MALLA REDDY UNIVERSITY

MR22-1CS0104

ADVANCED DATA STRUCTURES

II YEAR B.TECH. (CSE) / II – SEM

Unit-5

Graphs: The Graph ADT,

Data Structures for Graphs: Edge List, Adjacency List - Adjacency Map - Adjacency Matrix structure,

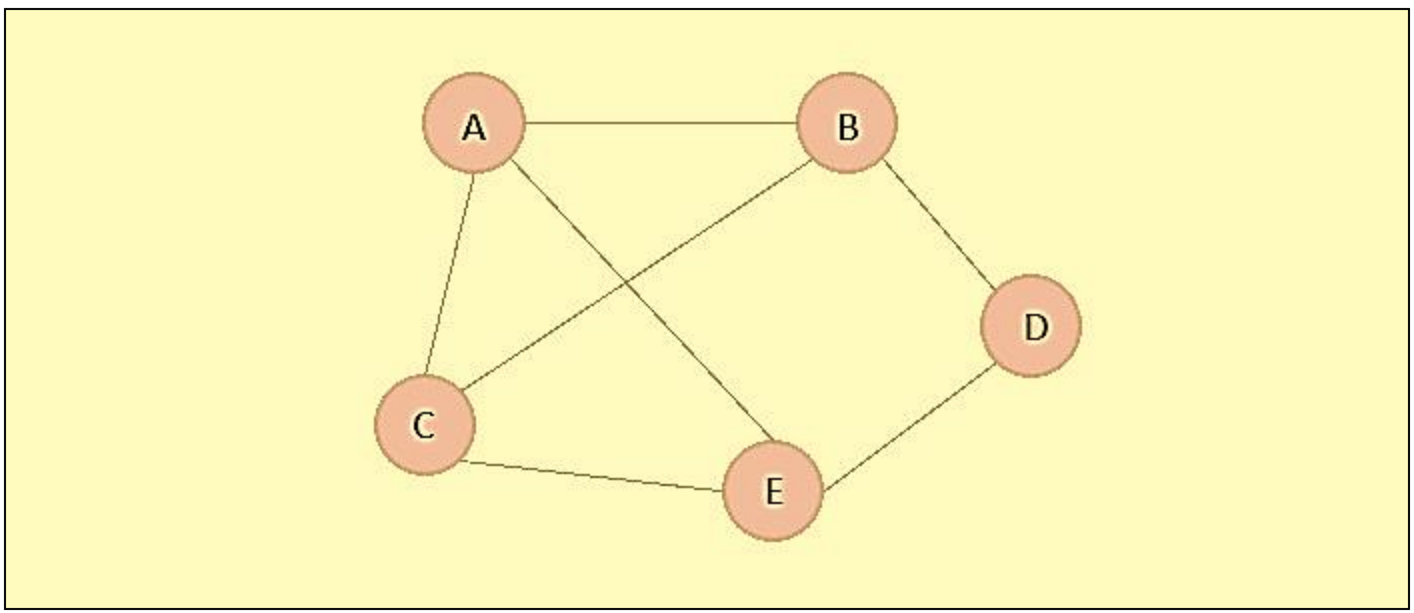
Directed Acyclic graph: Topological Sorting,

Shortest Path Algorithm: All Pairs

Shortest Paths: Floyd-Warshall’s Algorithm

The Graph ADT

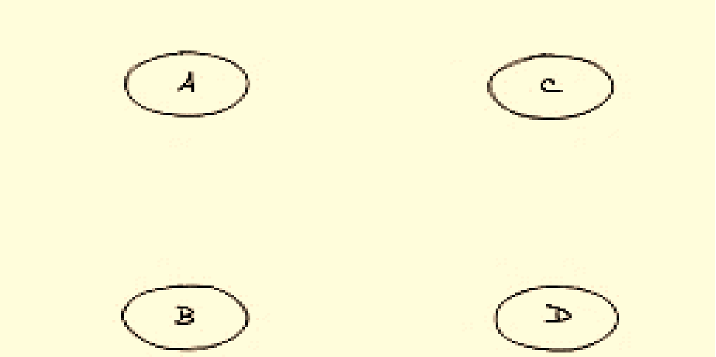
- a non-linear data structure.
- collection of nodes connected by edges.
- The graph is denoted by **G(V, E).**



Types of graphs

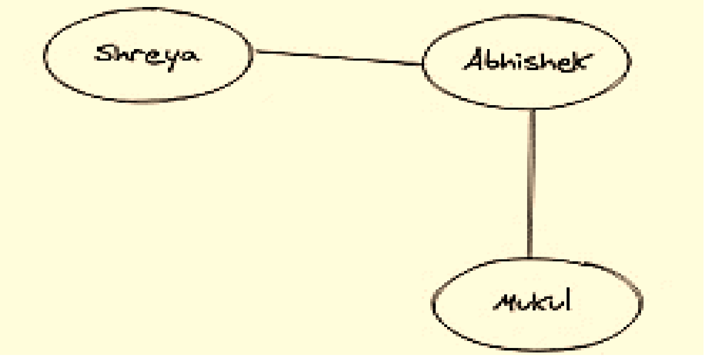
Null Graphs

- no edges in that graph



Undirected Graphs

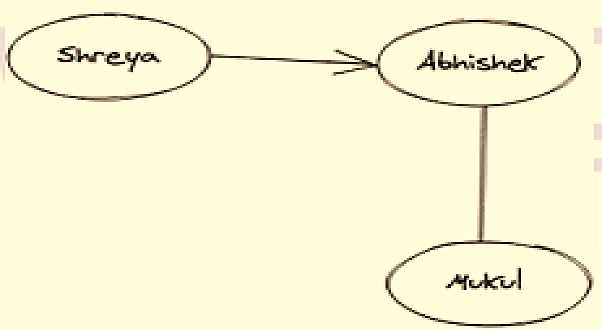
- edge doesn't have any kind of direction associated with it
- the relation is bi-directional



Types of graphs

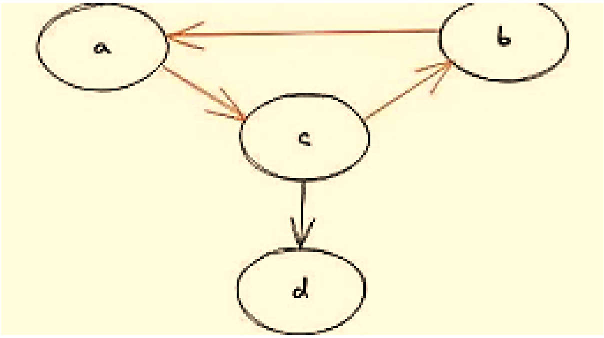
Directed Graphs

- edges with arrows
- relationship is one-way, and it does include a direction



Cyclic Graph

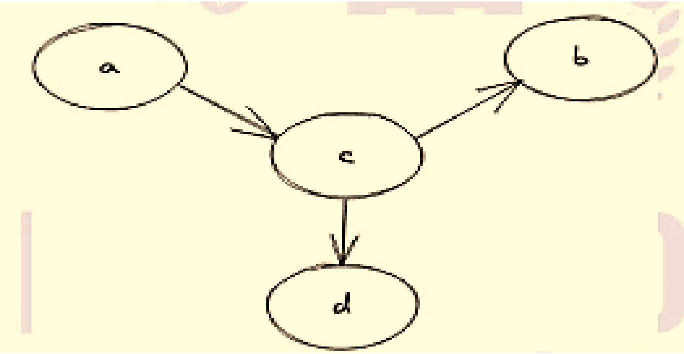
- least one node that traverses back to itself.
- the relation is bi-directional



Types of graphs

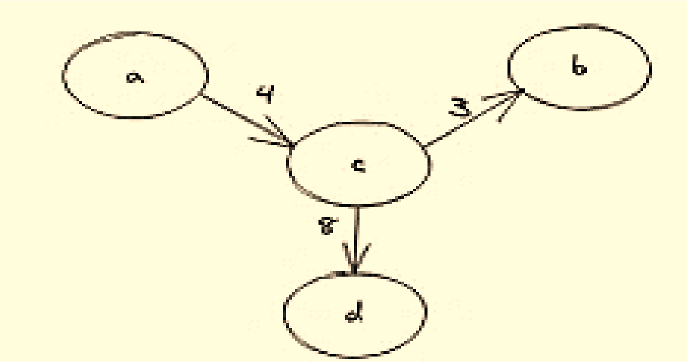
Acyclic Graph

- doesn't have a single cycle



Weighted Graph

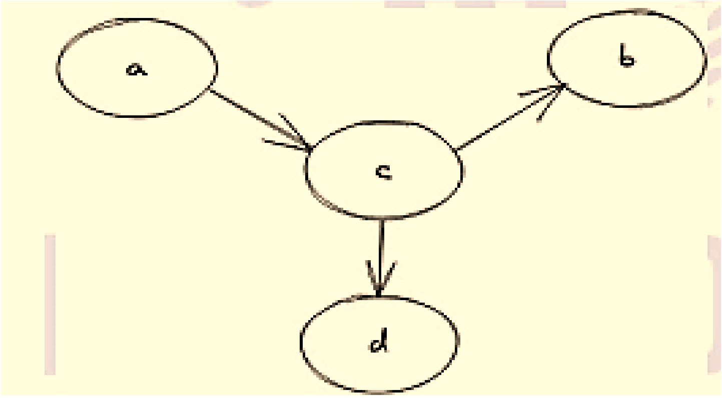
- edges in a graph has some weight associated with it



Types of graphs

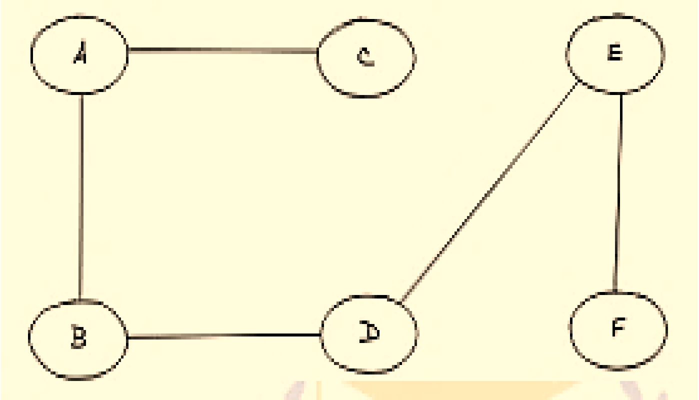
UnWeighted Graph

- doesn't have any weight associated with it



Connected Graph

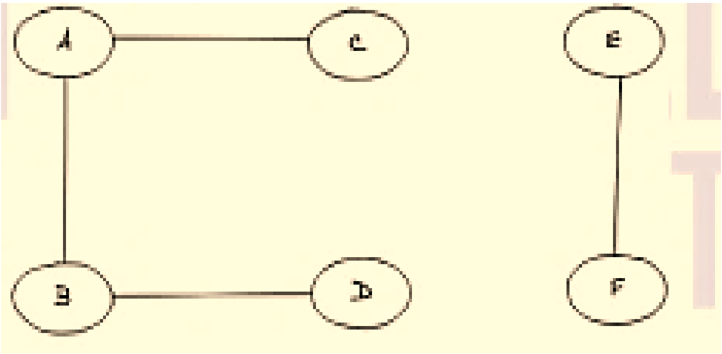
- we have a path between every two nodes of the graph



Types of graphs

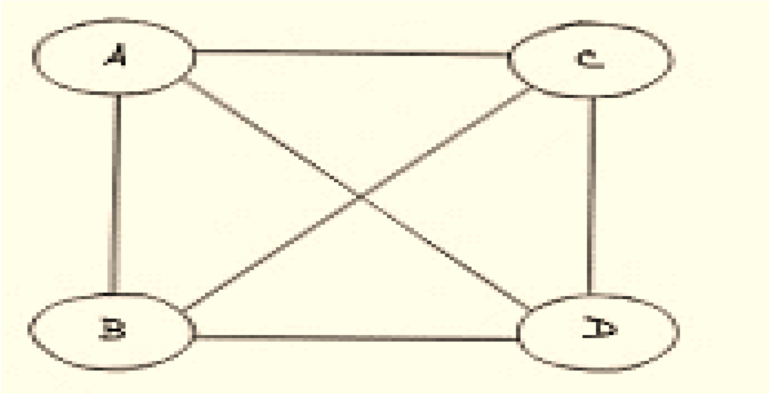
Disconnected Graph

- not connected
- not be able to find a path between every two nodes of the graph



Complete Graph

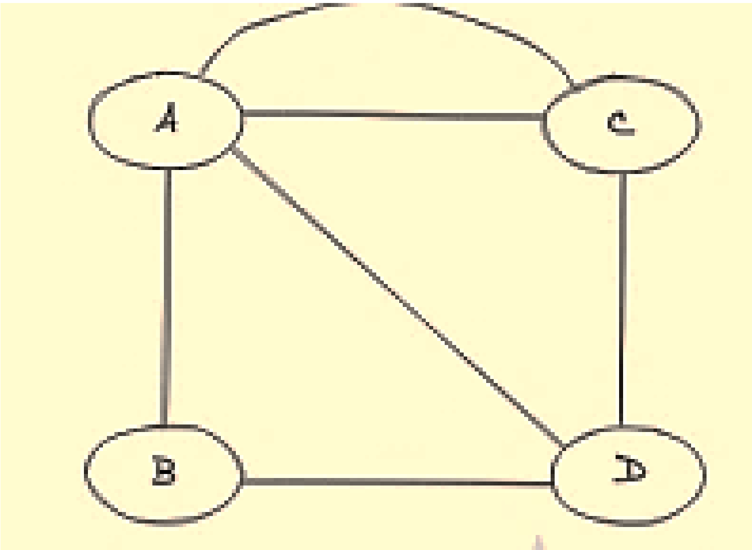
- if there exists an edge for every pair of vertices(nodes)



Types of graphs

Multigraph

- if there exist two or more than two edges between any pair of nodes

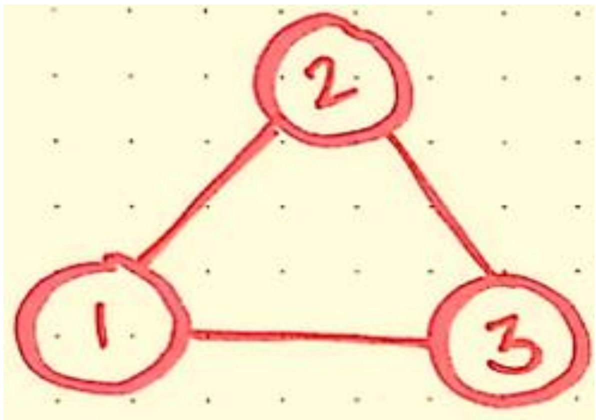


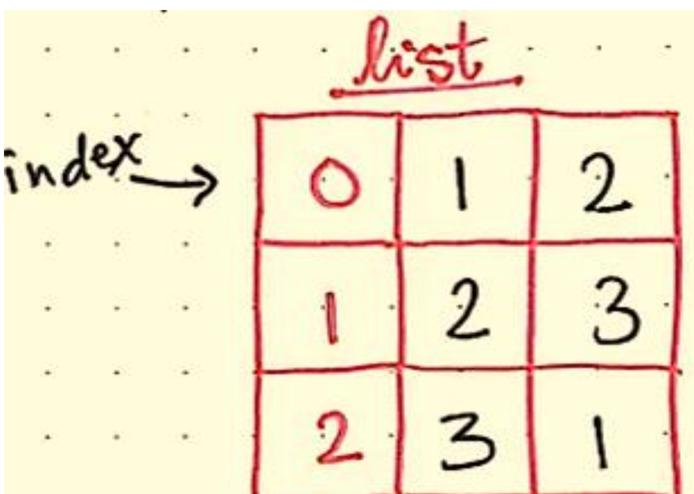
Representations of Graph

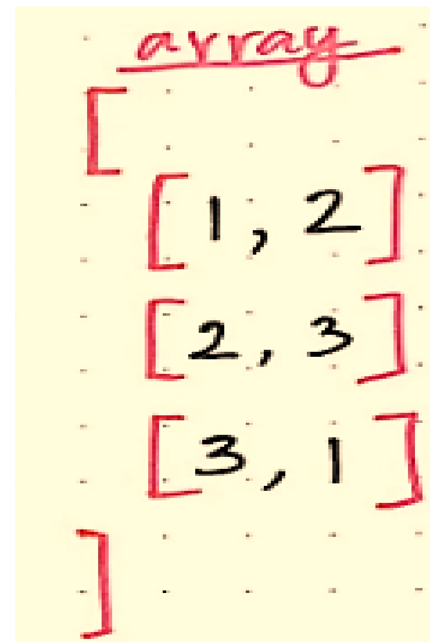
- The common data structures for graph representation are,
    - Edge List
    - Adjacency List
    - Adjacency Map
    - Adjacency Matrix structure

Edge List representation

- A list of all edges in a graph.
- Uses a list or array.

