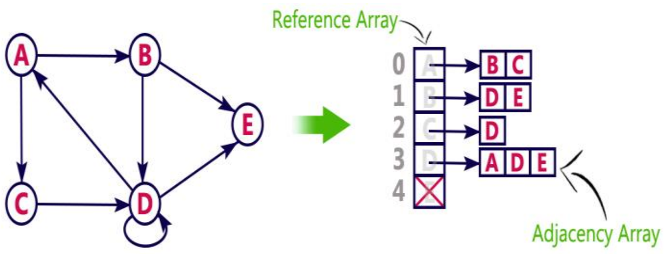


$$\begin{array}{c|cccccccc} & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\ \hline A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

3. Adjacency List: In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...





GRAPH TRAVERSAL TECHNIQUES:

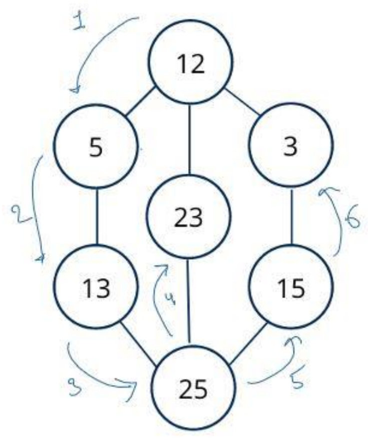
- Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

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

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

DFS (Depth first search)

- DFS stands for Depth First Search, is one of the graph traversal algorithms that use Stack data structure.
- In DFS Traversal go as deep as possible of the graph and then backtrack once reached a vertex that has all its adjacent vertices already visited.
- Depth First Search (DFS) algorithm traverses a graph in a depth-word motion and uses a stack data structure to remember to get the next vertex to start a search when a dead end occurs in any iteration.



Algorithm:

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.

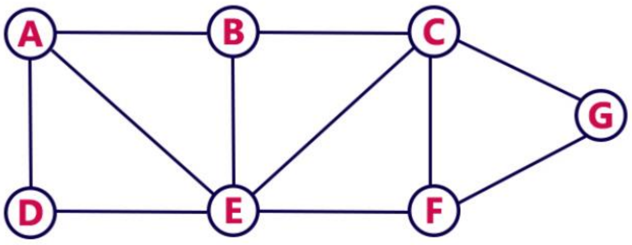
Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

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.

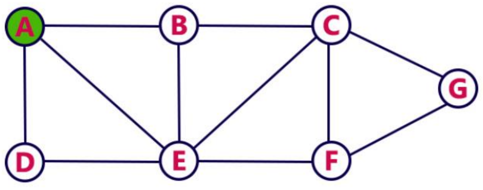
Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

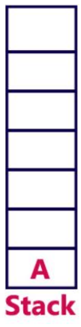
Example:



Step 1:

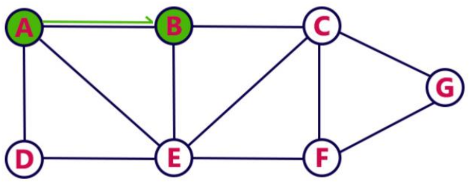
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

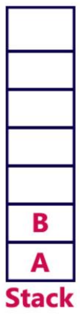




Step 2:

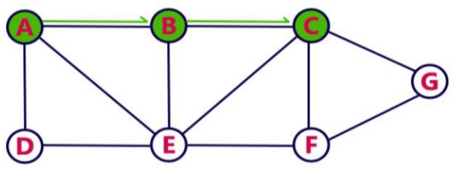
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

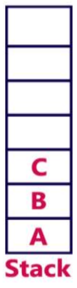




Step 3:

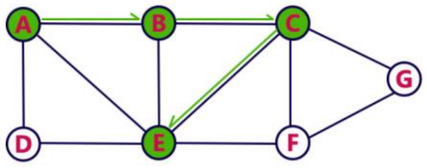
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

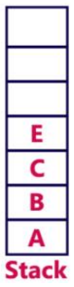




Step 4:

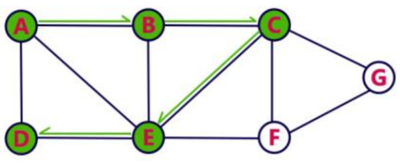
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

