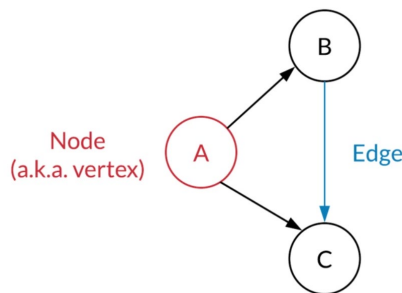


## **Graph Theory – Part 1**

### What is graph?

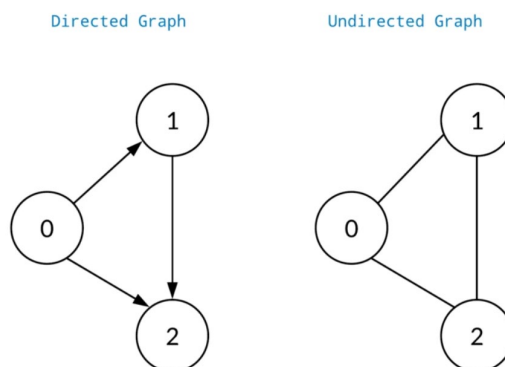
A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices (  $V$  ) and a set of edges (  $E$  ). The graph is denoted by  $G (E, V)$ .



### Directed & undirected graphs

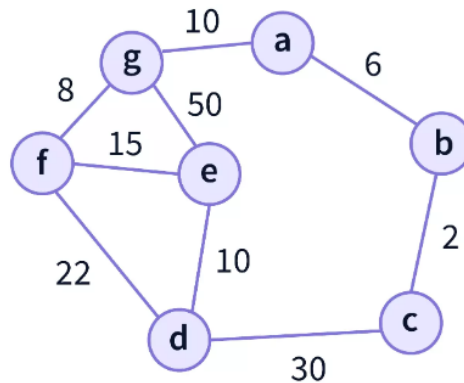
A directed graph is a graph that in which all edges are associated with a direction. An example of a directed edge would be a one-way street.

An undirected graph is a graph in which all edges do not have a direction. An example of this would be a friendship!



### Weighted Graph

In weighted graphs, each edge has a value associated with them (called weight).



## Degree

It is the number of vertices adjacent to a vertex  $V$ .

**Notation** –  $\deg(V)$ .

In a simple graph with  $n$  number of vertices, the degree of any vertices is –

$$\deg(v) = n - 1 \quad \forall v \in G$$

A vertex can form an edge with all other vertices except by itself. So the degree of a vertex will be up to the number of vertices in the graph minus 1. This 1 is for the self-vertex as it cannot form a loop by itself. If there is a loop at any of the vertices, then it is not a Simple Graph.

Degree of vertex can be considered under two cases of graphs –

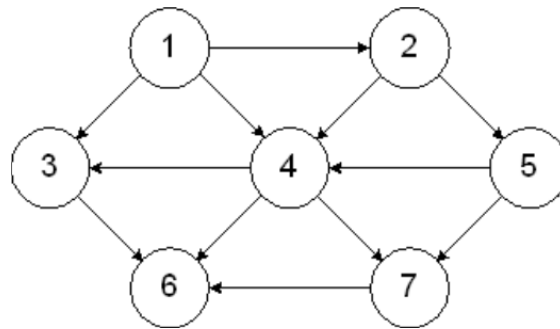
1. Undirected Graph
2. Directed Graph

For a directed graph  $G=(V(G),E(G))$  and a vertex  $x_1 \in V(G)$ , the Out-Degree of  $x_1$  refers to the number of arcs incident from  $x_1$ . That is, the number of arcs directed away from the vertex  $x_1$ . The In-Degree of  $x_1$  refers to the number of arcs incident to  $x_1$ . That is, the number of arcs directed towards the vertex  $x_1$ .

In simple words, the number of edges coming towards a vertex( $v$ ) in Directed graphs is the in degree of  $v$  and the number of edges going out from a vertex( $v$ ) in Directed graphs is the in degree of  $v$ .

### Example:

In the given figure:



- vertex 4 has 3 incoming edges and 3 outgoing edges, so in-degree is 3 and out-degree is 3.
- vertex 7 has 2 in-degree and 1 out-degree

## Connectivity

A graph is said to be **connected if there is a path between every pair of vertex**. From every vertex to any other vertex, there should be some path to traverse. That is called the connectivity of a graph. A graph with multiple disconnected vertices and edges is said to be disconnected.

