UNIT-3

Introduction of graphs : Representation of graphs by using linked list & adjacency matrix, graph operations & algorithms: insert an edge, delete an edge, insert a node & delete a node. Graph traversal algorithms: Breadth first search & Depth First search algorithm

Representation of graphs :-

A graph is a data structure that consists a sets of vertices (called nodes) & Edges there are two ways to store graphs in to computer's memory.

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

In Sequential representation an adjacency matrix is used to store the graph. whereas in linked list representation there is a use of an adjacency list to store the graph.

Sequential representation :-

There is use of an Adjacency matrix to represent the mapping b/w vertices & edges of the graph. we can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, & weighted undirected graph

if adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w

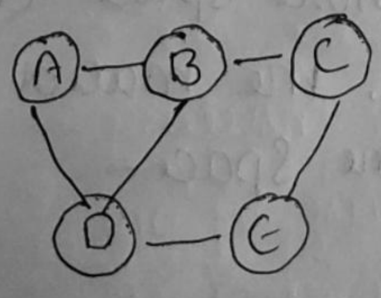
An entry A_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists b/w V_i & V_j if an undirected graph G consists of n vertices then the adjacency matrix for the graph is n×n & the matrix A = [a_{ij}] can be defined as

a_{ij} = 1 {if there is a path exists from V_i to V_j}

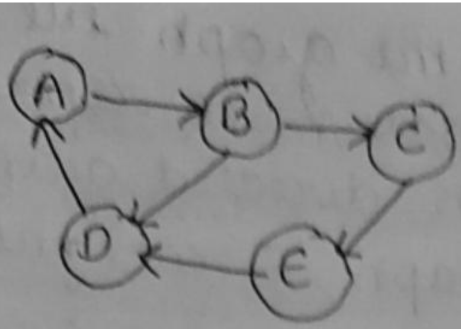
a_{ij} = 0 {otherwise}

If there is no self-loop present in the graph it means that the diagonal entries of the adjacency matrix will be 0

Ex:



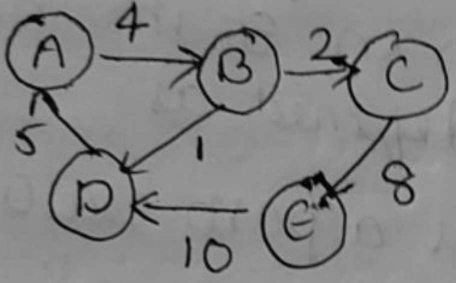
$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$



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

Adjacency matrix for weighted directed

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for existence of a path here we have to use the weight associated with the edge.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

Advantages :-

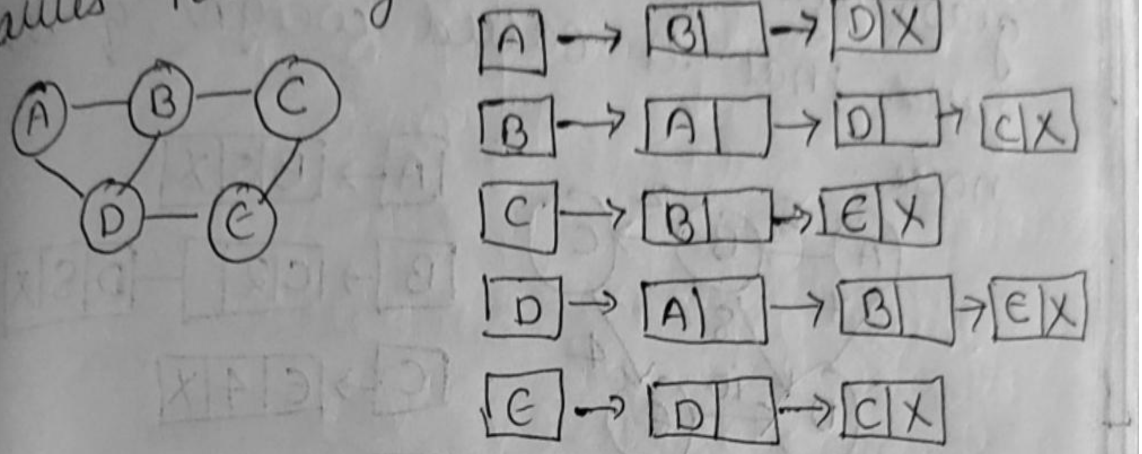
→ Easier to implement & follow

Disadvantage :-

→ It consumes more space. Even if the graph is sparse, the matrix still consumes the same space.