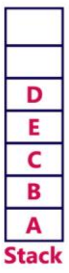




Step 5:

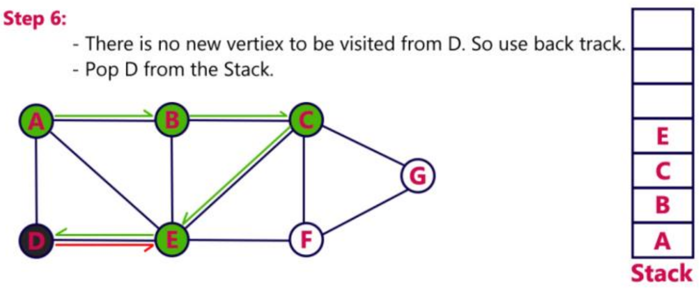
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack





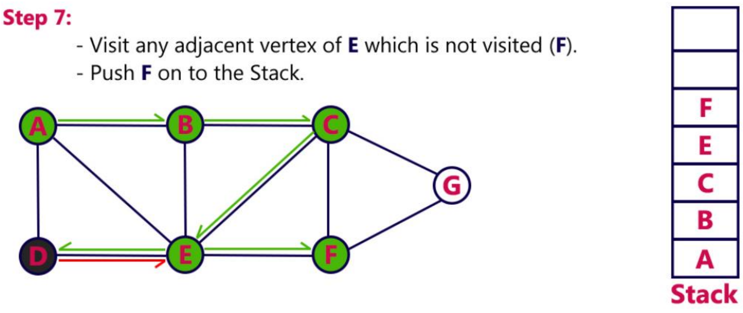
Step 6:

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



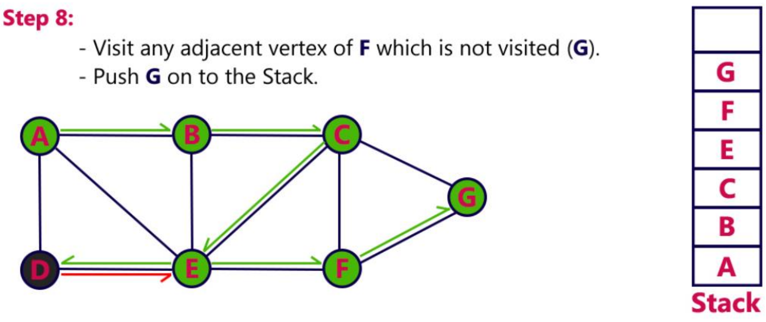
Step 7:

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



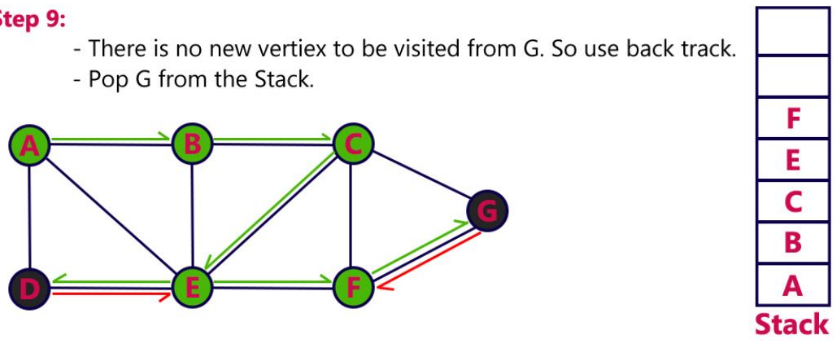
Step 8:

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



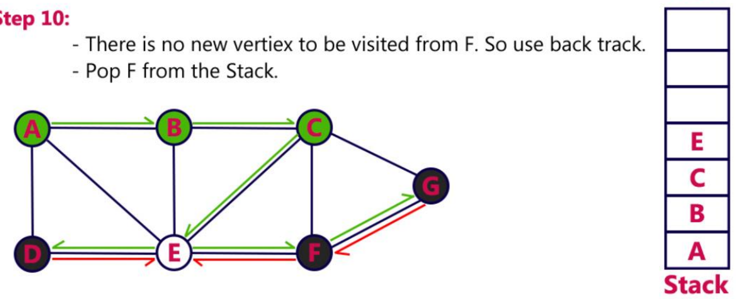
Step 9:

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



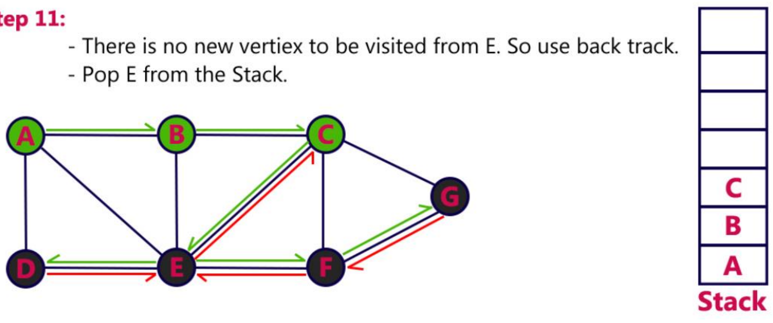
Step 10:

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



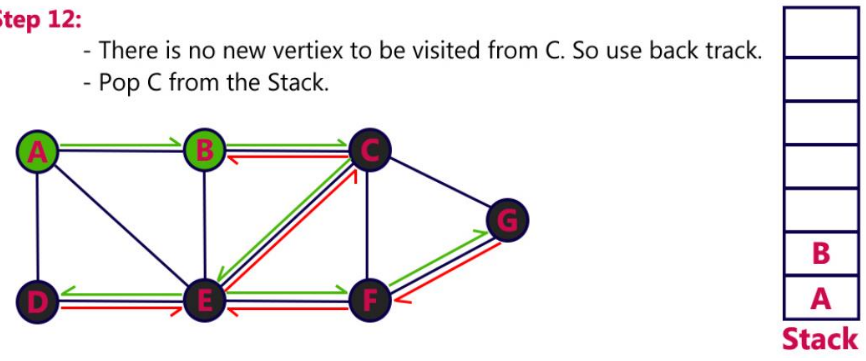
Step 11:

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



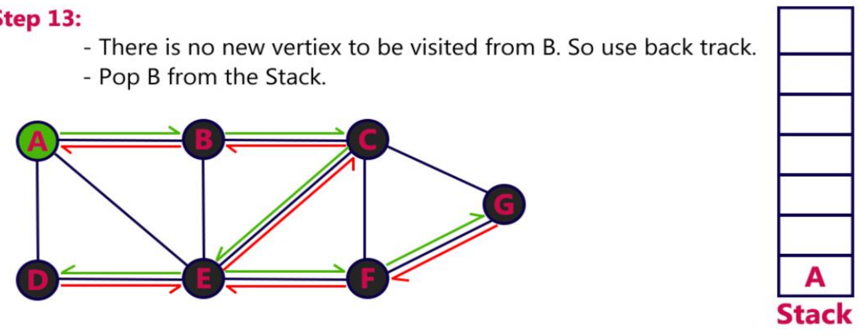
Step 12:

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



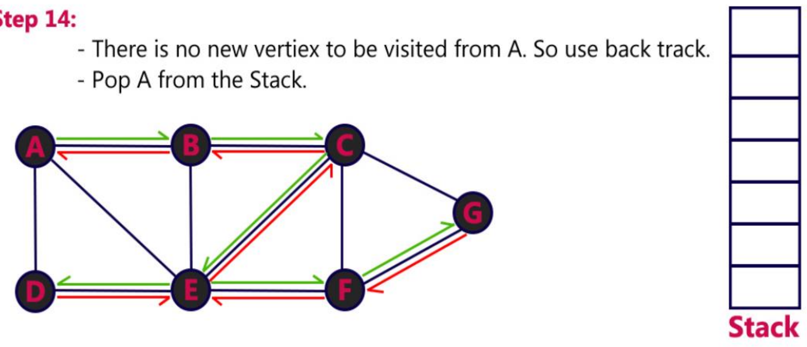
Step 13:

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

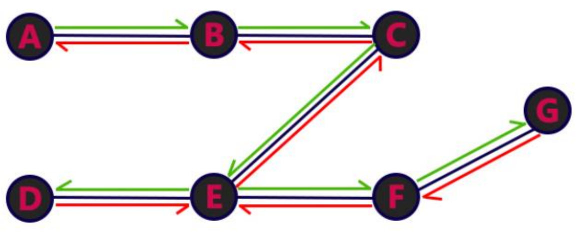


Step 14:

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

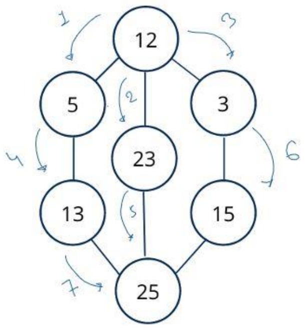


- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



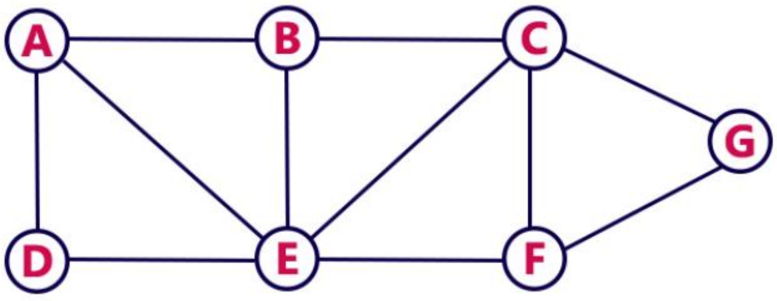
BFS (Breadth first search):

Breadth-first search graph traversal techniques use a queue data structure as an auxiliary data structure to store nodes for further processing. The size of the queue will be the maximum total number of vertices in the graph.



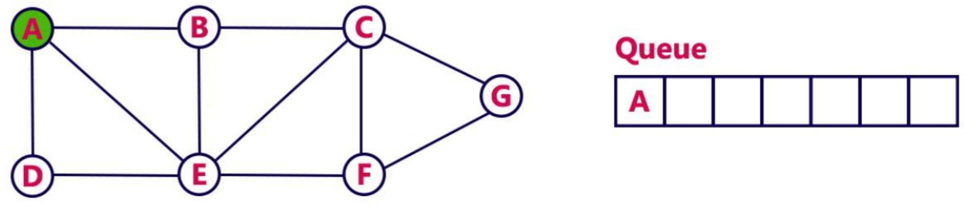
Algorithm

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



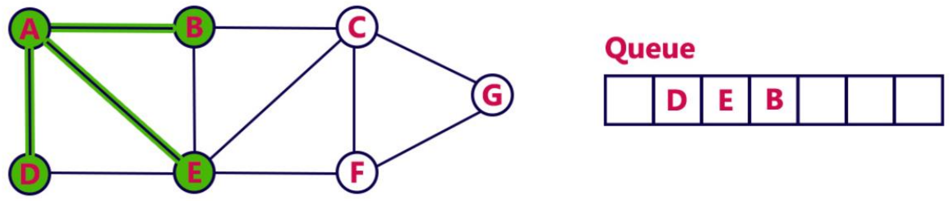
Step 1:

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



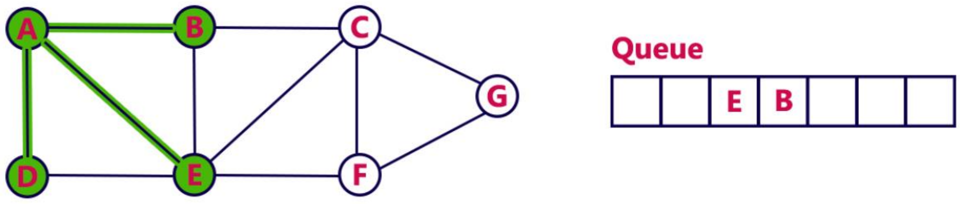
Step 2:

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



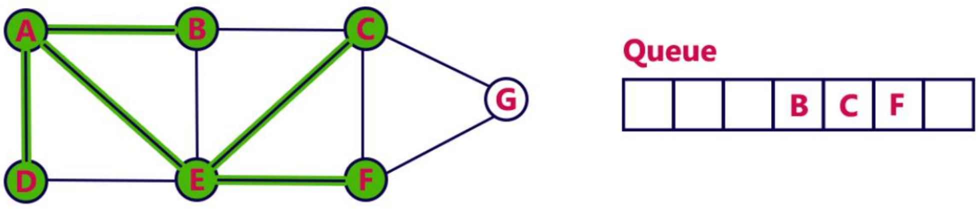
Step 3:

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



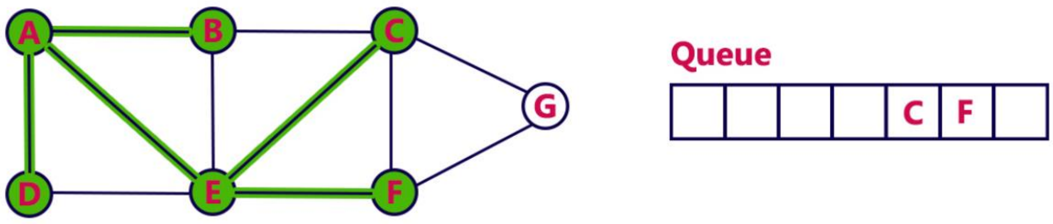
Step 4:

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



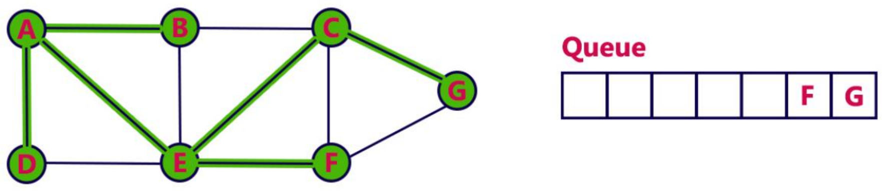
Step 5:

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



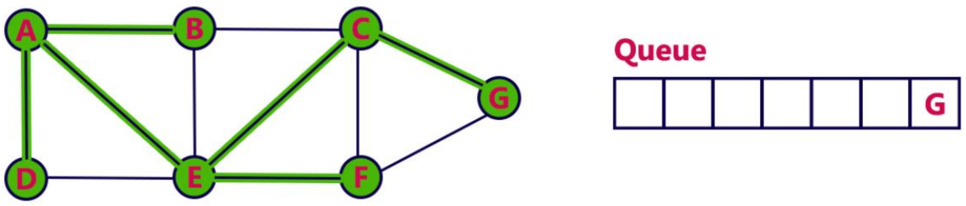
Step 6:

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



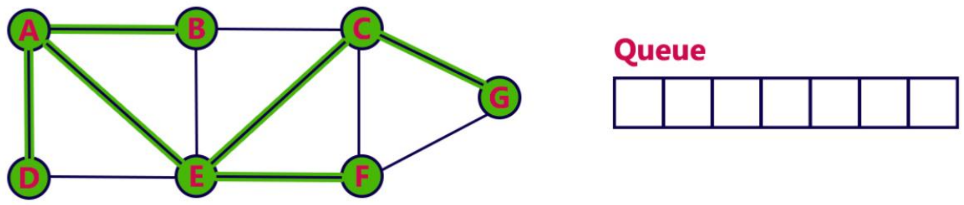
Step 7:

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

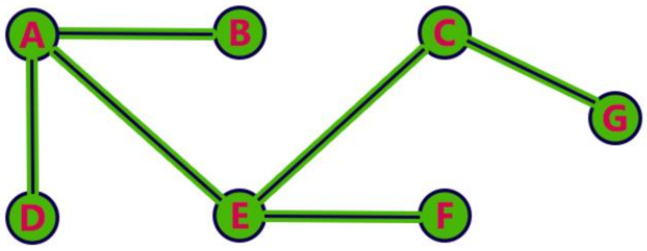


Step 8:

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



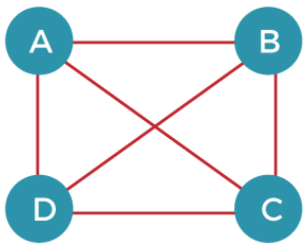
- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Minimum Spanning Tree:

What is spanning tree?

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges



- This graph can be represented as G(V, E), where 'V' is the number of vertices, and 'E' is the number of edges.
- The spanning tree of the above graph would be represented as G\`(V\`, E\`).
- In this case, V\` = V means that the number of vertices in the spanning tree would be the same as the number of vertices in the graph, but the number of edges would be different.
- The number of edges in the spanning tree is the subset of the number of edges in the original graph

Conditions for spanning tree

- The number of vertices in the spanning tree would be the same as the number of vertices in the original graph.