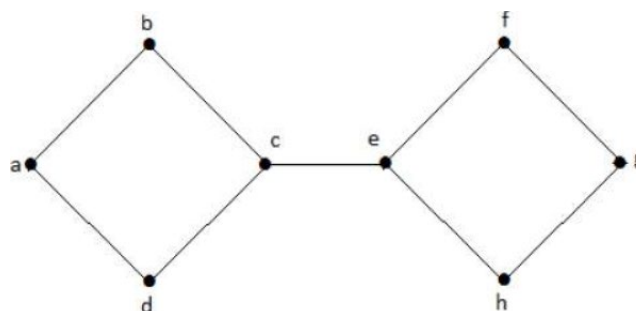
## Cut Vertex / Articulation Point

A single vertex whose removal disconnects a graph is called a cut-vertex or articulation point.

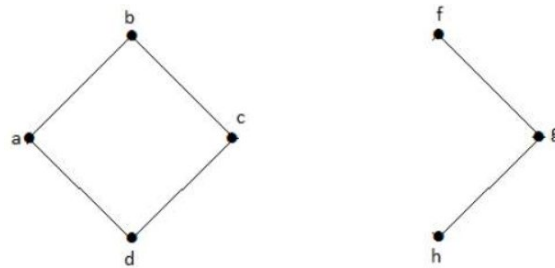
Let  $G$  be a connected graph. A vertex  $v$  of  $G$  is called a cut vertex of  $G$ , if  $G-v$  (Remove  $v$  from  $G$ ) results a disconnected graph. When we remove a vertex from a graph then graph will break into two or more graphs. This vertex is called a cut vertex.

### Example:

In the following graph, vertices 'e' and 'c' are the cut vertices.



By removing 'e', the graph will become a disconnected graph.



Some more explanation over the Articulation Point

In a graph, a vertex is called an articulation point if removing it and all the edges associated with it results in the increase of the number of connected components in the graph. For example, consider the graph given in following figure.

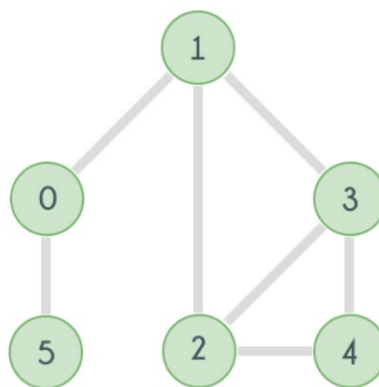


Fig. 1

If in the above graph, vertex 1 and all the edges associated with it, i.e. the edges 1-0, 1-2 and 1-3 are removed, there will be no path to reach any of the vertices 2, 3 or 4 from the vertices 0 and 5, that means the graph will split into two separate components. One consisting of the vertices 0 and 5 and another one consisting of the vertices 2, 3 and 4 as shown in the following figure.

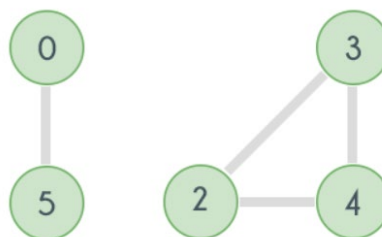
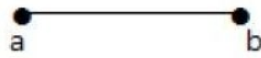


Fig. 2

Likewise removing the vertex 0 will disconnect the vertex 5 from all other vertices. Hence the given graph has two articulation points: 0 and 1.

## Pendant Vertices

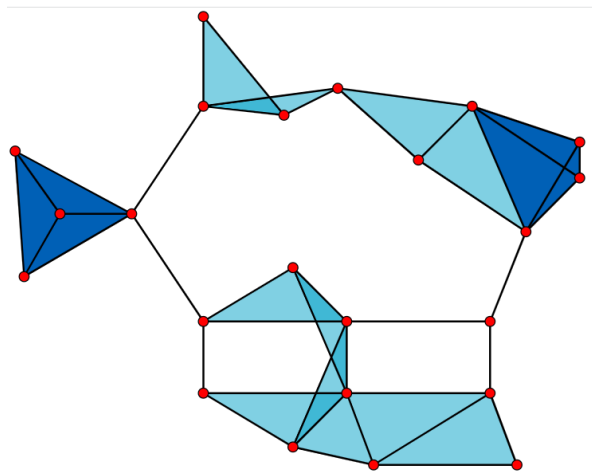
Let  $G$  be a graph, a vertex  $v$  of  $G$  is called a pendant vertex if and only if  $v$  has degree 1. In other words, pendant vertices are the vertices that have degree 1, also called pendant vertex.