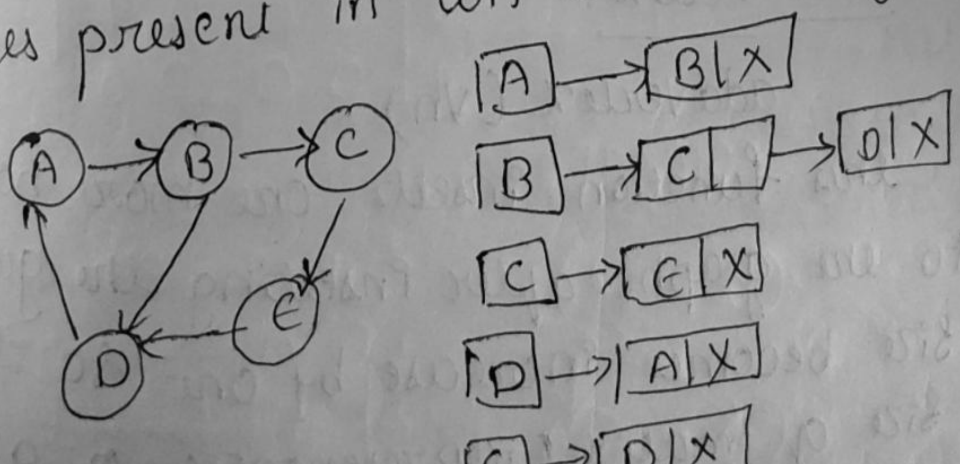
Linked list representation :-

An adjacency list is used in the linked representation to store the graph in the computers memory. It is Efficient in terms of storage as we only have to store the values for Edges



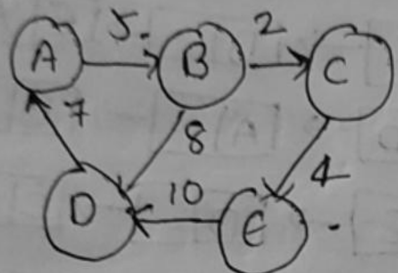
An adjacency list is maintained for each node present in the graph, which stores the node value & a pointer to the next adjacent node to the respective node if all the adjacent nodes are traversed then store the null in the pointer field of the last node of the list.

The sum of the lengths of adjacency lists is equal to twice the no. of edges present in an undirected graph



For a directed graph the sum of the lengms of adjacency lists is equal to no. of Edges present in the graph.

In the case of a weighted directed graph each node contains an extra field that is called weight of the node.



A → B|5|X

B → C|2| → D|8|X

C → E|4|X

D → A|7|X

E → D|10|X

Graph operations & Algorimms :-

Graph operations 1) Insert a Edge

2) Delete a Edge

3) Insert a node/Vertex

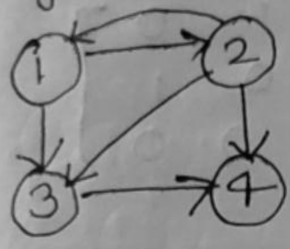
4) delete a node/vertex.

Insert vertex :-

addvertex (Vn)

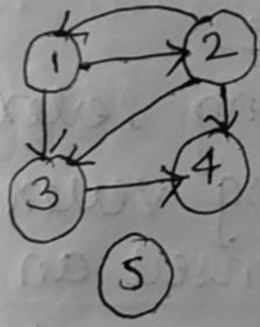
this function inserts one more node in to the graph, after inserting the graph size becomes increase by one. so the size of matrix (representation of graph) increases by 1 at row level & column level means simply after inserting vertex n×n becomes (n+1)×(n+1)

→ The newly inserted vertex do not have indegree or out degree



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 \end{array}$$

Add vertex 5



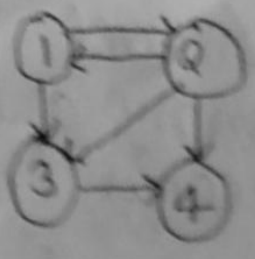
$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 & 0 \\ 3 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Delete vertex :-

deletevertex (VG)