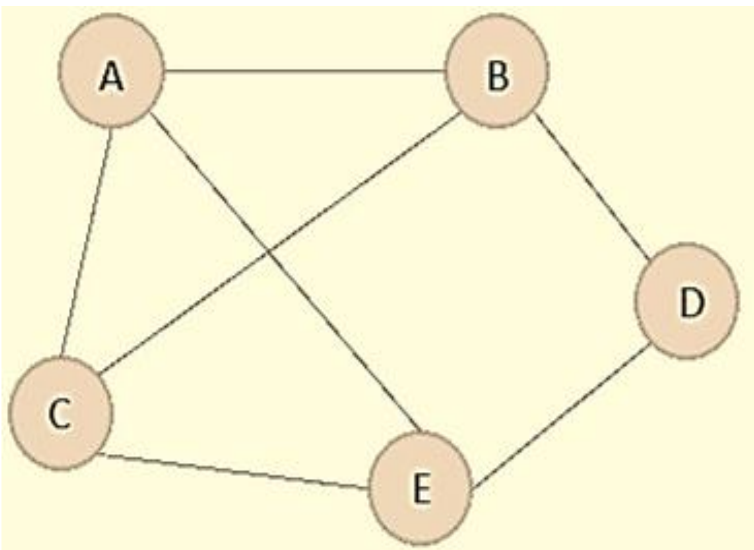


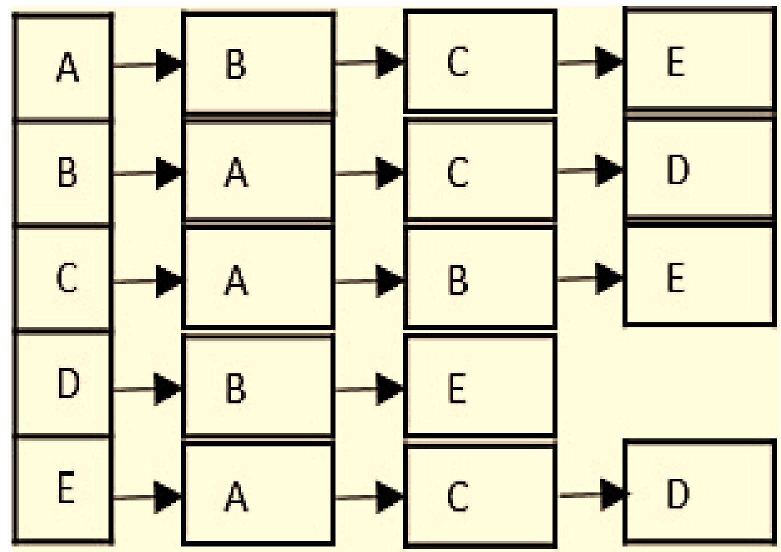




Adjacency List representation

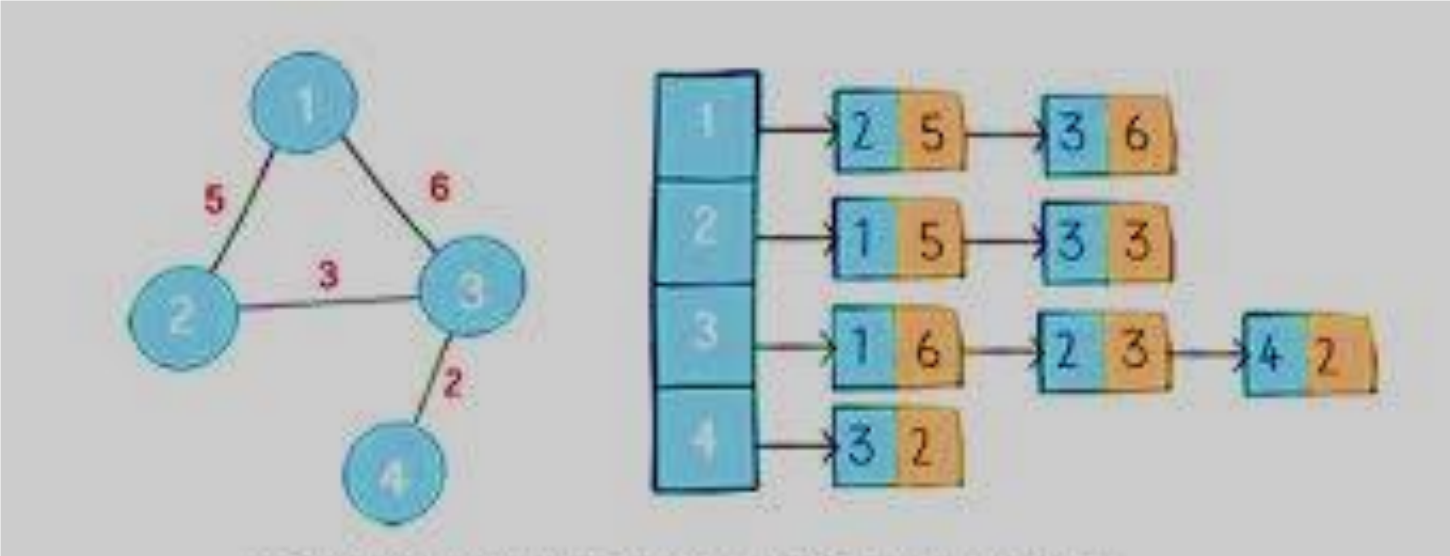
- Uses List.
- The adjacency list for the given graph is,





Adjacency List representation

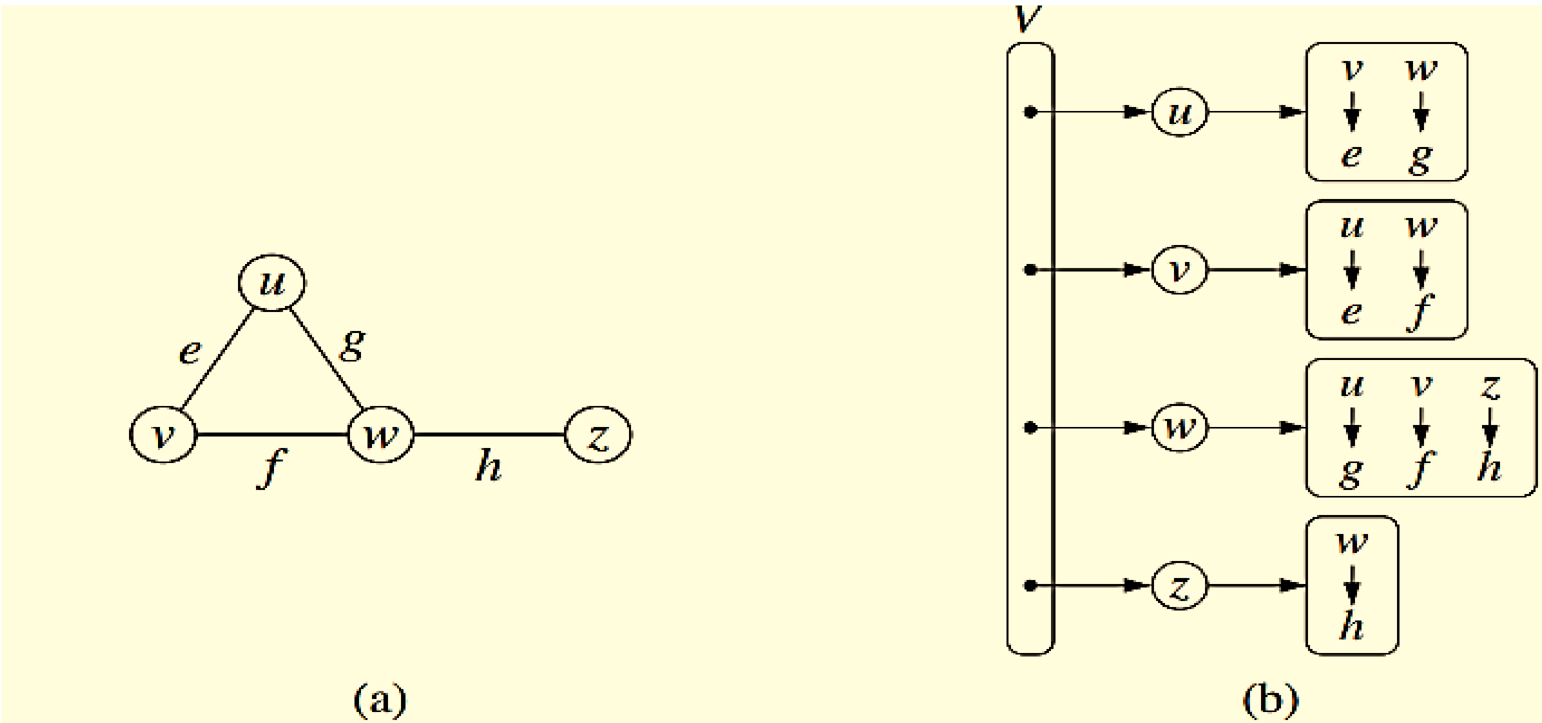
- Uses List.
- The adjacency list for the given graph is,



weighted undirected graph and its adjacency list

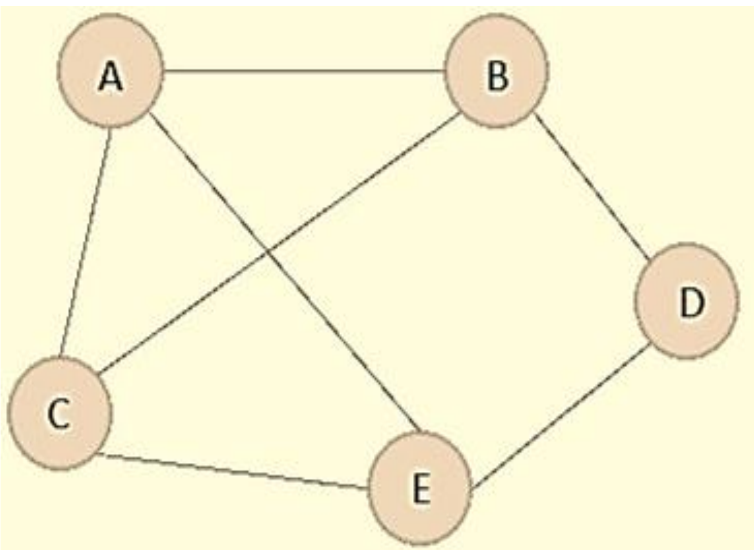
Adjacency Map representation

- Improves the performance by using a hash-based map .
- The advantage of the adjacency map, relative to an adjacency list, is that the getEdge(u, v) method can be implemented in expected O(1) time.



Adjacency Matrix representation

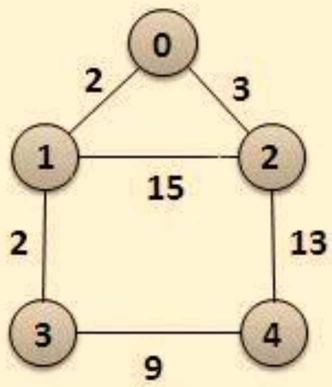
- a V x V matrix where the values are filled with either 0 or 1.
- if the link exists between Vi and Vj, it is recorded 1; otherwise, 0.



$$
\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}
\begin{array}{c}
\begin{array}{ccccc} A & B & C & D & E \end{array} \\
\left[\begin{array}{ccccc}
0 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0
\end{array}\right]
\end{array}
$$

Adjacency Matrix representation

- a V x V matrix where the values are filled with either 0 or 1.
- if the link exists between Vi and Vj, it is recorded 1; otherwise, 0.



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Adjacency Matrix Representation of
Weighted Graph

Graph traversal

- Graph traversal is a technique used for a searching vertex in a graph.
- Also decides the order of vertices is visited in the search process.
- we visit all the vertices of the graph without getting into looping path.
- There are two graph traversal techniques
    - DFS (Depth First Search)
    - BFS (Breadth First Search)

DFS (Depth First Search)

- produces a spanning tree as final result.
- Spanning Tree is a graph without loops.
- use Stack data structure with maximum size of total number of vertices in the graph.
- 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.

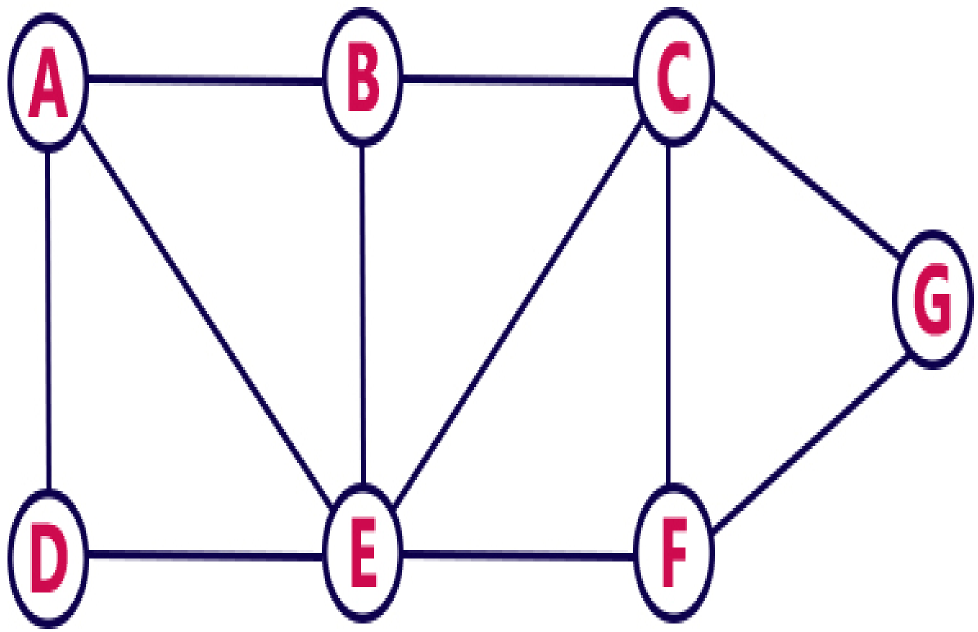
DFS (Depth First Search)

- 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.