Here, in this example, vertex 'a' and vertex 'b' have a connected edge 'ab'. So with respect to the vertex 'a', there is only one edge towards vertex 'b' and similarly with respect to the vertex 'b', there is only one edge towards vertex 'a'. Finally, vertex 'a' and vertex 'b' has degree as one which are also called as the **pendent vertex**.

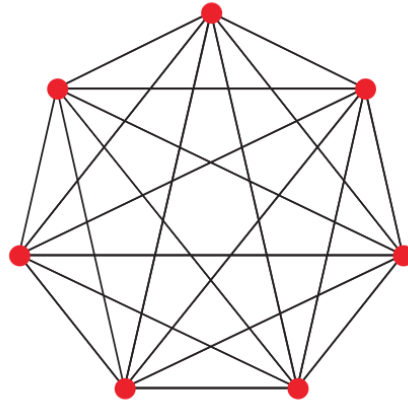
## Clique

In the mathematical area of graph theory, a clique is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent. That is, a clique of a graph  $G$  is an induced subgraph (An induced subgraph of a graph is another graph, formed from a subset of the vertices of the graph and all of the edges connecting pairs of vertices in that subset.) of  $G$  that is complete.



## Complete Graph

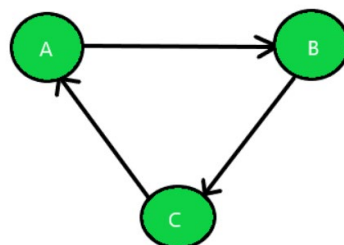
A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge. Moreover, that, a complete digraph is a directed graph in which every pair of distinct vertices is connected by a pair of unique edges (one in each direction).



## Connected Graph

A graph is said to be connected if every pair of vertices in the graph is connected. This means that there is a path between every pair of vertices. An undirected graph that is not connected is called disconnected. An undirected graph  $G$  is therefore disconnected if there exist two vertices in  $G$  such that no path in  $G$  has these vertices as endpoints. A graph with just one vertex is connected.

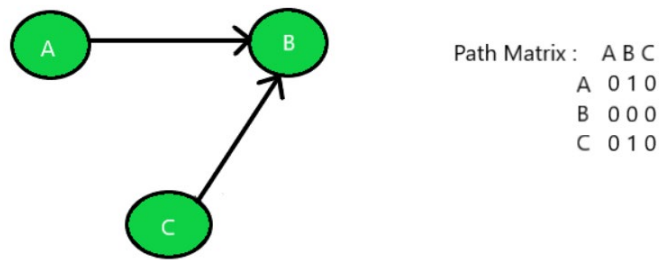
**Strongly Connected:** A digraph is strongly connected if every vertex is reachable from every other following the directions of the arcs. I.e., for every pair of distinct vertices  $u$  and  $v$  there exists a directed path from  $u$  to  $v$ . The elements of the path matrix of such a graph will contain all 1's.



Path Matrix :    A B C  
                   A 1 1 1  
                   B 1 1 1  
                   C 1 1 1

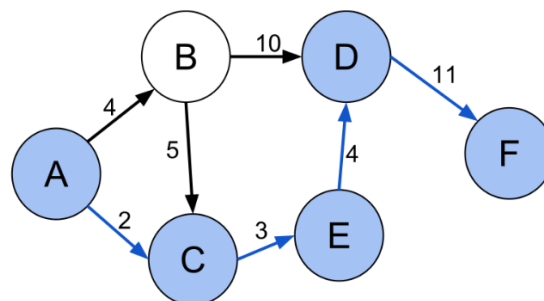
**Weakly Connected:** A digraph is weakly connected if when considering it as an undirected graph it is connected. I.e., for every pair of distinct vertices  $u$  and  $v$  there exists an undirected path (potentially running opposite the direction on an edge) from  $u$  to  $v$ .

In other words, a weakly connected digraph is a directed graph in which it is possible to reach any node starting from any other node by traversing edges in some direction (i.e., not necessarily in the direction they point). The nodes in a weakly connected digraph therefore must all have either outdegree or indegree of at least 1.