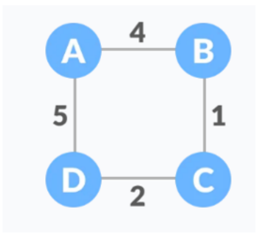
  V\` = V

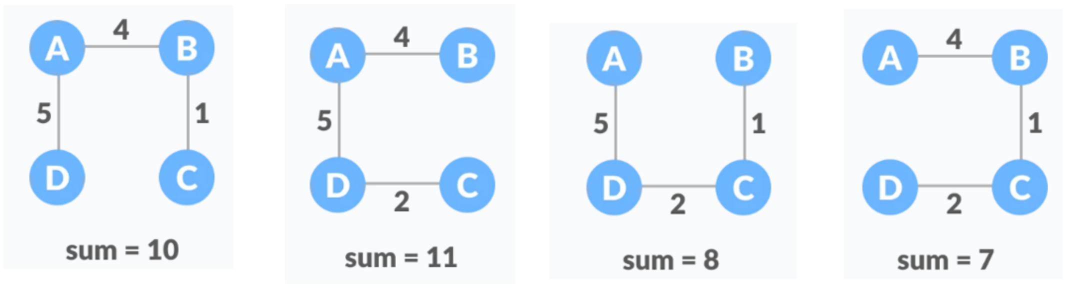
- The number of edges in the spanning tree would be equal to the number of edges minus 1.

  E\` = |V| - 1

- The spanning tree should not contain any cycle.
- The spanning tree should not be disconnected.
- There are two famous algorithms for finding the Minimum Spanning Tree:
  - Kruskal’s Algorithm
  - Prim’s algorithm

Example





Minimum Spanning Tree – (1) Prim's algorithm:

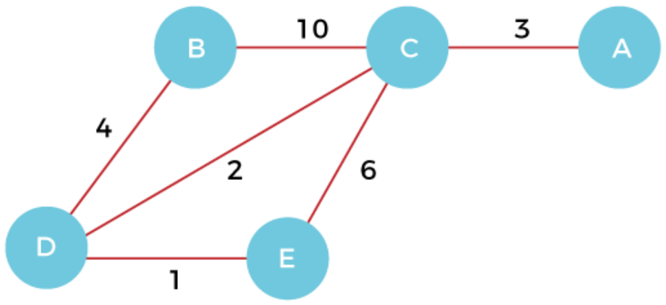
- Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached.

The steps to implement the prim's algorithm are given as follows -

- Step1 : First, we have to initialize an MST with the randomly chosen vertex.
- Step 2: Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Step 3: Repeat step 2 until the minimum spanning tree is formed

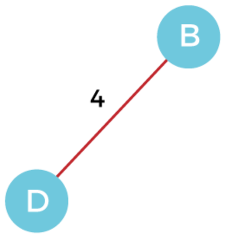
Example:

let's see the working of prim's algorithm using an example

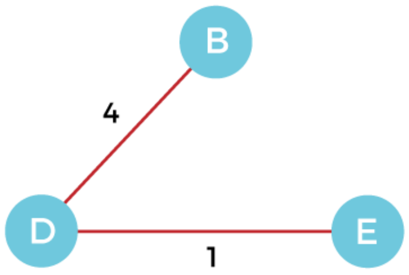


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B

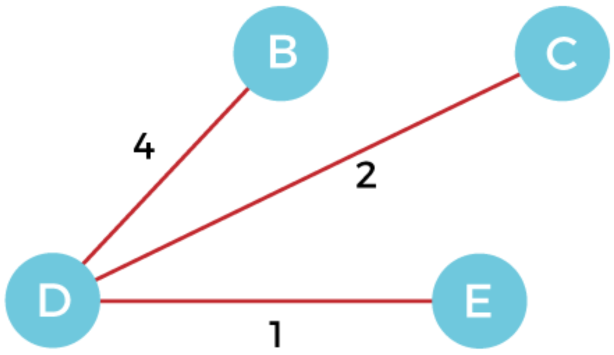
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