DFS (Depth First Search)

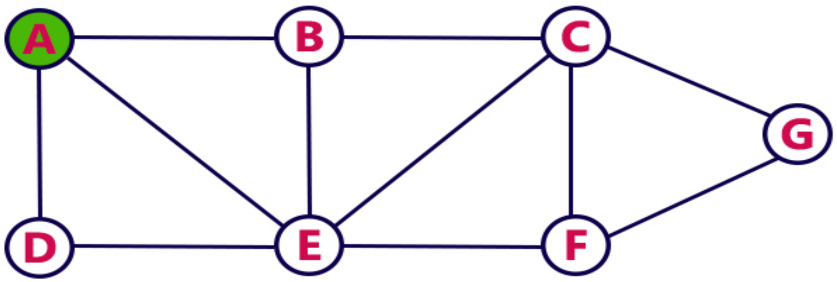
Consider the following example graph to perform DFS traversal

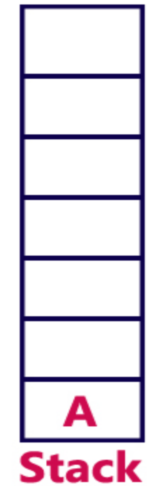


DFS (Depth First Search)

Step 1:

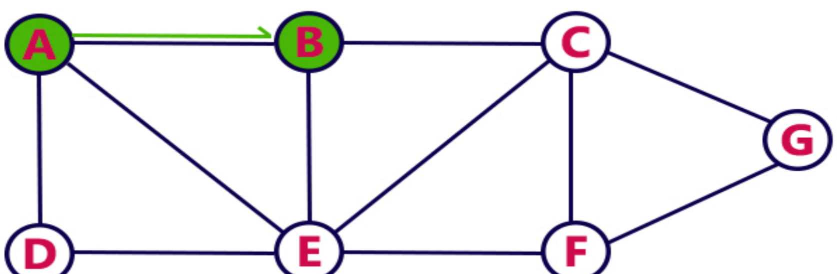
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

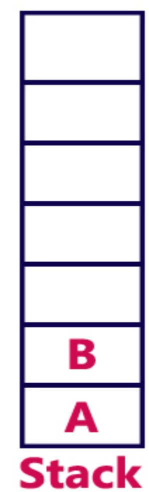




Step 2:

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

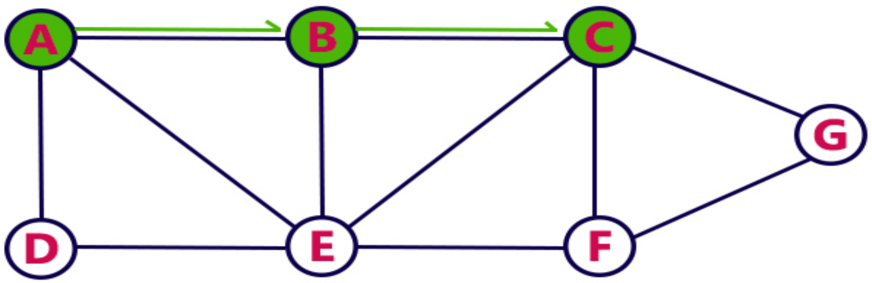


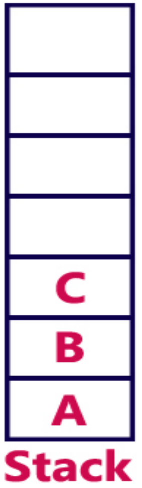


DFS (Depth First Search)

Step 3:

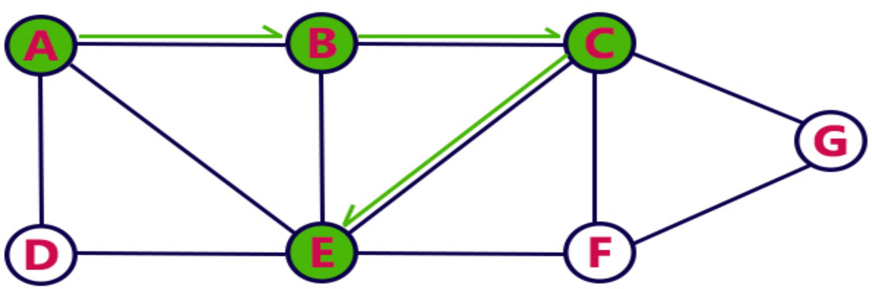
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

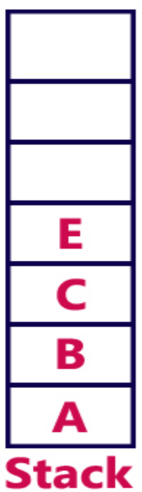




Step 4:

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

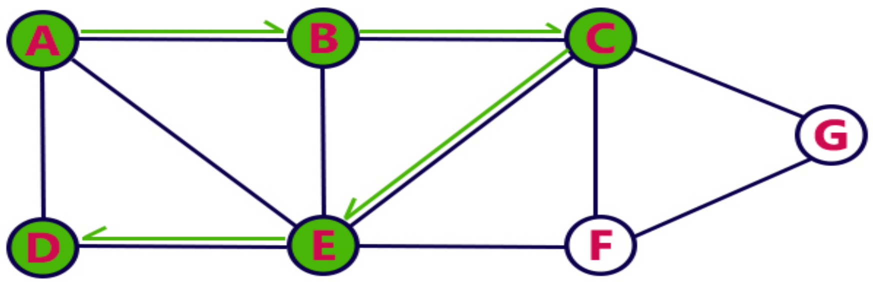


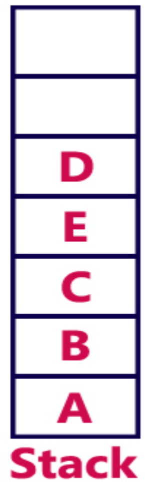


DFS (Depth First Search)

Step 5:

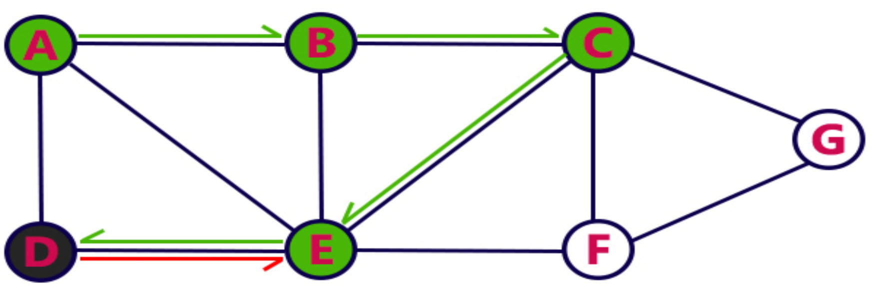
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

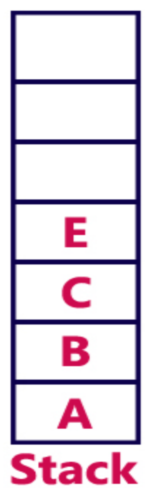




Step 6:

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

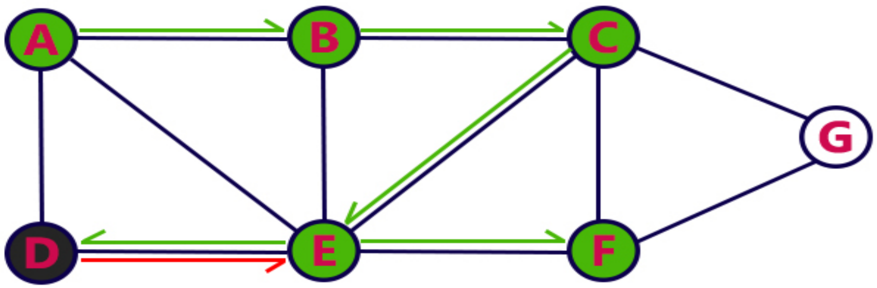


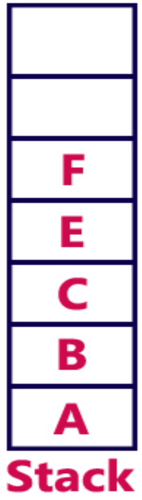


DFS (Depth First Search)

Step 7:

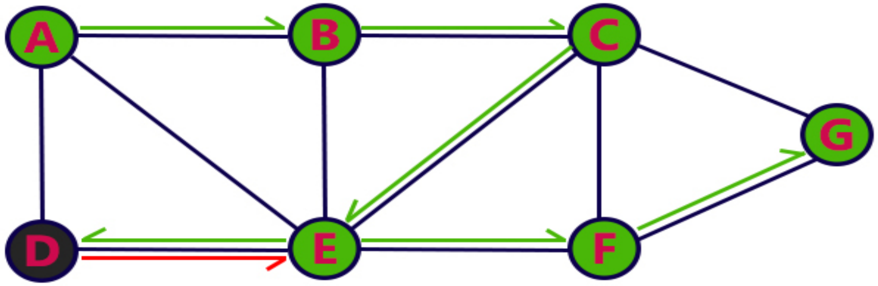
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

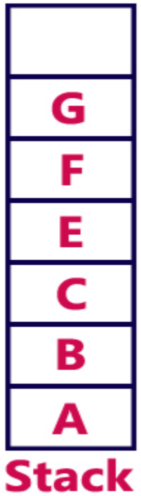




Step 8:

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

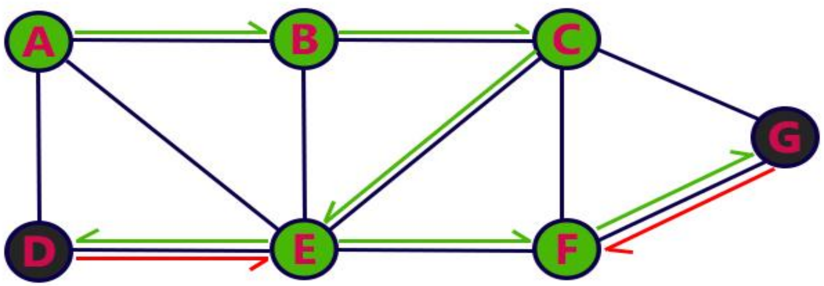


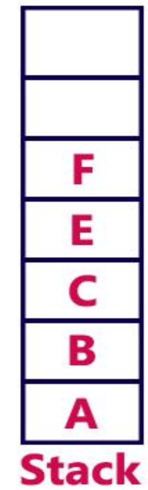


DFS (Depth First Search)

Step 9:

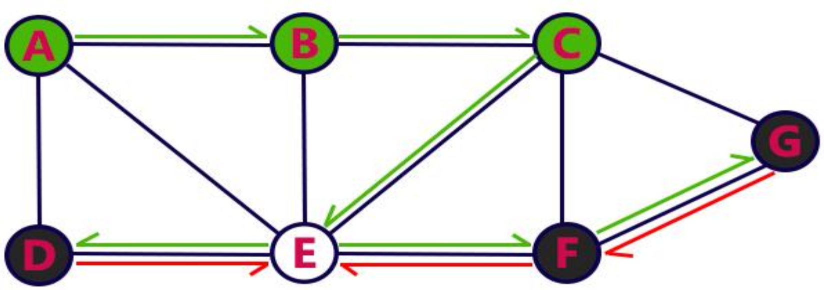
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

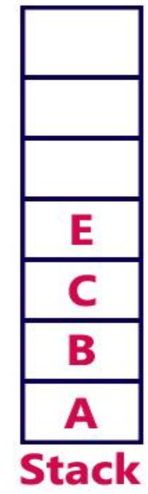




Step 10:

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

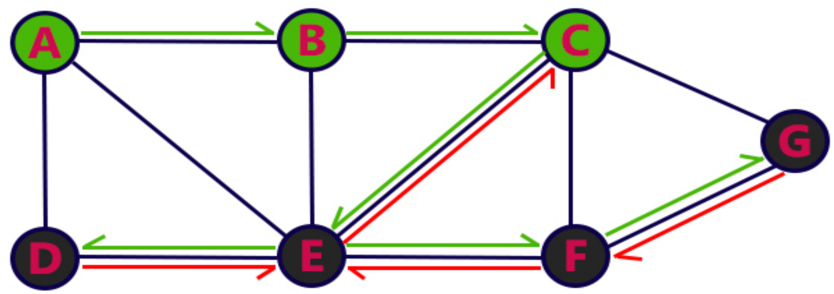


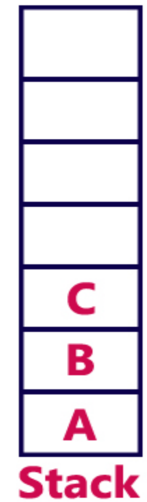


DFS (Depth First Search)

Step 11:

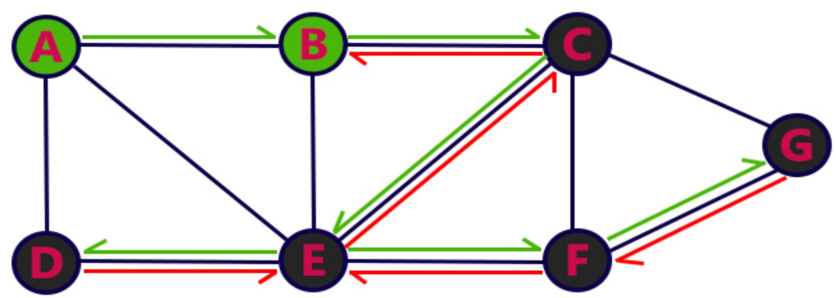
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

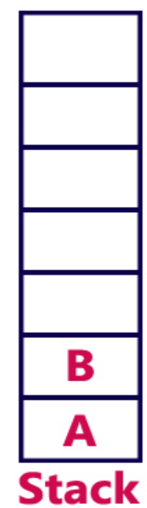




Step 12:

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

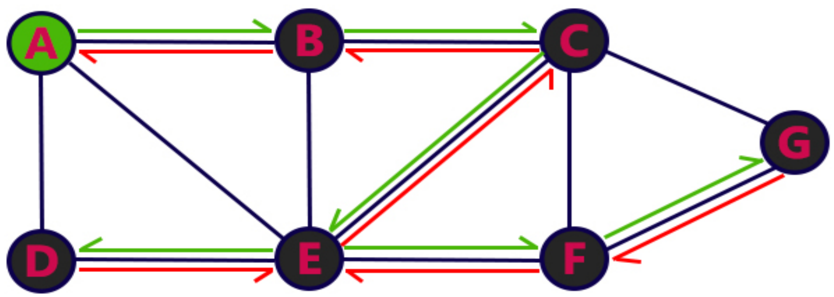


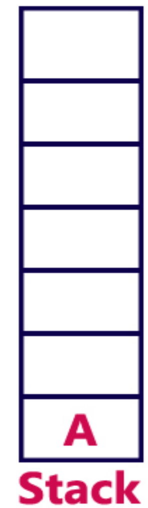


DFS (Depth First Search)

Step 13:

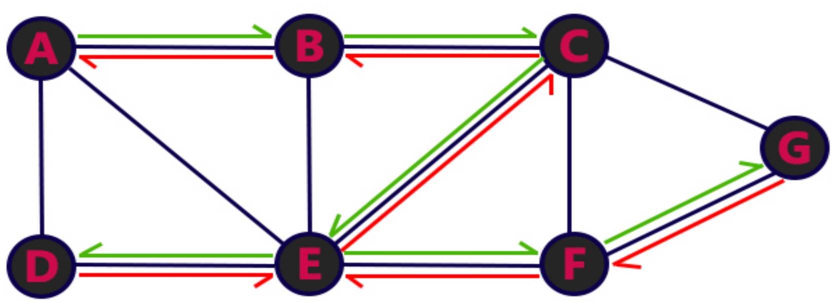
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

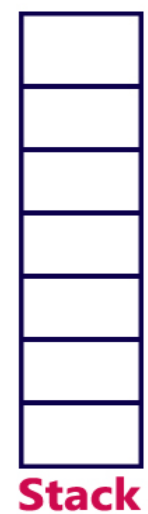




Step 14:

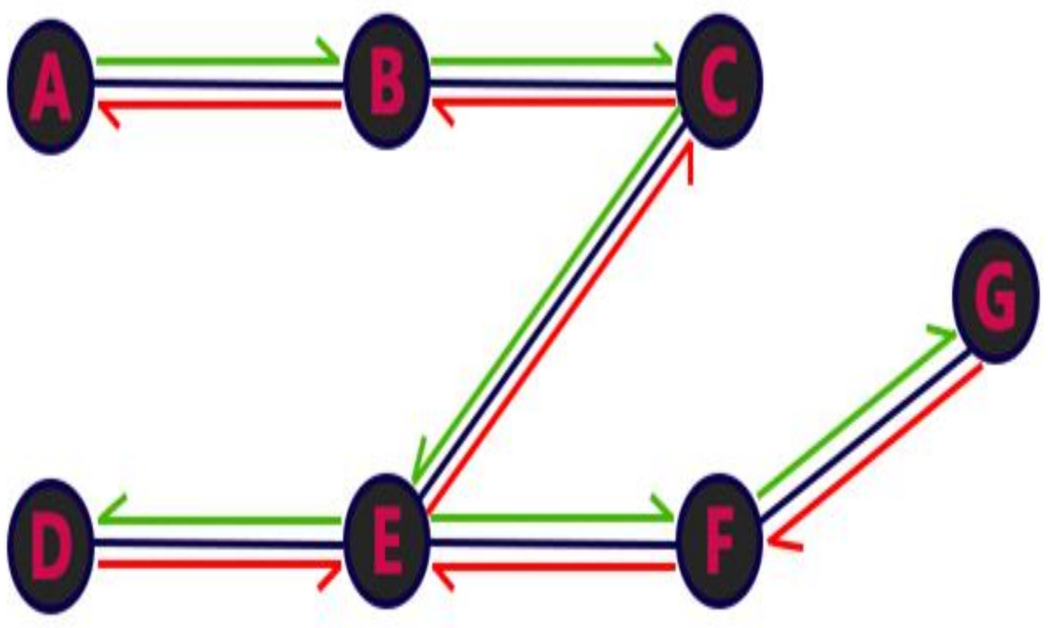
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