## Shortest Path

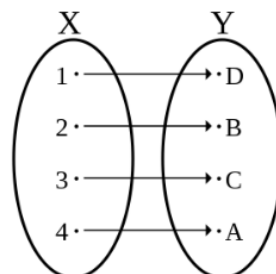
In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.



## Isomorphism

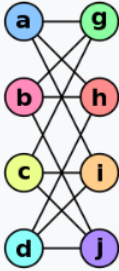
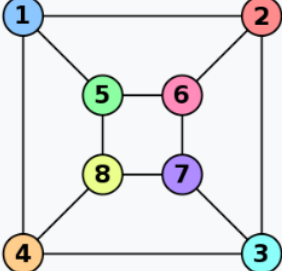
In graph theory, an isomorphism of graphs  $G$  and  $H$  is a bijection between the vertex sets of  $G$  and  $H$ . such that any two vertices  $u$  and  $v$  of  $G$  are adjacent in  $G$  if and only if and are adjacent in  $H$ .

In mathematics, a bijection, also known as a **bijective** function, one-to-one correspondence.



### Example:

The two graphs shown below are isomorphic:

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

## Graph representation

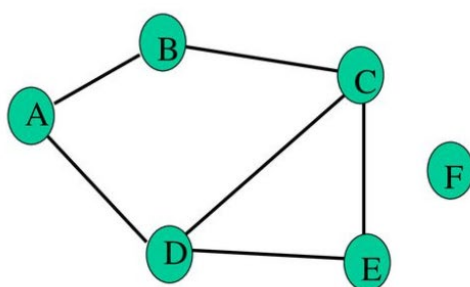
There are two ways to store Graphs into the computer's memory:

1. Sequential representation (or, Adjacency matrix representation)
2. Linked list representation (or, Adjacency list representation)

## Graph Representations

The *adjacency matrix* representation:

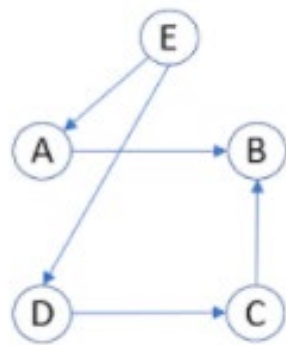
$$M(v, w) = \begin{cases} 1 & \text{if } (v, w) \in E \\ 0 & \text{otherwise} \end{cases}$$



$$\begin{matrix} & \begin{matrix} A & B & C & D & E & F \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \end{matrix} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{matrix}$$

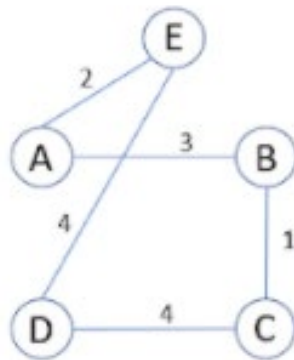
Some more example:





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

Adjacency Matrix for a directed graph



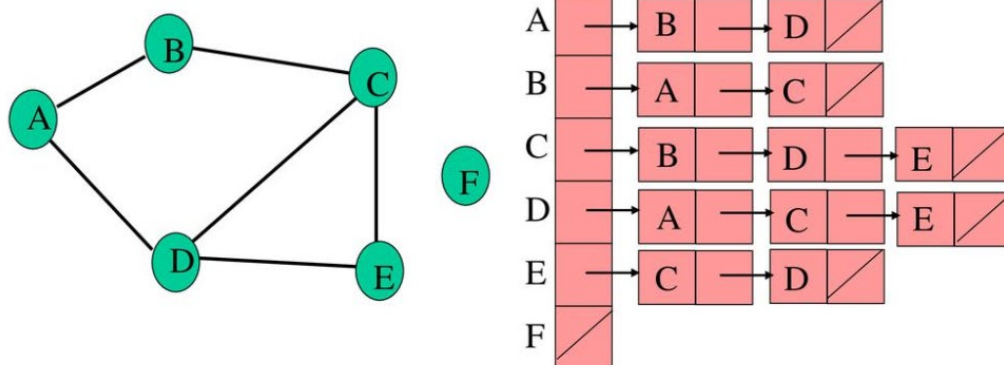
	A	B	C	D	E
A	0	3	0	0	2
B	3	0	1	0	0
C	0	1	0	4	0
D	0	0	4	0	4
E	2	0	0	4	0