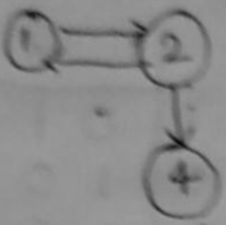
This function used to delete specified node/vertex which are present in the stored graph G

If the vertex is present in the graph In matrix representation the node is deleted & associated edges are removed



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 \end{array}$$

Delete vertex 3



$$\begin{array}{c|ccc} & 1 & 2 & 4 \\ \hline 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 4 & 0 & 0 & 0 \end{array}$$

Insert an Edge :-

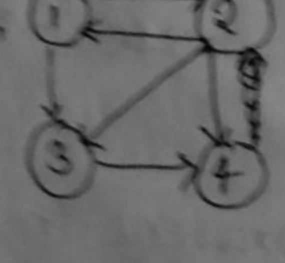
addEdge (vs, ve)

where $v_s \rightarrow$ Starting vertex.

$v_e \rightarrow$ Ending vertex

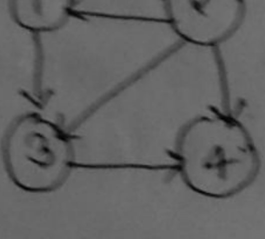
This function used to insert an edge b/w two vertices. If two vertices that specific in addEdge are available in given graph then

G[vs][ve] = cost of edge.



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 \end{array}$$

G[4][2] = 1



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 0 & 1 & 0 & 0 \end{array}$$

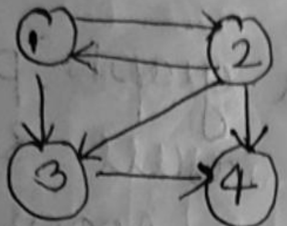
Delete edge :-

deleteEdge (Vs, Ve) ;

This function used to delete an edge b/w two vertices those are

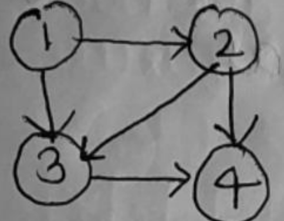
Vs → Starting vertex

Ve → Ending vertex.



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 \end{array}$$

$G[2][1] = 0$



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 \end{array}$$

Graph Traversal Techniques :-

Graph traversal is a technique used for searching a 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 edgis 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 in to looping pam.

There are two graph Traversal techniques & They are as follows

1) DFS (Depm first Search)
2) BFS (Breadm first search)

BFS :-

BFS traversal of a graph produces a Spanning Tree as final result. Spanning tree is a graph without loops. we use queue Data Structure with max size of total no. of vertices in the graph to impement BFS traversal.

Algorithm :-

1) Define a queue of size total no. of vertices in the graph

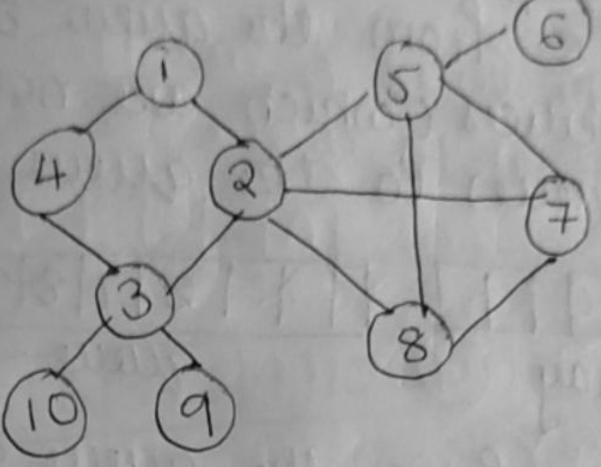
2) Select any vertex as starting point for traversal. visit that vertex & insert it in to the queue.

3) Visit all the non-visited adjacent vertices of the vertex which is at front of the queue & insert them in to the queue.

4) when there is no new vertex to be visited from the vertex which is at front of the queue then delete that vertex

5) Repeat steps 3 & 4 untill queue becomes Empty

6) when queue becomes empty then produce final Spanning tree by removing unused edges from the graph



visited array

1									

queue

4	2						

Dequeue 4 from the queue & add it to visited array & add the adjacent nodes of 4 to the queue.

visited array.