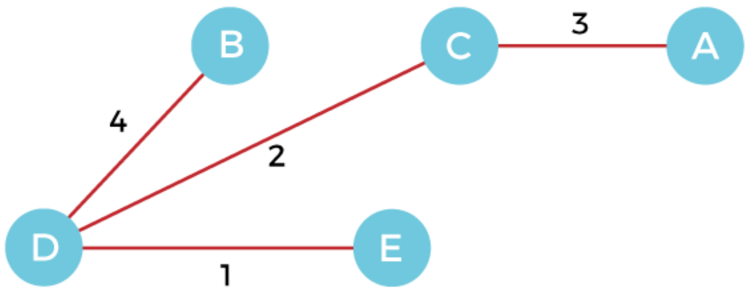
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



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.

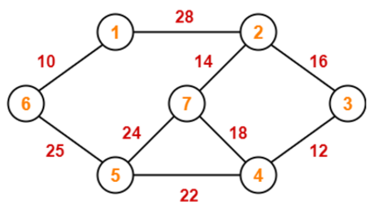


So, the graph produced in step 5 is the minimum

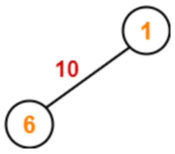
spanning tree of the given graph. The cost of the MST = 4 + 2 + 1 + 3 = 10 units

Example 2:

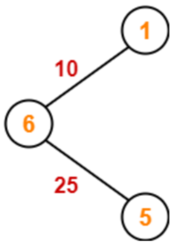
Example 2:



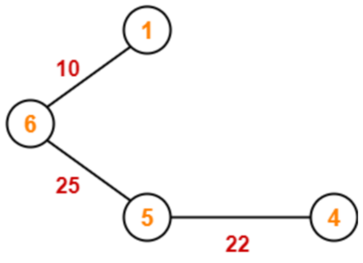
Step-01:



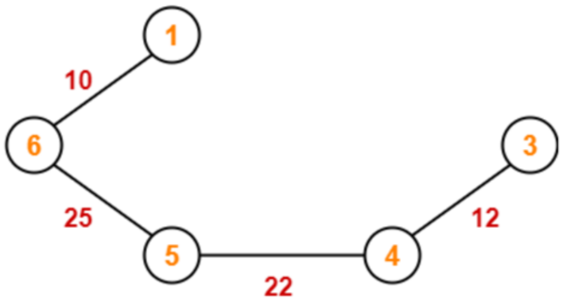
Step-02:



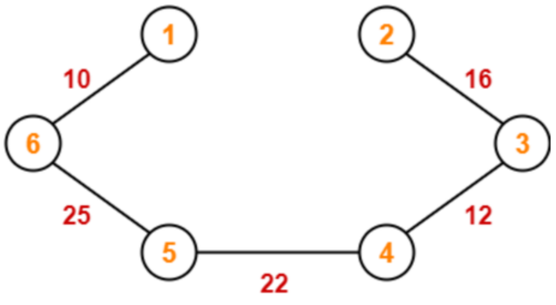
Step-03:



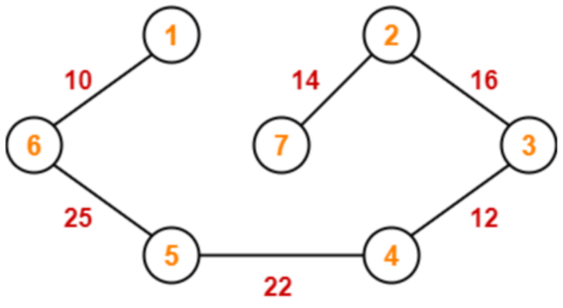
Step-04:



Step-05:



Step-06:



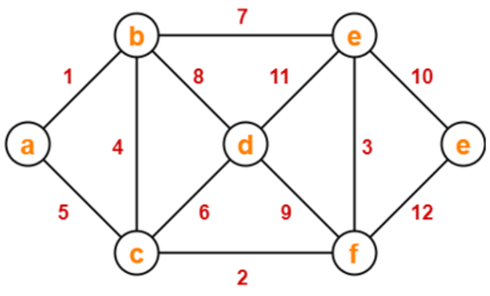
Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