Adjacency Matrix for a weighted graph

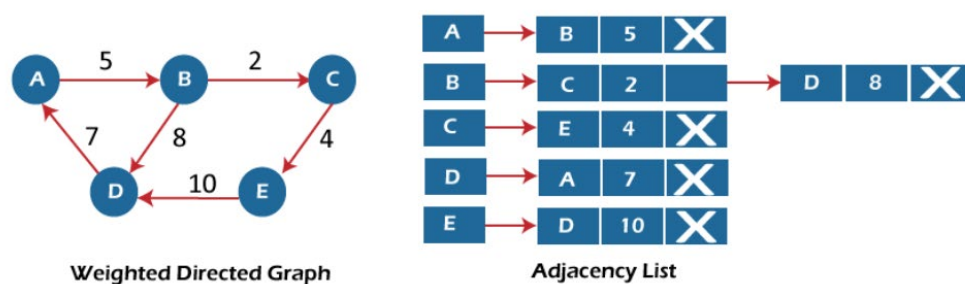
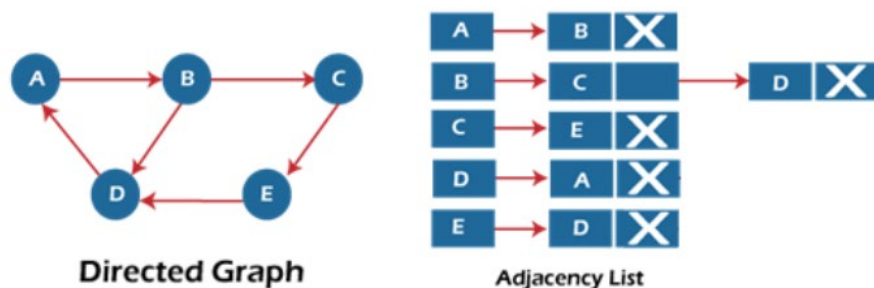
## Graph Representations

The *adjacency list* representation:

$$L(v) = \text{list of } w \text{ such that } (v, w) \in E, \\ \text{for } v \in V$$



Some more example:



## Graph Traversal

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

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

**Step 1** - Define a Stack of size total number of vertices in the graph.

**Step 2** - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

**Step 3** - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.

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

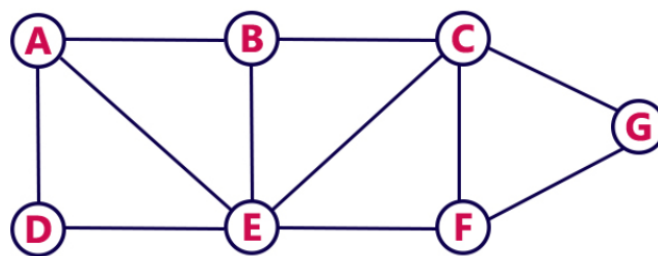
**Step 5** - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.

**Step 6** - Repeat steps 3, 4 and 5 until stack becomes Empty.

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

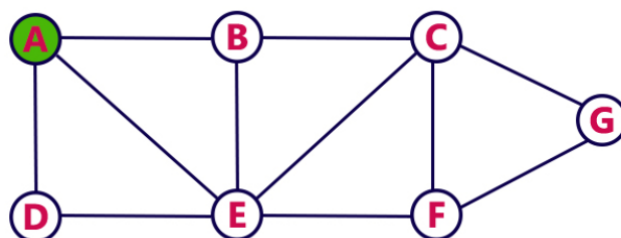
### Example:

Consider the following example graph to perform DFS traversal



#### Step 1:

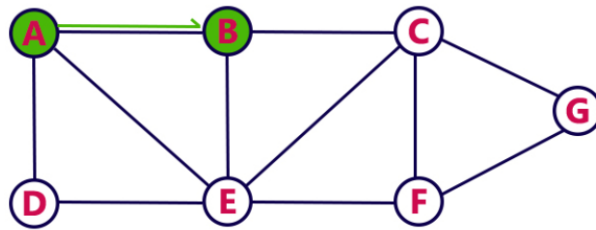
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