DFS (Depth First Search)

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



DFS (Depth First Search)

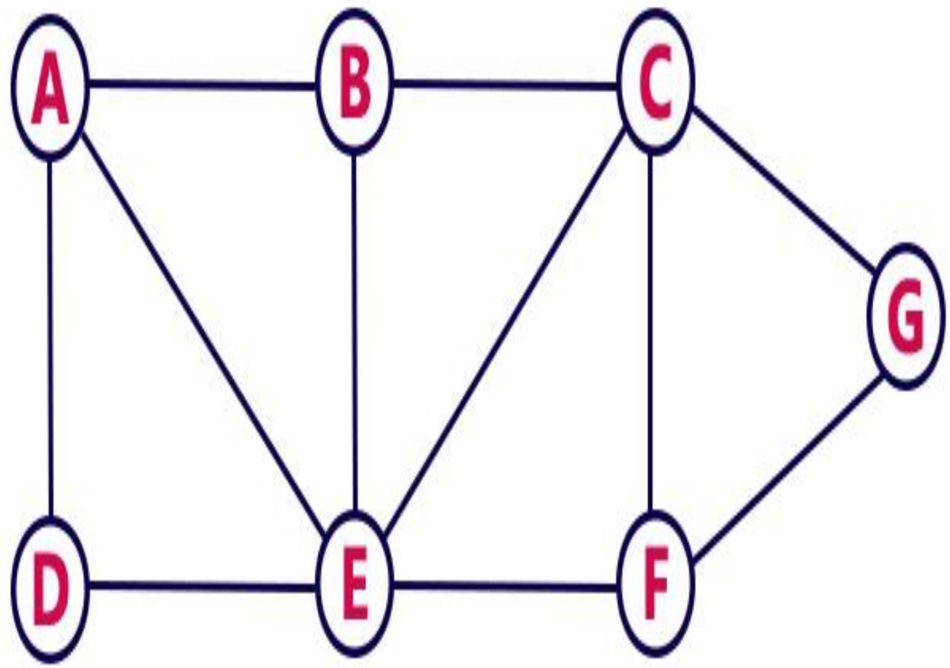
```
void DFSUtil(int v, boolean visited[])
    {
      //Mark the current node as visited and print it
          visited[v] = true;
         System.out.print(v + " ");
       // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext()) {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);        }    }

// The function to do DFS traversal uses recursive DFSUtil()
    void DFS(int v)
    {
        // Mark all the vertices as not visited(set as false by default in
   java)
        boolean visited[] = new boolean[V];
         // Call the recursive helper function to print DFS traversal
        DFSUtil(v, visited);    }
```

BFS (Breadth First Search)

- produces a spanning tree as final result.
- use Queue data structure with maximum size of total number of vertices in the graph
- 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

BFS (Breadth First Search)

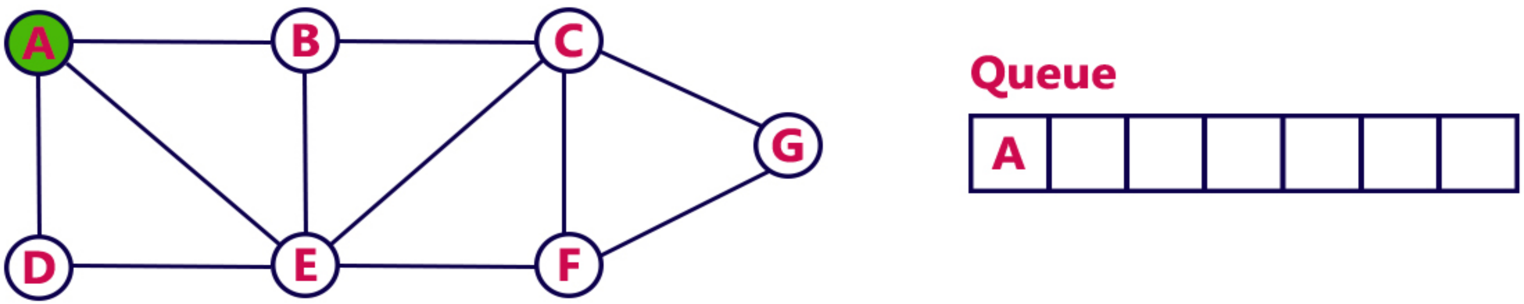
Consider the following example graph to perform BFS traversal



BFS (Breadth First Search)

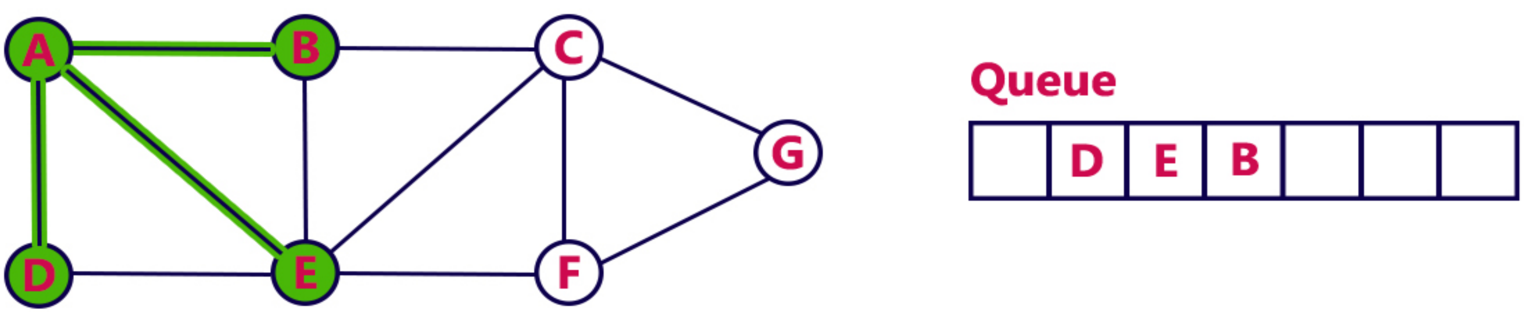
Step 1:

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



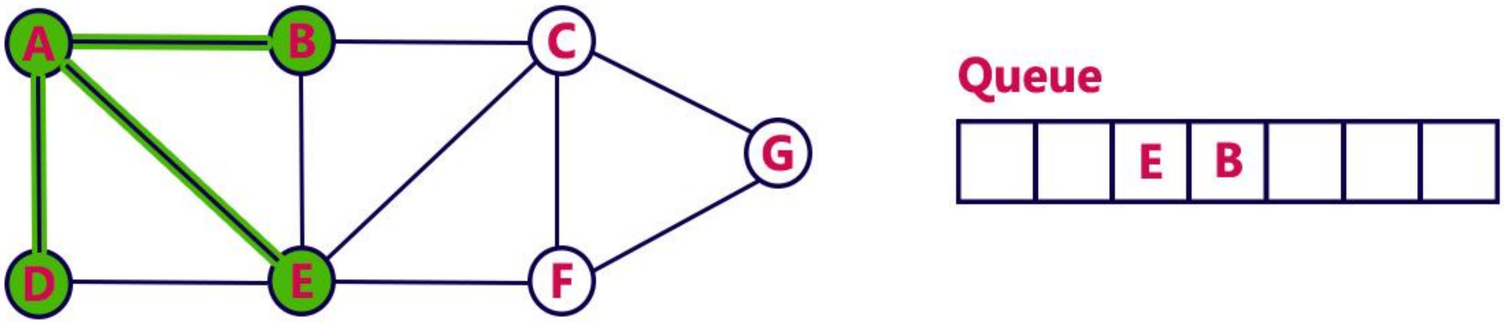
Step 2:

- 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..



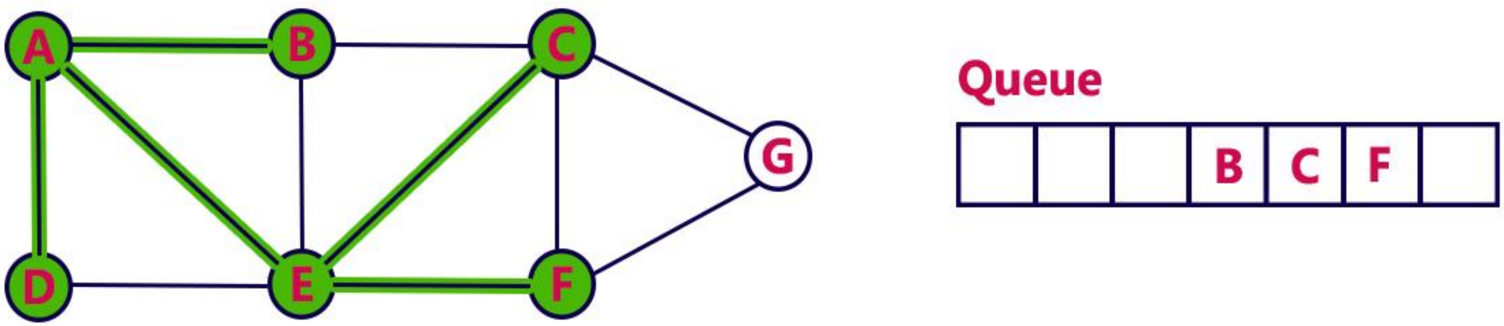
Step 3:

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



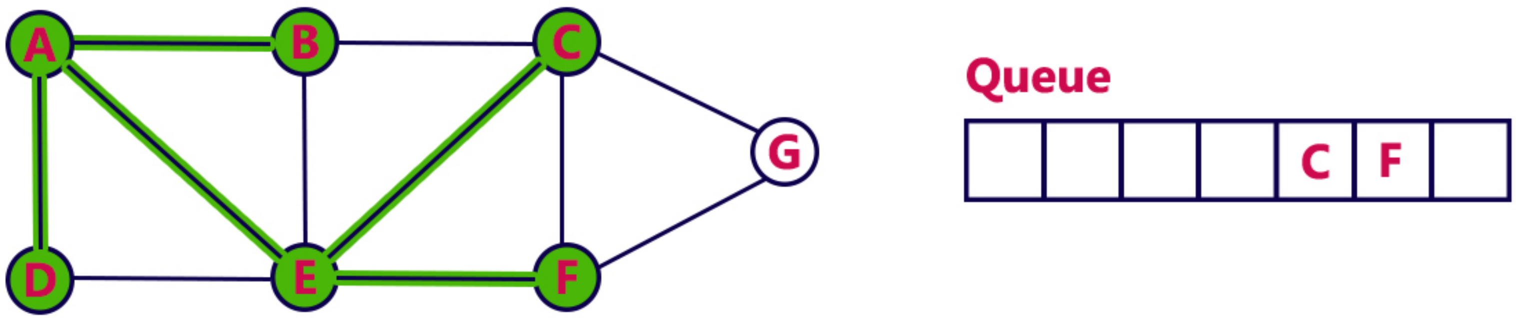
Step 4:

- 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.



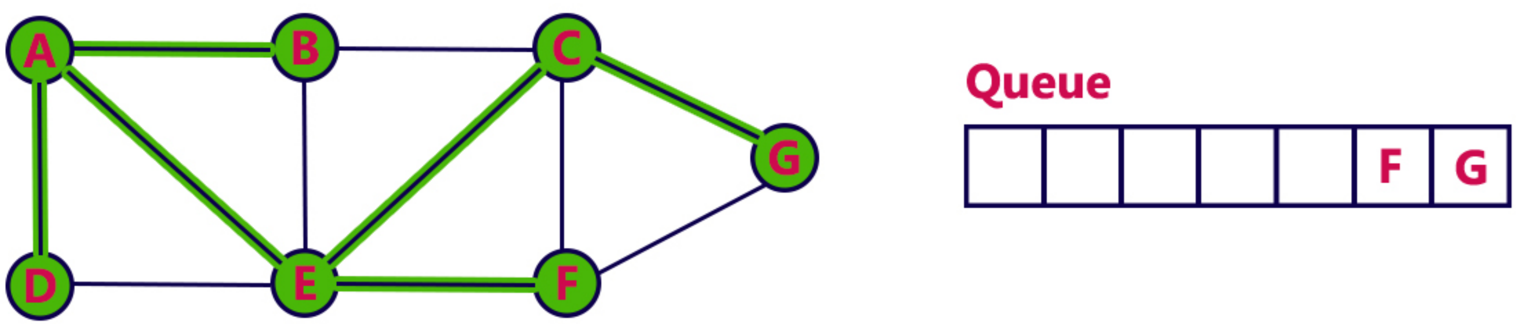
Step 5:

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



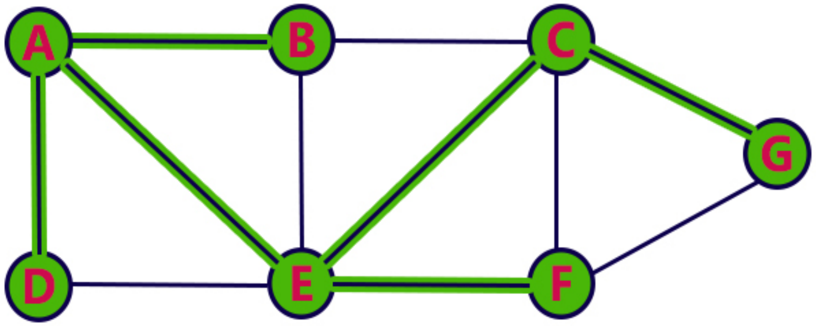
Step 6:

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



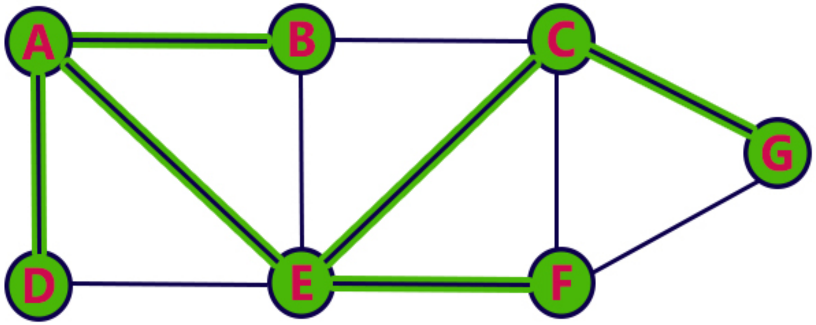
Step 7:

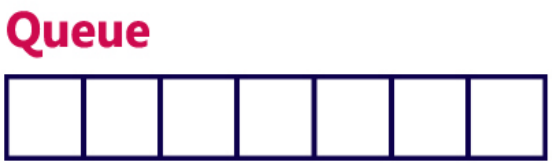
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



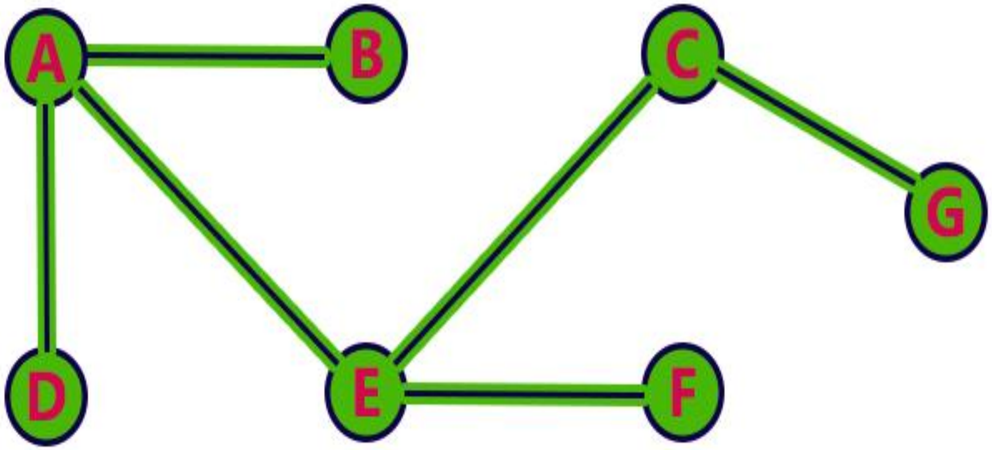
Step 8:

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



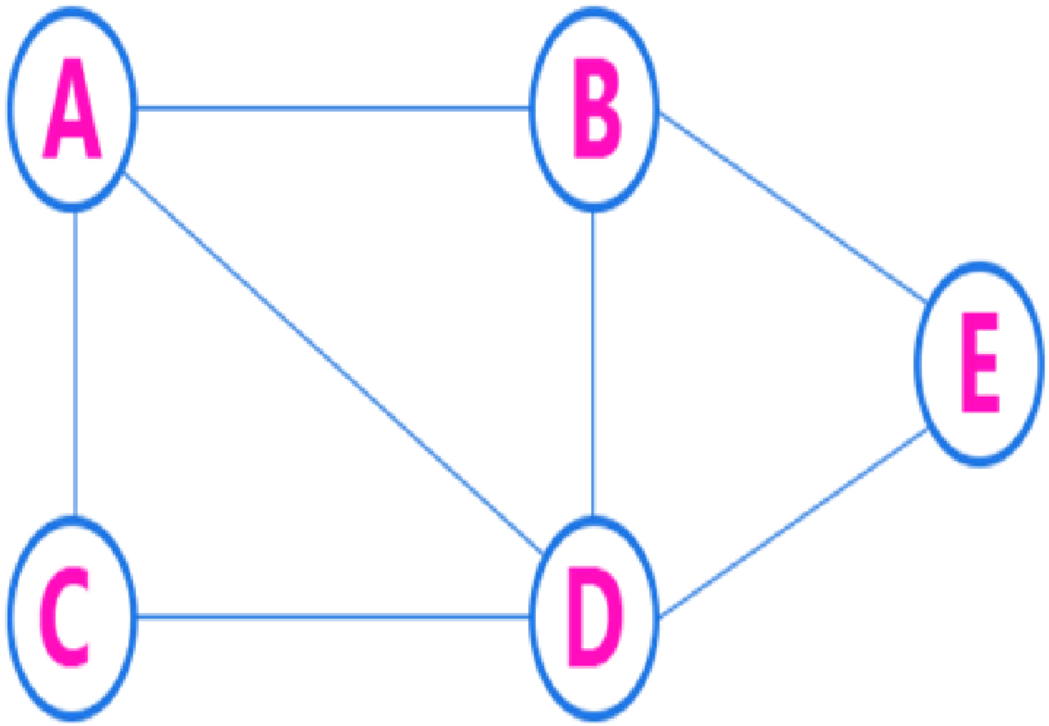


- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...

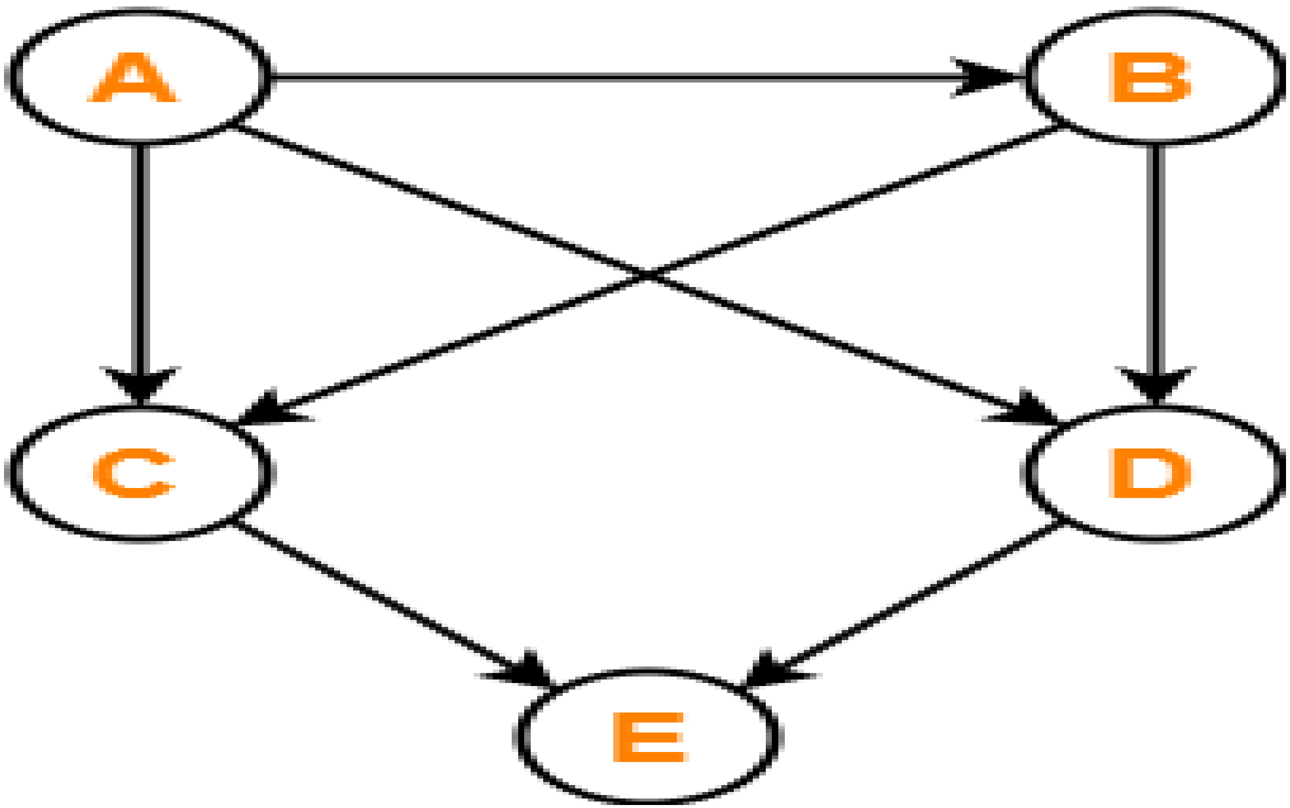


Example1:

Traverse a graph using Breadth First Search



Directed acyclic graph



□ Topological Sorting is the best example for Directed Acyclic Graph.

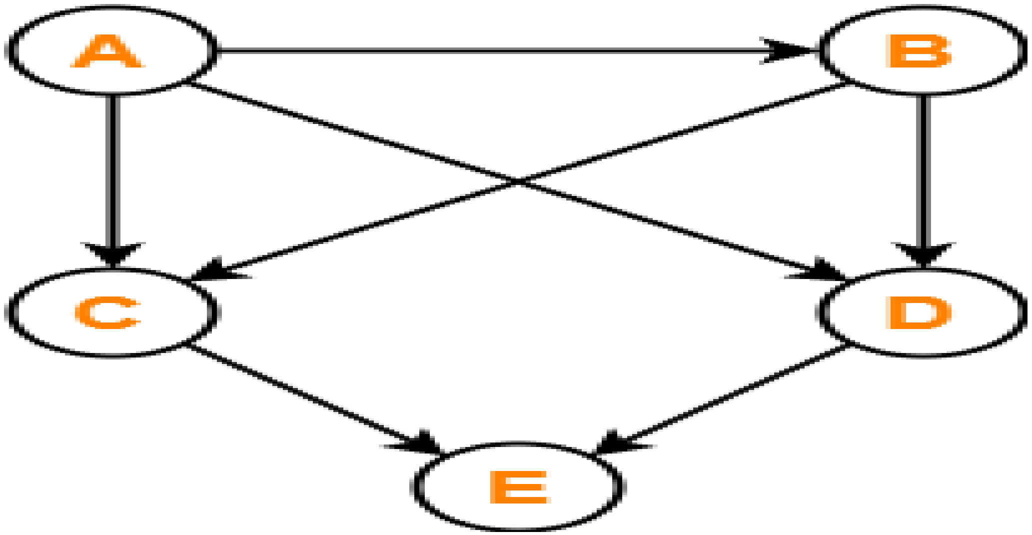
TOPOLOGICAL SORTING

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

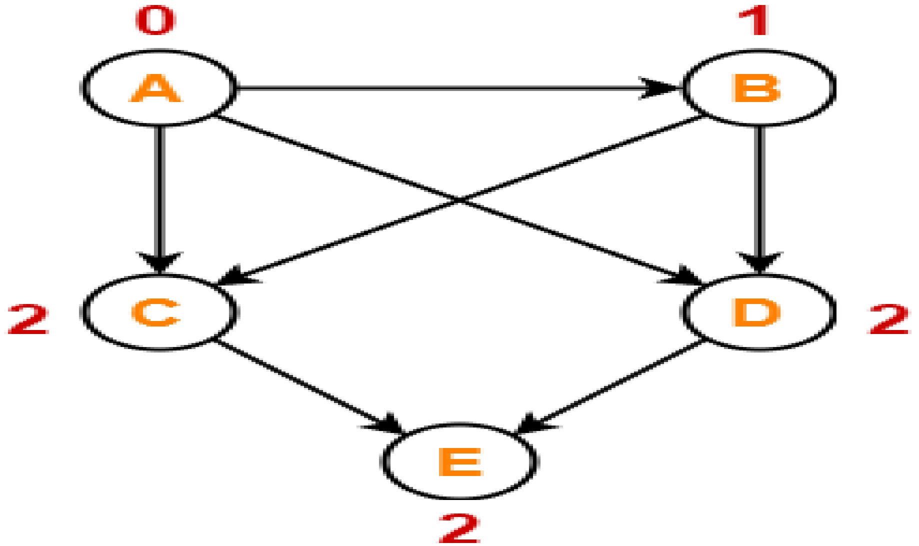
Applications of Topological Sort:

- Scheduling jobs
- Instruction Scheduling
- Determining the order of compilation tasks