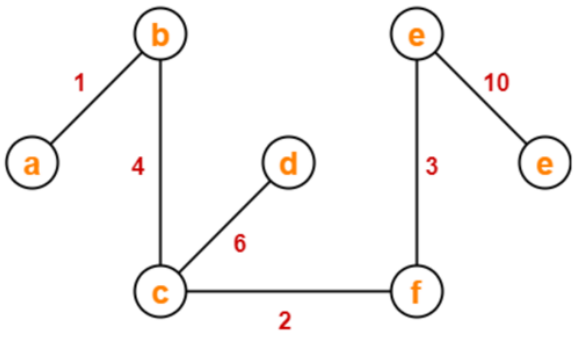
= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

Example 3



Solution-



Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

= 1 + 4 + 2 + 6 + 3 + 10

= 26 units

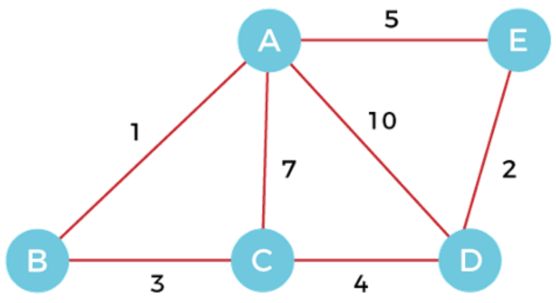
Minimum Spanning Tree – Kruskal's algorithm:

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

The steps to implement Kruskal's algorithm are listed as follows -

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

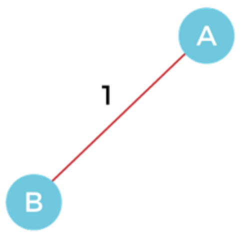
Example 1:



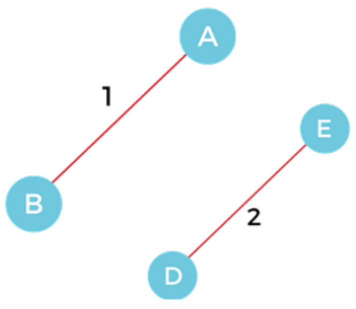
Now write the edges in lower to higher

A ----> B	=1
E ----> D	= 2
B ----> C	= 3
C ----> D	= 4
A ----> E	= 5
A ----> C	= 7
A ----> D	= 10

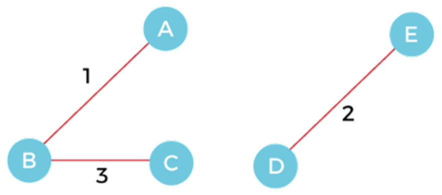
Step 1 - First, add the edge **AB** with weight **1** to the MST.



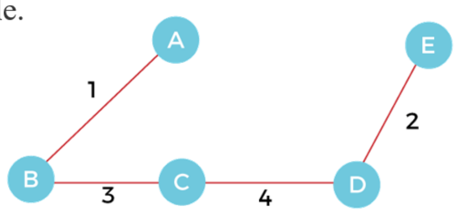
Step 2 - Add the edge DE with weight 2 to the MST as it is not creating the cycle.



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

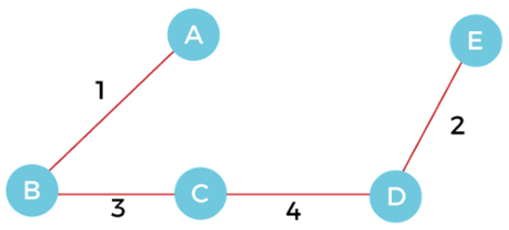


Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.



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.

Applications:

1. Helps to define the flow of computation of software programs.
2. Used in Google maps for building transportation systems. In google maps, the intersection of two or more roads represents the node while the road connecting two nodes represents an edge. Google maps algorithm uses graphs to calculate the shortest distance between two vertices.
3. Used in social networks such as Facebook and Linked-in.
4. Operating system: use Resource Allocation Graph where every process and resource acts as a node while edges are drawn from resources to the allocated process.
5. Used in the world wide web where the web pages represent the nodes.
6. Blockchain: - also use graphs. The nodes are blocks that store many transactions while the edges connect subsequent blocks.
7. Used in modeling data.