**Stack**

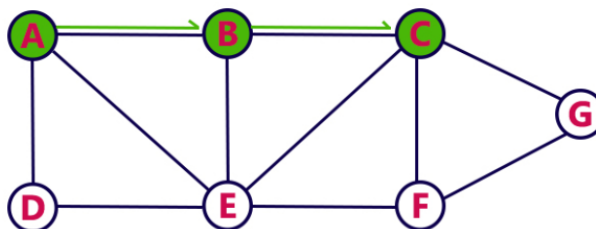
**Step 2:**

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



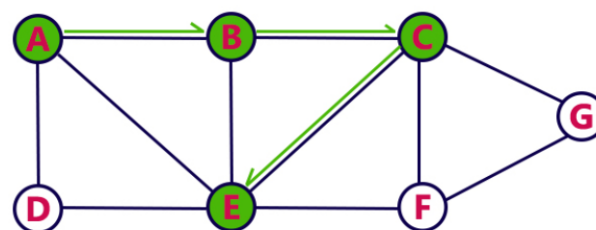
**Step 3:**

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



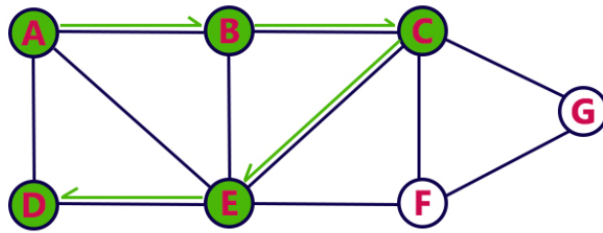
**Step 4:**

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



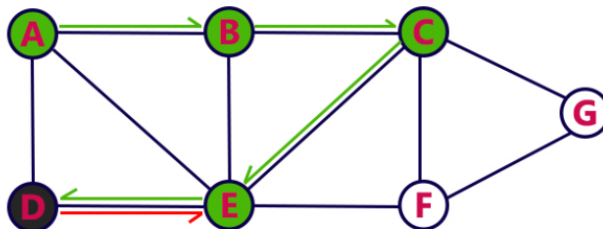
**Step 5:**

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



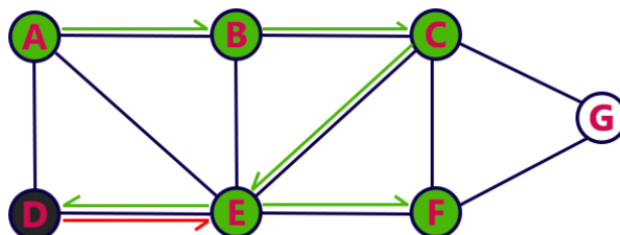
**Step 6:**

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



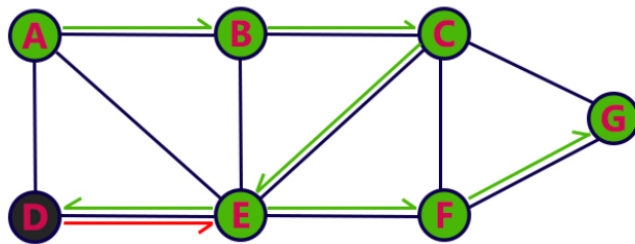
**Step 7:**

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



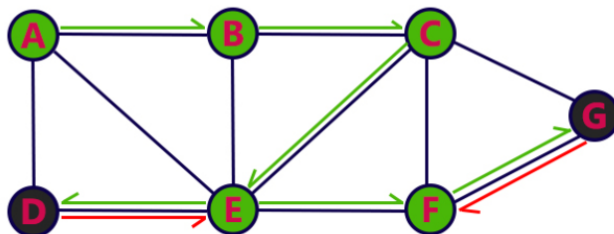
**Step 8:**

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



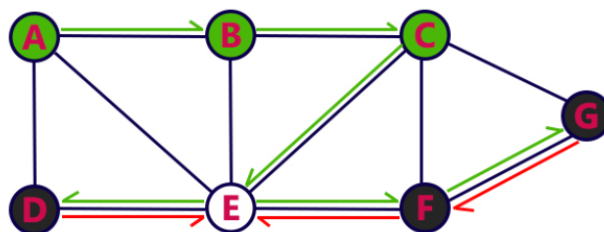
**Step 9:**

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



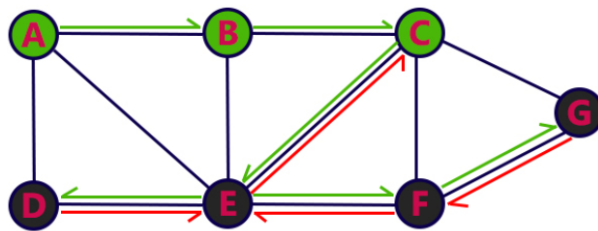
**Step 10:**

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



**Step 11:**

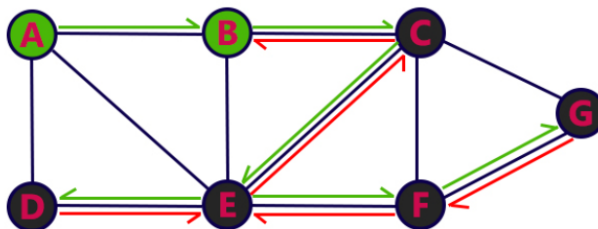
- There is no new vertex to be visited from E. So use back track.
- Pop E from the Stack.