TOPOLOGICAL SORTING

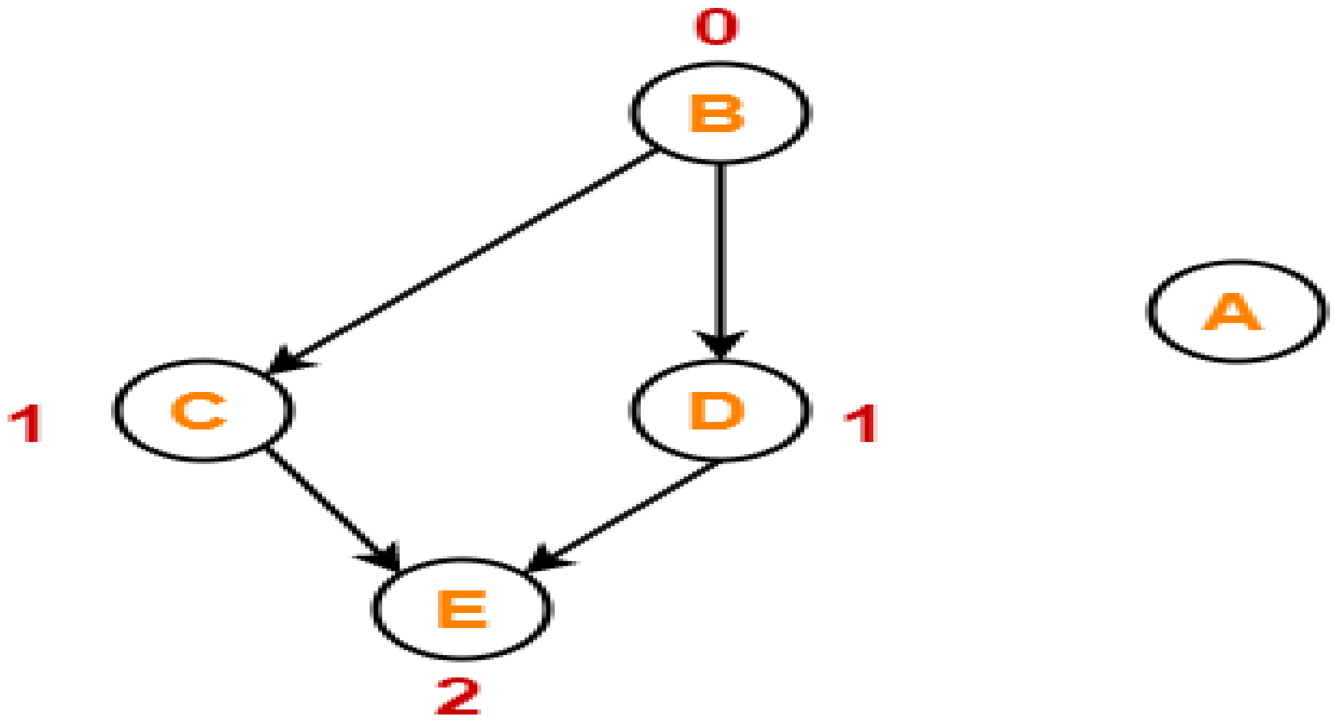


Write in-degree of each vertex-



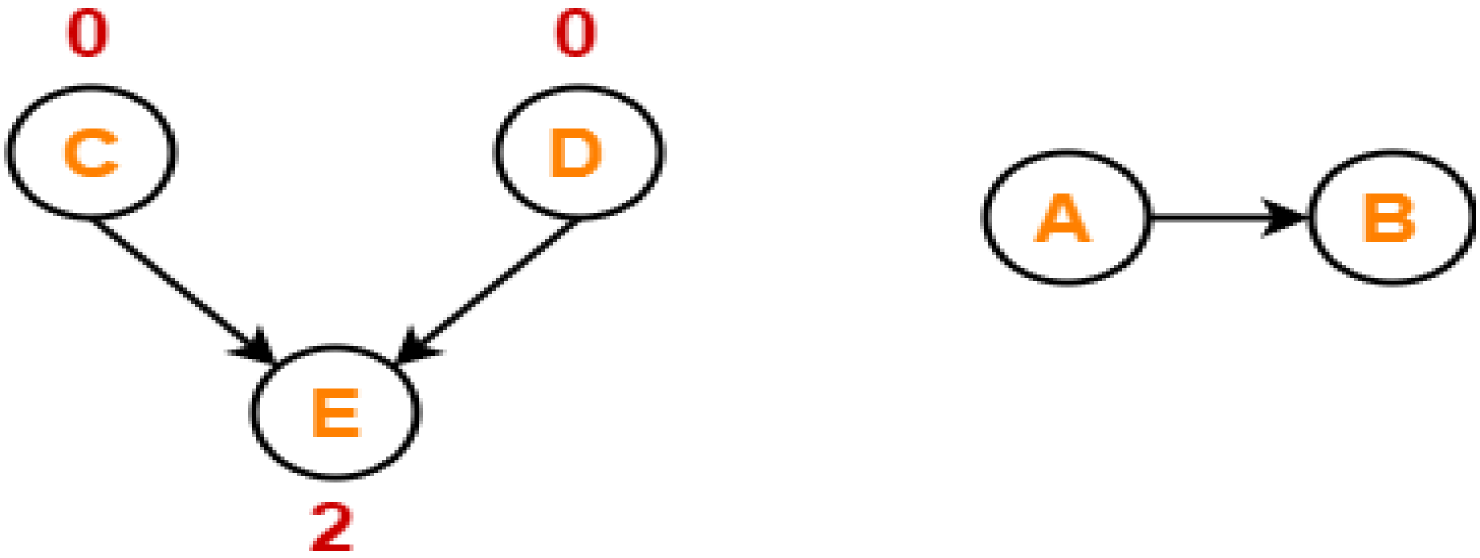
TOPOLOGICAL SORTING

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



TOPOLOGICAL SORTING

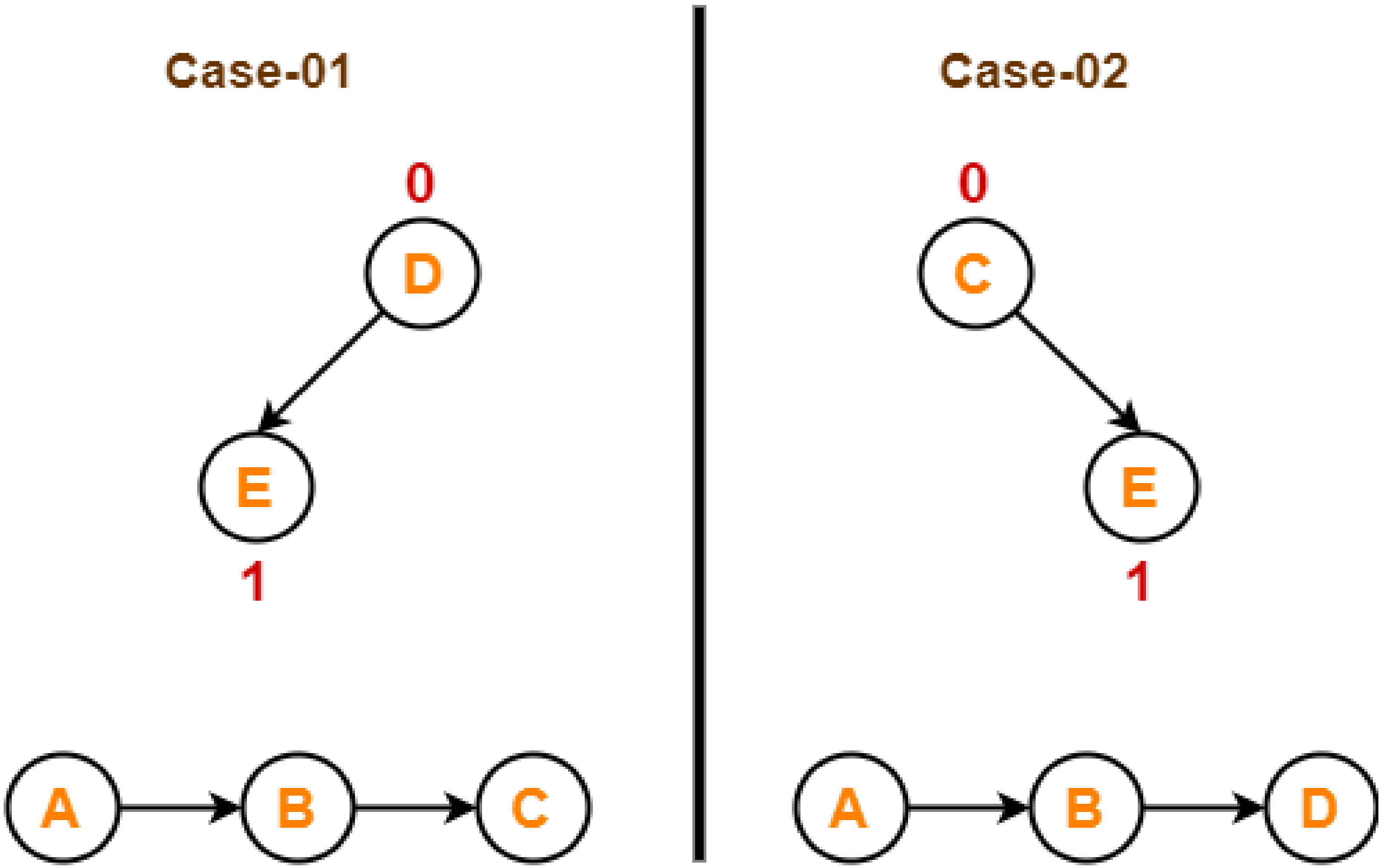
- Vertex-B has the least in-degree.
- So, remove vertex-B and its associated edges.
- Now, update the in-degree of other vertices.



TOPOLOGICAL SORTING

- There are two vertices with the least in-degree. So, following 2 cases are possible-
- In case-01,
  - Remove vertex-C and its associated edges.
  - Then, update the in-degree of other vertices.
- In case-02,
  - Remove vertex-D and its associated edges.
  - Then, update the in-degree of other vertices.

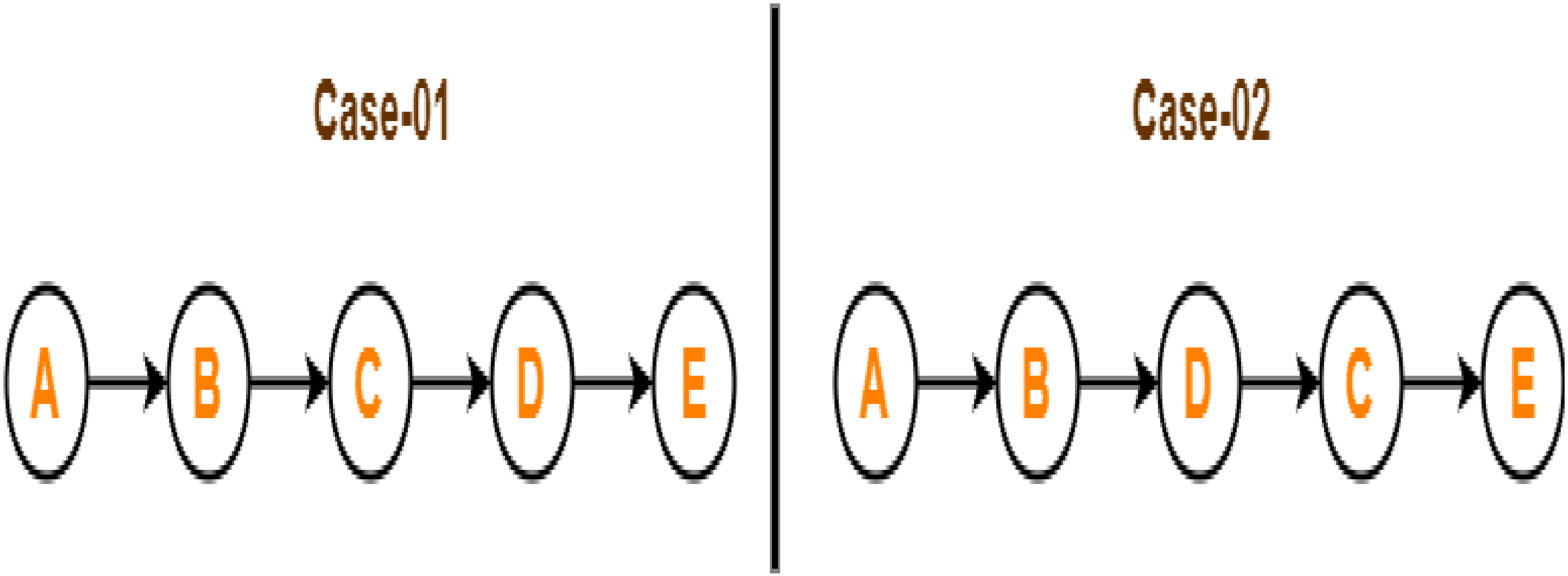
TOPOLOGICAL SORTING



TOPOLOGICAL SORTING

- Now, the above two cases are continued separately in the similar manner.
- In case-01,
  - Remove vertex-D since it has the least in-degree.
  - Then, remove the remaining vertex-E.
- In case-02,
  - Remove vertex-C since it has the least in-degree.
  - Then, remove the remaining vertex-E.

TOPOLOGICAL SORTING



Dijsktra's Algorithm

- used for solving the single source shortest path problem.
- computes the shortest path from one particular source node to all other remaining nodes of the graph.
- works only for connected graphs and for those graphs that do not contain any negative weight edge.
- only provides the value or cost of the shortest paths.
- works for directed as well as undirected graphs.

Step-01

- In the first step, two sets are defined:
  - One set contains all those vertices which have been included in the shortest path tree.
  - In the beginning, this set is empty.
  - Other set contains all those vertices which are still left to be included in the shortest path tree.
- In the beginning, this set contains all the vertices of the given graph.

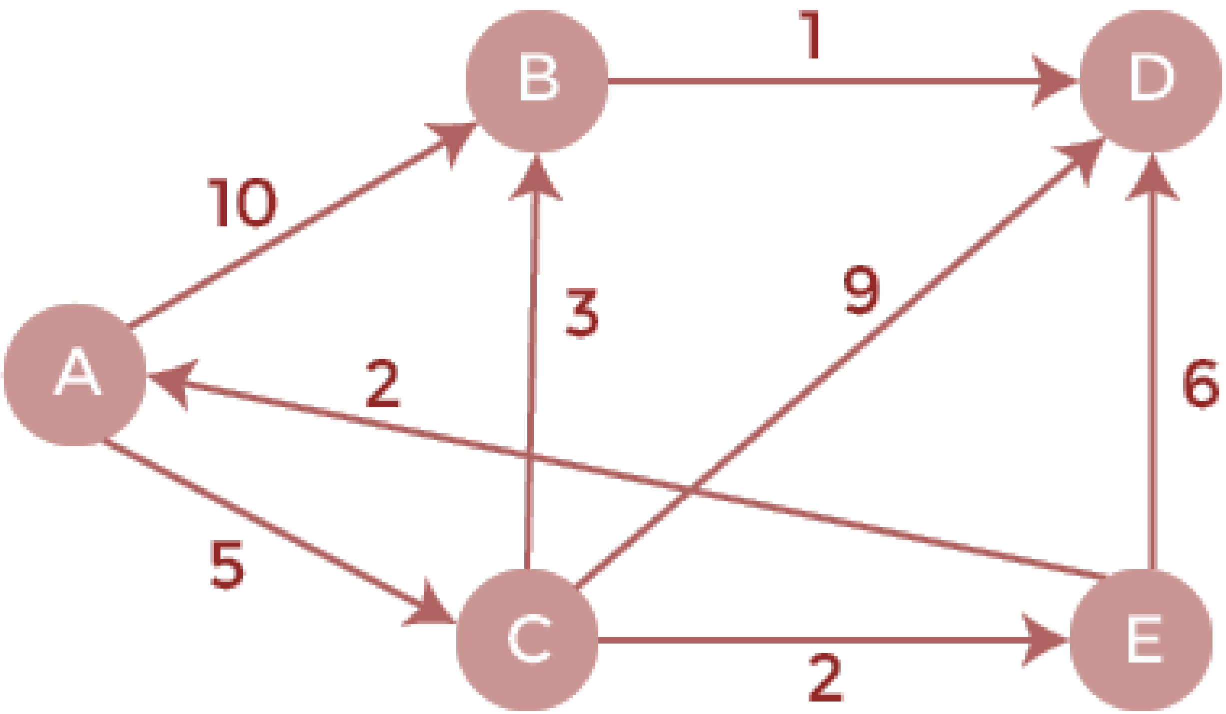
Step-02

- For each vertex of the given graph, two variables are defined as-
- Π[v] which denotes the predecessor of vertex ‘v’
- d[v] which denotes the shortest path estimate of vertex ‘v’ from the source vertex.
- Initially, the value of these variables is set as-
- The value of variable ‘Π’ for each vertex is set to NIL i.e. Π[v] = NIL
- The value of variable ‘d’ for source vertex is set to 0 i.e. d[S] = 0
- The value of variable ‘d’ for remaining vertices is set to ∞ i.e. d[v] = ∞

Step-03

- The following procedure is repeated until all the vertices of the graph are processed-
- Among unprocessed vertices, a vertex with minimum value of variable ‘d’ is chosen.
- Its outgoing edges are relaxed.
- After relaxing the edges for that vertex, the sets created in step-01 are updated.

Example 1



A	B	C	D	E
∞	∞	∞	∞	∞

	A	B	C	D	E
A	0	∞	∞	∞	∞

	A	B	C	D	E
A	0	∞	∞	∞	∞
		10	5	∞	∞

	A	B	C	D	E
A	0	∞	∞	∞	∞
		10	5	∞	∞
		8			

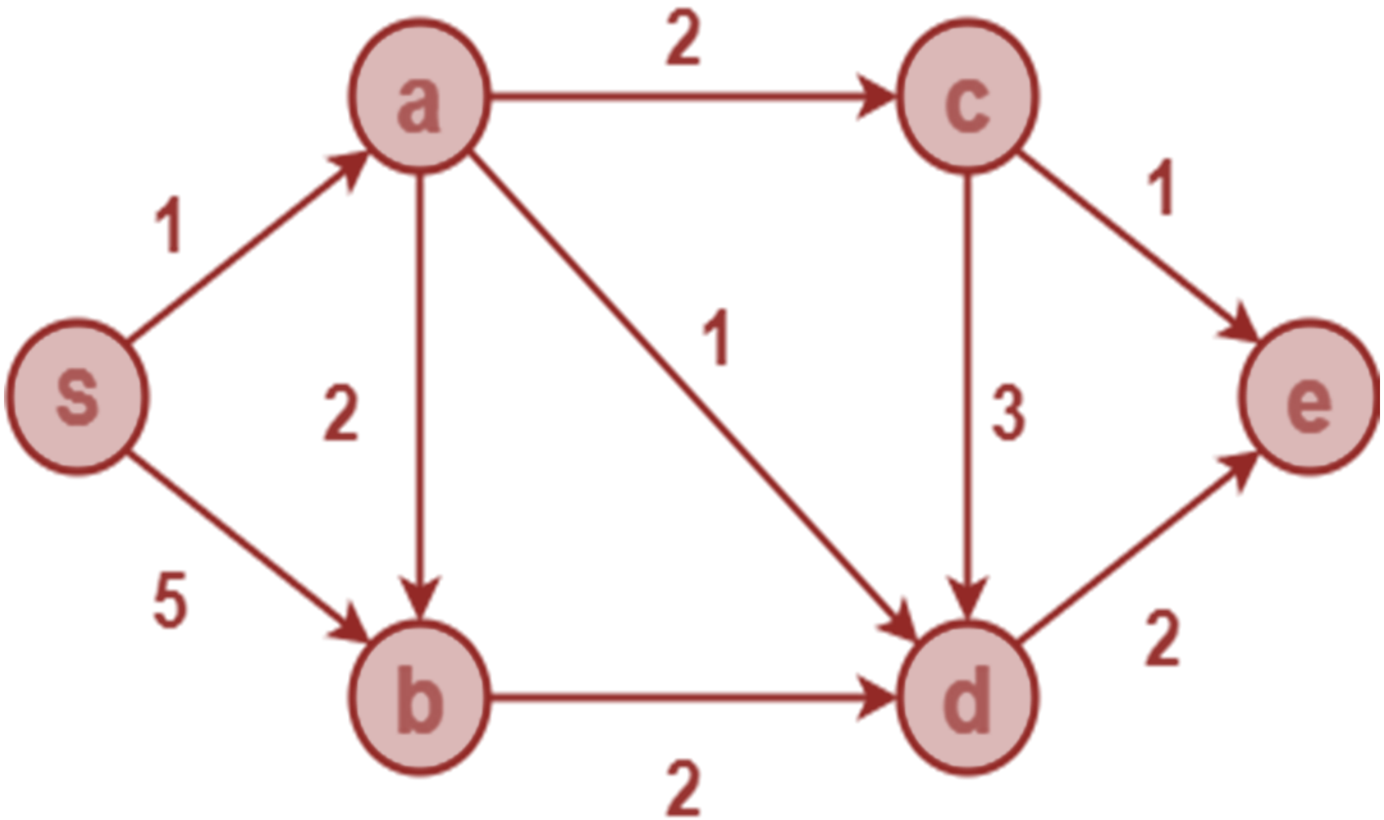
	A	**B**	**C**	**D**	**E**
A	0	∞	∞	∞	∞
C		10	5	∞	∞
		8		14	7