**Stack**

**Step 12:**

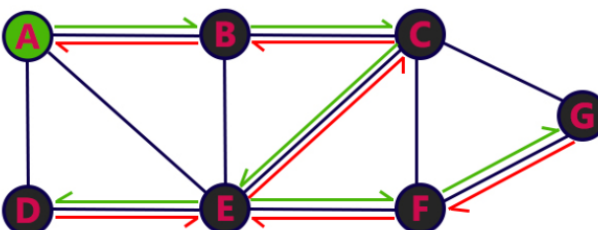
- There is no new vertex to be visited from C. So use back track.
- Pop C from the Stack.



**Stack**

**Step 13:**

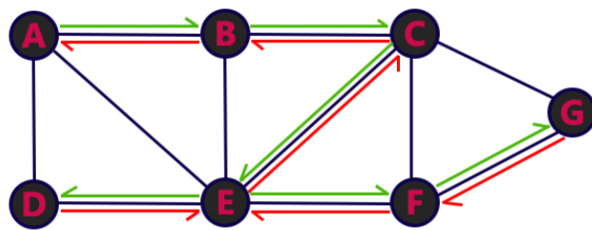
- There is no new vertex to be visited from B. So use back track.
- Pop B from the Stack.



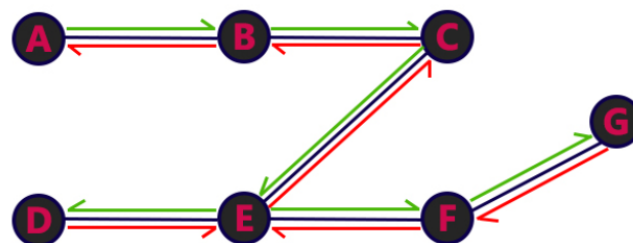
**Stack**

**Step 14:**

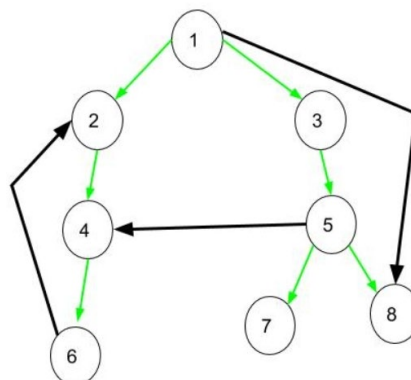
- There is no new vertex to be visited from A. So use back track.
- Pop A from the Stack.



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



Some definition related to DFS, consider the follow DFS output:



- **Tree Edge:** It is an edge which is present in the tree obtained after applying DFS on the graph. All the Green edges are tree edges.
- **Forward Edge:** It is an edge  $(u, v)$  such that  $v$  is a descendant but not part of the DFS tree. An edge from 1 to 8 is a forward edge.
- **Back edge:** A Back Edge is an edge that connects a vertex to a vertex that is discovered before it's parent. It is an edge  $(u, v)$  such that  $v$  is the ancestor of node  $u$  but is not part of the DFS tree. Edge from 6 to 2 is a back edge. Presence of back edge indicates a cycle in directed graph.
- **Cross Edge:** It is an edge that connects two nodes such that they do not have any ancestor and a descendant relationship between them. The edge from node 5 to 4 is a cross edge.