	A	**B**	**C**	**D**	**E**
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7

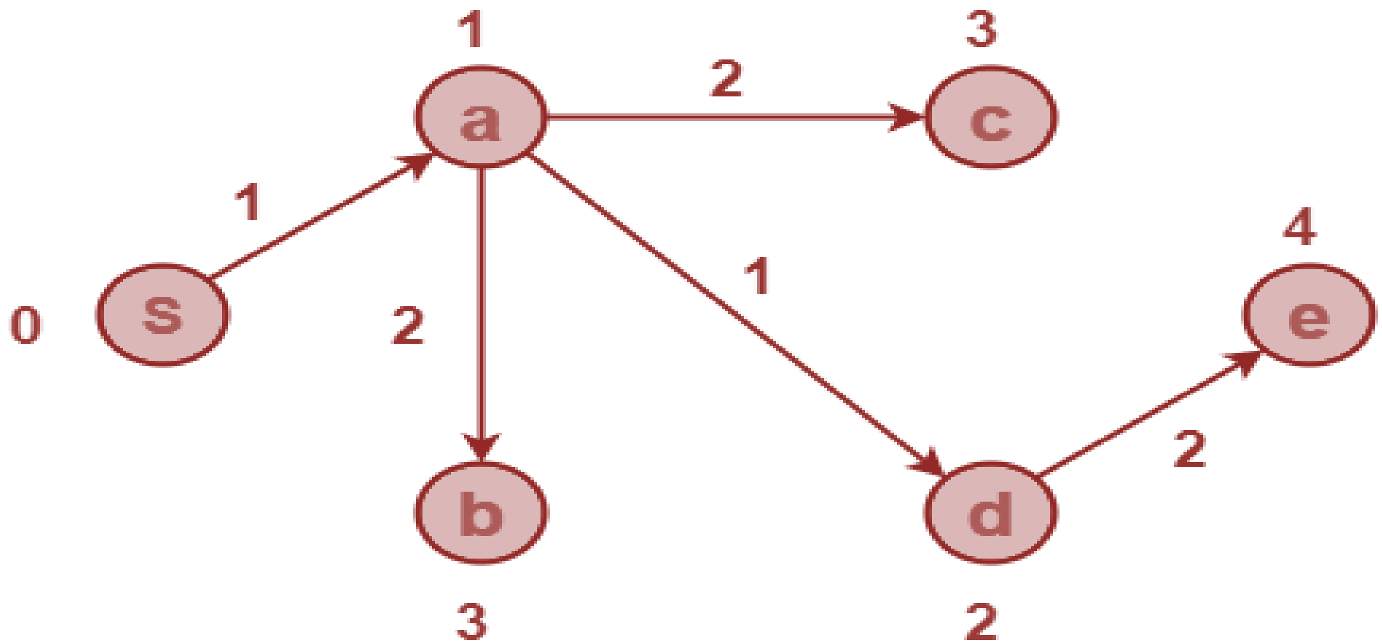
	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	
D				9	

Example 2



Final Shortest Path tree



Shortest Path Tree

Single Source Shortest Path Algorithm

Algorithm Shortest-Paths (v, cost, dist, n)

// dist [j], 1 < j < n, is set to the length of the shortest path from vertex v to vertex j in the digraph G with n vertices.
// dist [v] is set to zero.

```
{         for i :=1 to n do
          {         S [i] := false;                       // Initialize S.
                    dist [i] :=cost [v, i];   }
          S[v] := true;      dist[v] := 0.0;              // Put v in S.
          for num := 2 to n – 1 do
          {         Determine n - 1 paths from v.
                    Choose u among those vertices not in S such that dist[u]
is minimum;
          S[u] := true;                                   // Put u is S.
          for (each w adjacent to u with S [w] = false) do
                    if (dist [w] > (dist [u] + cost [u, w]) then
                              dist [w] := dist [u] + cost [u, w];    }    }
```

All Pairs Shortest Path Problem

- It is an algorithm for finding the shortest path between all the pairs of vertices in a given edge-weighted directed Graph.
- The Floyd-Warshall Algorithm is for solving all pairs of shortest-path problems.

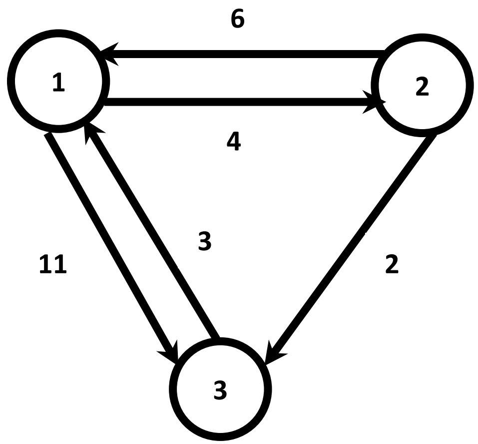
Steps to solve:

- Construct a Matrix for the given graph G by the following ways,
- Cost of the graph is cost of each edges and cost(i,i)=0
- If there is an edge between i and j then cost(i,j)=cost of the edge from i to j
- If there is no edge then cost(i,j) = ∞
- Calculate the shortest path between any two vertices using intermediary vertex.
- The minimum cost can be calculated using the formula,

$$\mathbf{A^{k}[i, j] = \min\{ A^{k-1}[i, j], A^{k-1}[i, k] + A^{k-1}[k, j] \}}$$

All Pairs Shortest Path Problem

Example 1:



Here, A^0=Cost= $\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$

When we calculate A^1 will omit column 1 and row 1 and calculate cost for rest of the 4 element.

$A^1(2,3)=\min\{A^{1-1}(2,3), A^{1-1}(2,1)+A^{1-1}(1,3)\}$

$=\min\{2,17\}=2$

$A^1(3,2)=\min\{A^{1-1}(3,2), A^{1-1}(3,1)+A^{1-1}(1,2)\}$

$=\min\{\infty,7\}=7$

$A^1= \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$

All Pairs Shortest Path Problem

Now we will calculate A^2

$A^2(1,3)=\min\{A^{2-1}(1,3), A^{2-1}(1,2)+A^{2-1}(2,3)\}$

$=\min\{11,6\}$

$=6$

$A^2(3,1)=\min\{A^{2-1}(3,1), A^{2-1}(3,2)+A^{2-1}(2,1)\}$

$=\min\{3,13\}$

$=3$

$$A^2=\begin{bmatrix}0 & 4 & 6\\ 6 & 0 & 2\\ 3 & 7 & 0\end{bmatrix}$$

All Pairs Shortest Path Problem

Now we will calculate A^3

$$A^3(1,2)=\min\{A^{3-1}(1,2),\ A^{3-1}(1,3)+A^{3-1}(3,2)\}$$
$$=\min\{4,13\}$$
$$=4$$

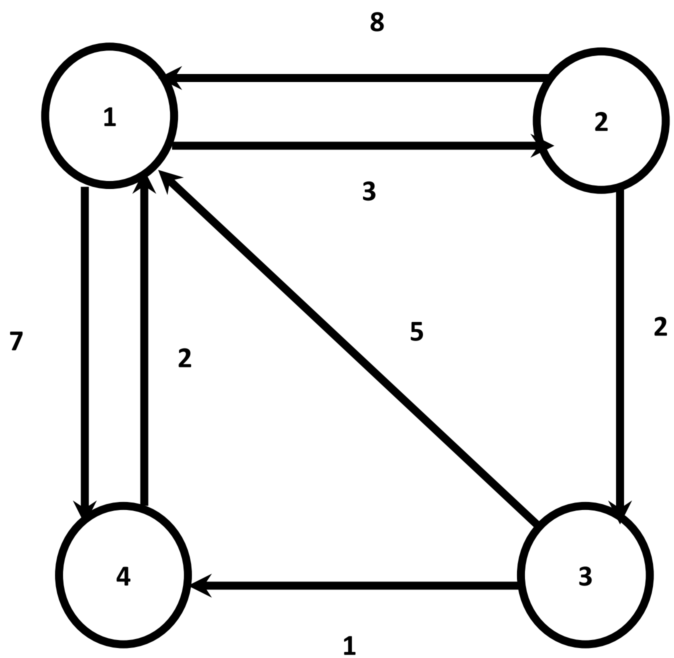
$$A^3(2,1)=\min\{A^{3-1}(2,1),\ A^{3-1}(2,3)+A^{3-1}(3,1)\}$$
$$=\min\{6,5\}$$
$$=5$$

$$A^3=\begin{bmatrix}0 & 4 & 6\\ 5 & 0 & 2\\ 3 & 7 & 0\end{bmatrix}$$

All Pairs Shortest Path Problem

<u>Example 2:</u>

<u>Solution:</u>



$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix} \end{array}$$

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

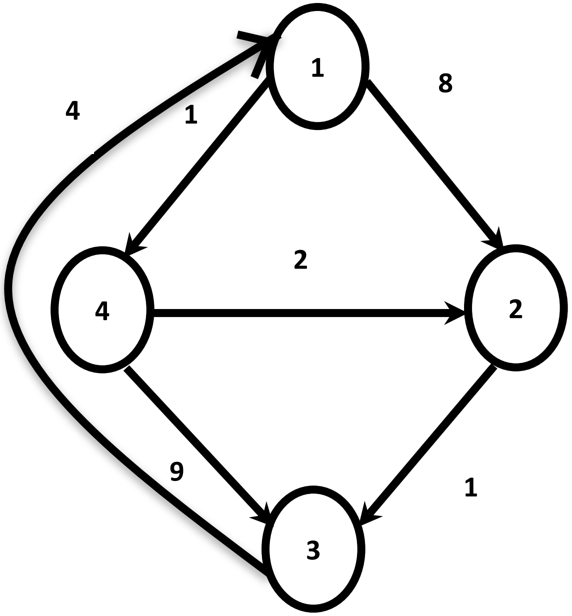
$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

All Pairs Shortest Path Problem

Example 3:

Solution:



$$A^0 = \begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$A^1 = \begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$A^2 = \begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$A^3 = \begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$A^4 = \begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

All Pairs Shortest Path Problem

```
import java.io.*;

import java.lang.*;

import java.util.*;

class AllPairShortestPath {

    final static int INF = 99999, V = 4;

    void floydWarshall(int dist[][])

        {       int i, j, k;

            for (k = 0; k < V; k++) {

            for (i = 0; i < V; i++) {

            for (j = 0; j < V; j++) {

            if (dist[i][k] + dist[k][j] < dist[i][j])

                    dist[i][j] = dist[i][k] + dist[k][j];          }          }      }

printSolution(dist);    }
```

All Pairs Shortest Path Problem

```
void printSolution(int dist[][])
    {           System.out.println("The following matrix shows the shortest " +
"distances between every pair of vertices");
        for (int i = 0; i < V; ++i) {
            for (int j = 0; j < V; ++j) {
                if (dist[i][j] == INF)
                    System.out.print("INF ");
                else
                    System.out.print(dist[i][j] + "   ");          }
            System.out.println();       }    }

public static void main(String[] args)
    {
  int graph[][] = { { 0, 5, INF, 10 },
                    { INF, 0, 3, INF },
                    { INF, INF, 0, 1 },
                    { INF, INF, INF, 0 } };
        AllPairShortestPath a = new AllPairShortestPath();
        a.floydWarshall(graph);    }}
```

All Pairs Shortest Path Problem

The running time of the algorithm is computed as :

$$T(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \Theta(1) = \sum_{k=1}^{n} \sum_{i=1}^{n} n = \sum_{k=1}^{n} n^2 = O(n^3)$$

Minimum Spanning Trees (MST)

Applications of Minimum Spanning Tree:

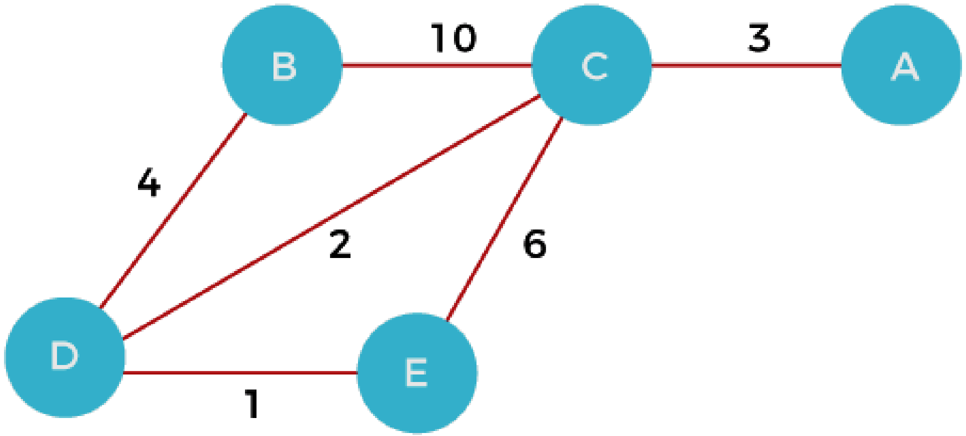
- Minimum Spanning Tree is used for designing telecommunication networks and water supply networks.
- For designing Local Area Networks.
- For solving the Travelling salesman problem.

Two algorithms to find Minimum Spanning Tree:

1. Kruskal’s algorithm - uses edges
2. Prim’s algorithm - uses vertex connections

Prim’s Algorithm

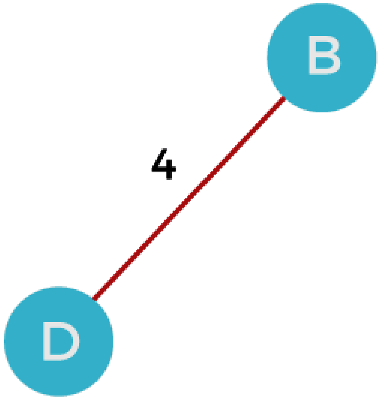
- find the minimum spanning tree from a graph
- Starts with the single node and explores all the adjacent nodes
- The edges with the minimal weights causing no cycles in the graph got selected



Prim’s Algorithm

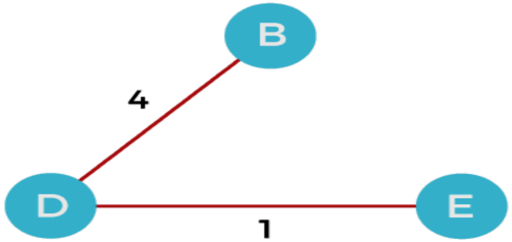
Step 1 - First, choose a vertex from the graph. Let's choose B.

Step 2 - Now, choose and add the shortest edge from vertex B. There are two edges from vertex B. Among the edges, the edge BD has the minimum weight. So, add it to the MST.

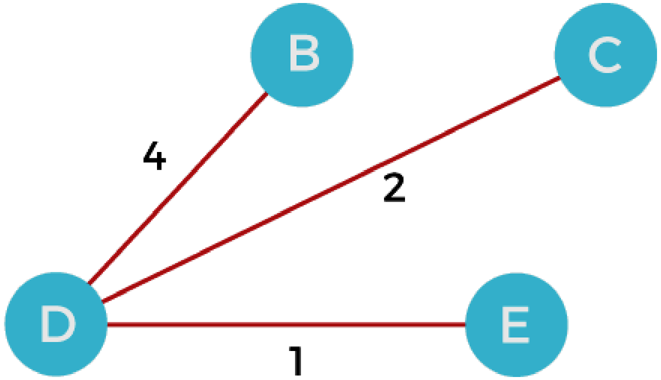


Prim’s Algorithm

Step 3 - Now, again, from D choose the edge with the minimum weight among all the other edges. So, select the edge DE and add it to the MST.

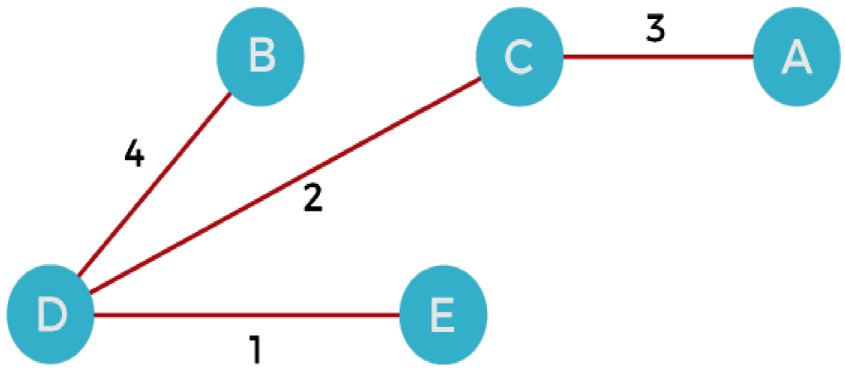


Step 4 - Now, select the edge CD, and add it to the MST.



Prim's Algorithm

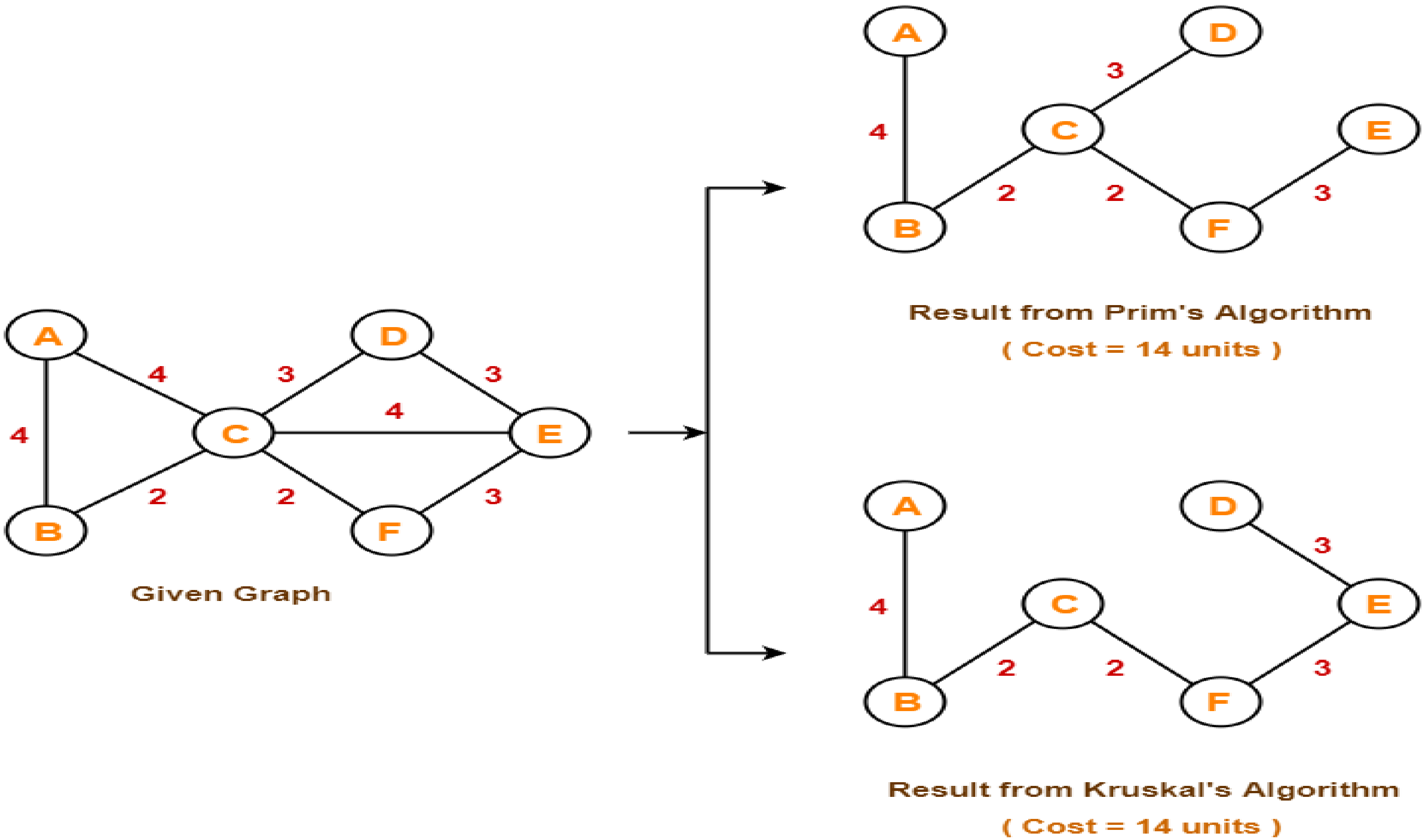
Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph.

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

Example - Prim's & Kruskal's Algorithm



Prim’s Algorithm

- For the Adjacency Matrix representation of the graph.

  Time Complexity: $O(V^2)$

  Space Complexity: $O(V^2)$

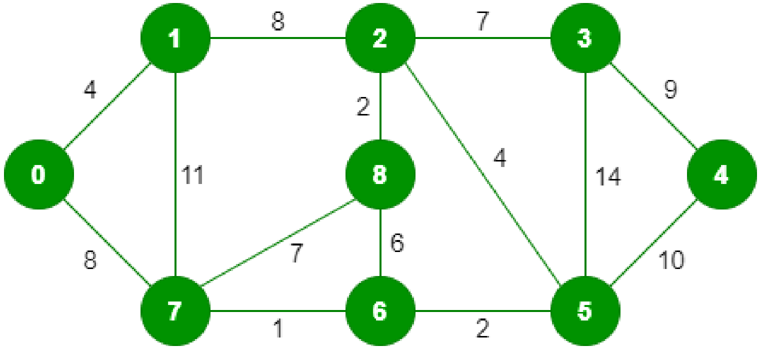
  In Prim's Algorithm, the time required for traversing the matrix is $O(V^2)$ so the overall time complexity is $O(V^2)$. Also to represent the matrix we use a 2-Dimensional array. So, we will require $O(V^2)$ space where V is the number of vertices in graph G.

Kruskal's Algorithm

A minimum spanning tree has (V – 1) edges where **V** is the number of vertices in the given graph.

Algorithm:

- Sort all the edges in non-decreasing order of their weight.
- Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
- Repeat step#2 until there are (V-1) edges in the spanning tree.



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

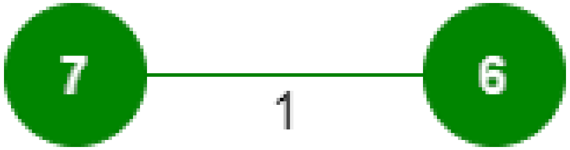
Kruskal's Algorithm

After sorting:

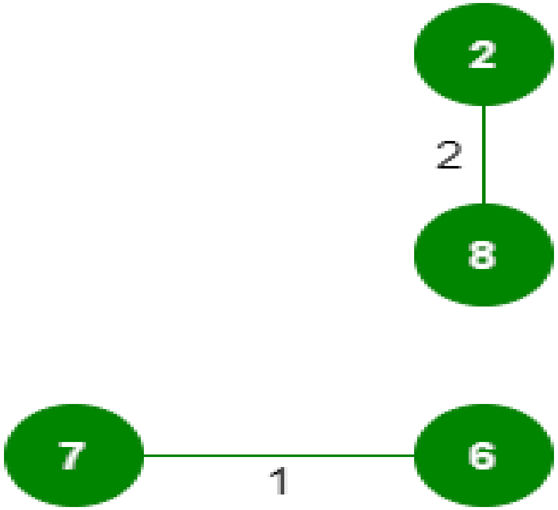
Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Kruskal's Algorithm

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

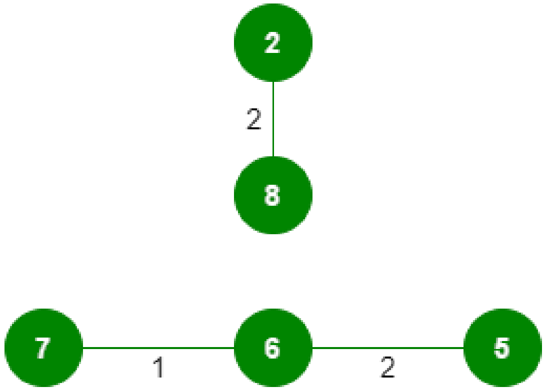


Step 2: Pick edge 8-2: No cycle is formed, include it.

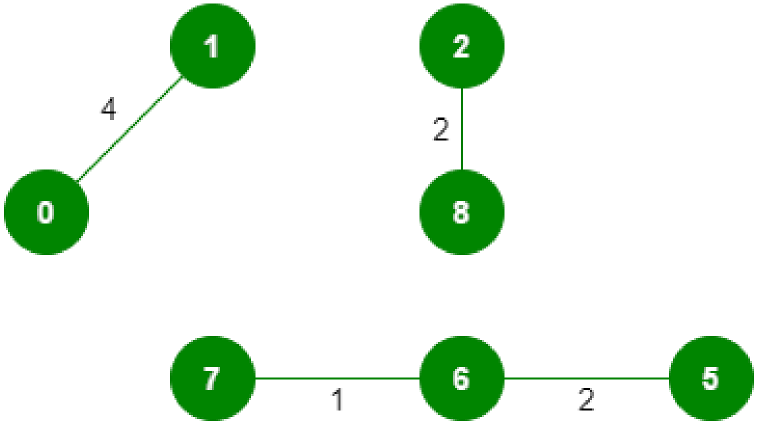


Kruskal’s Algorithm

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

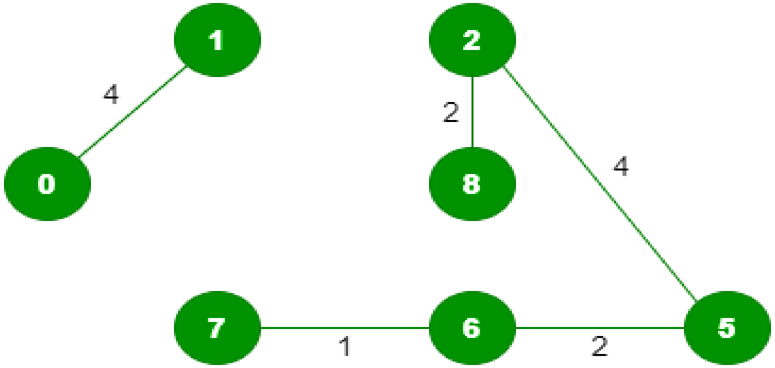


Step 4: Pick edge 0-1: No cycle is formed, include it.

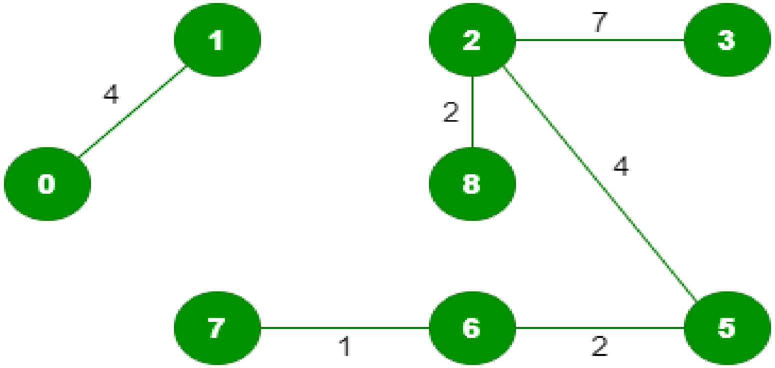


Kruskal's Algorithm

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

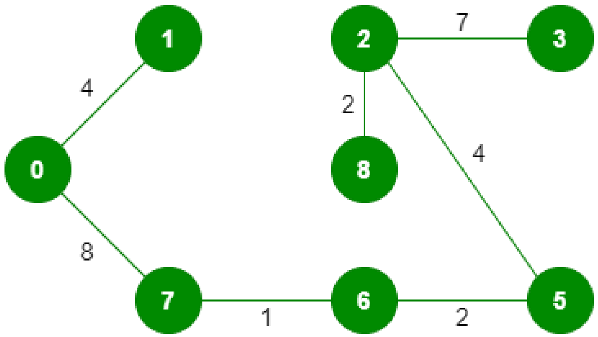


Step 6: Pick edge 8-6: Since including this edge results in the cycle, discard it.
Step 7: Pick edge 2-3: No cycle is formed, include it.

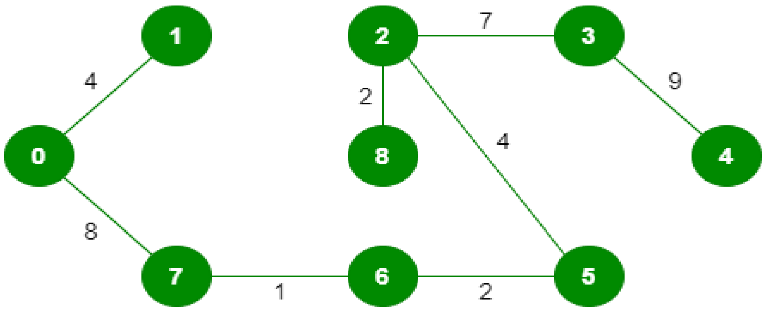


Kruskal’s Algorithm

Step 8: Pick edge 7-8: Since including this edge results in the cycle, discard it.
Step 9: Pick edge 0-7: No cycle is formed, include it.

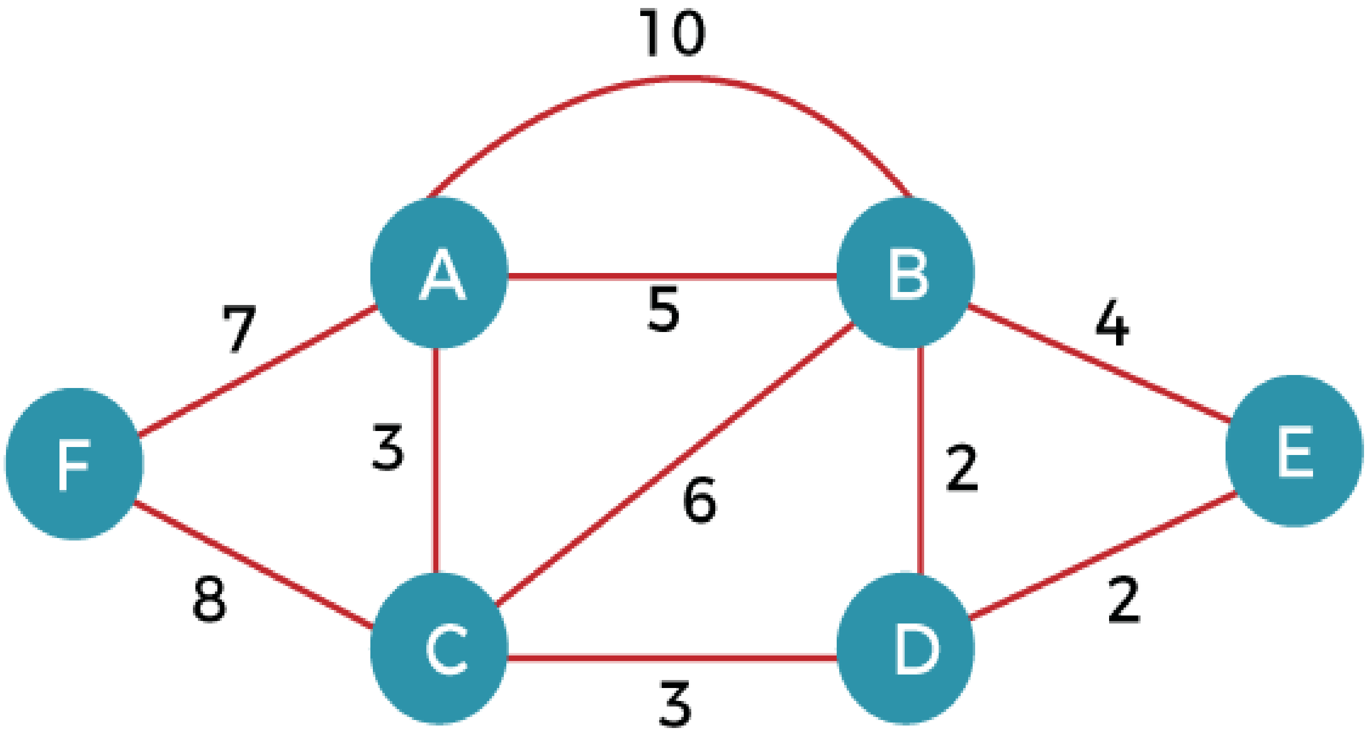


Step 10: Pick edge 1-2: Since including this edge results in the cycle, discard it.
Step 11: Pick edge 3-4: No cycle is formed, include it.



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

Kruskal’s Algorithm



Kruskal's algorithm

Running time:

- The time complexity of Kruskal's Algorithm is **O(ElogE)**, where E is the number of edges in the graph.
- This complexity is because the algorithm uses a priority queue with a time complexity of O(logE).
- However, the space complexity of the algorithm is **O(E)**, which is relatively high.