## 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.

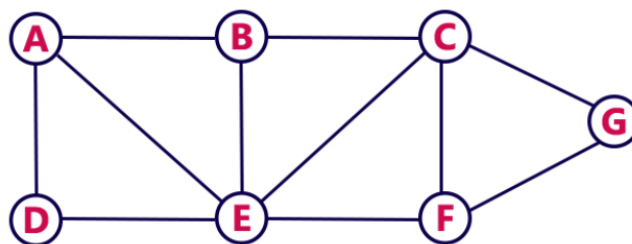
**Step 4** - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

**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

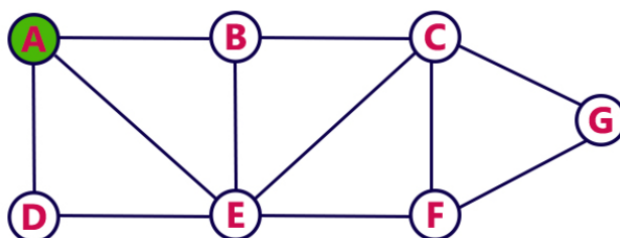
### Example:

Consider the following example graph to perform BFS traversal



#### Step 1:

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

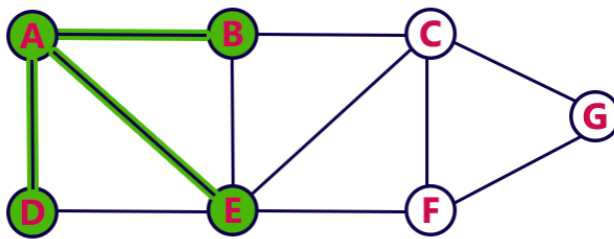


#### Queue



**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..

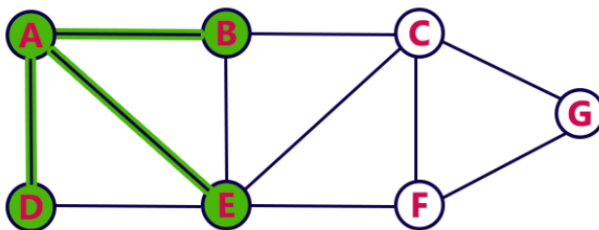


**Queue**



**Step 3:**

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

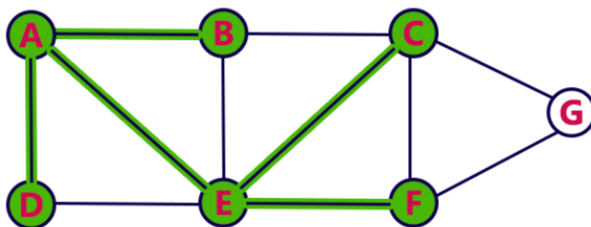


**Queue**



**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.

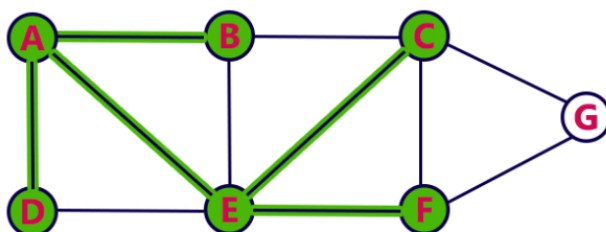


**Queue**



**Step 5:**

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

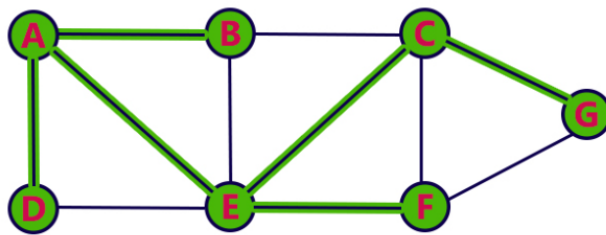


**Queue**



**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.

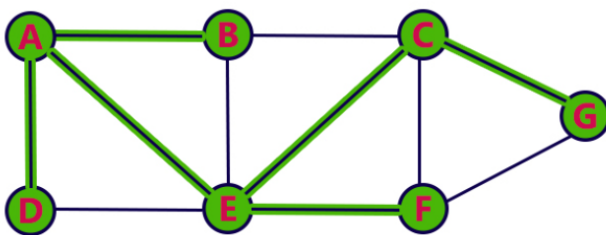


**Queue**



**Step 7:**

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

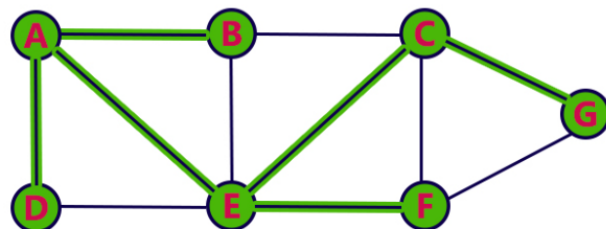


**Queue**



**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...