1	4								

queue

~~4~~	2	3							

Dequeue 2 from the queue & add it to visited array & add the adjacent nodes of 2 (which are not visited & already in the queue) to the queue.

1	4	2							

visited array

	~~2~~	3	5	7	8				

Queue.

Dequeue 3 from the queue & add its adjacent vertices (which are not in visited array & queue) to the queue.

1	4	2	3						

visited array

		~~3~~	5	7	8	9	10		

queue

Dequeue 5 from the queue & add its adjacent vertices (which are not in visited array & queue) to the queue.

1	4	2	3	5					

visited array

			~~5~~	7	8	9	10	6	

queue.

Dequeue 7 from the queue & add its adjacent vertices (which are not in visited array & queue) to the queue.

1	4	2	3	5	7			

visited array

				~~7~~	8	9	10	6

queue.

Dequeue 8 from the queue from & add its adjacent vertices (which are not in visited array & queue) to the queue.

1	4	2	3	5	7	8		

visited array

					~~8~~	9	10	6

Dequeue 9 from the queue & add its adjacent vertices (which are not in visited array & queue) to the queue.

1	4	2	3	5	7	8	9		

visited array

					~~9~~	10	6

Dequeue 10 from the queue & add its adjacent vertices (which are not in visited array & queue) to the queue.

1	4	2	3	5	7	8	9	10

visited array

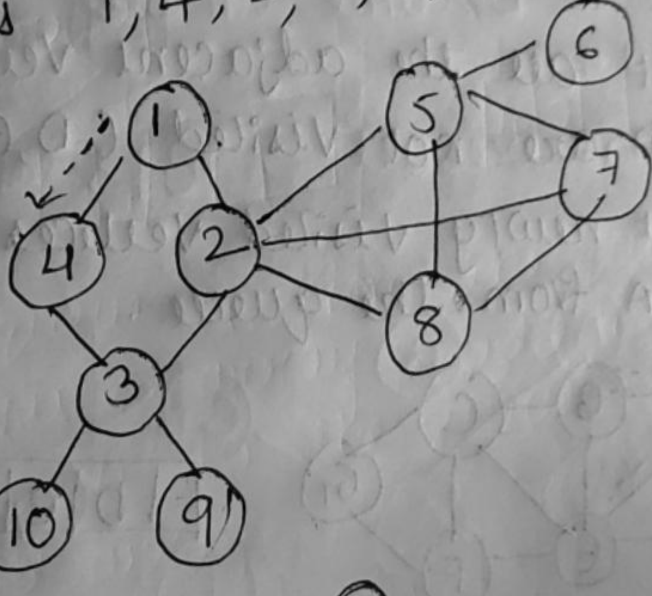
						~~10~~	6

Dequeue ~~9~~6 from the queue & add its adjacent vertices (which are not in visited array & queue) to the queue.

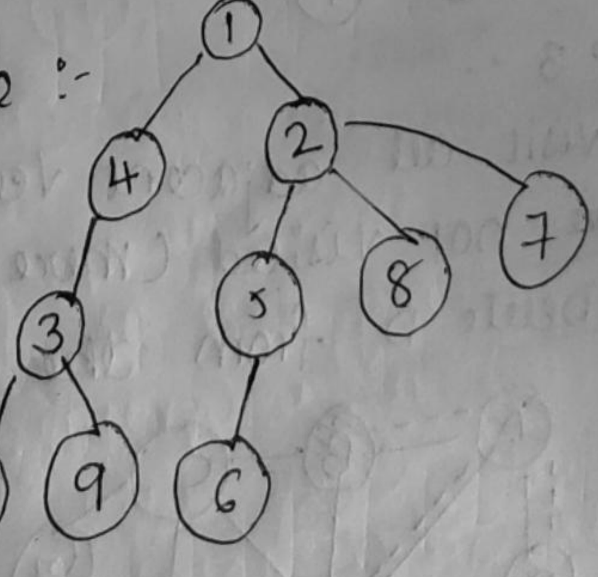
1	4	2	3	5	7	8	9	10	6

								~~6~~

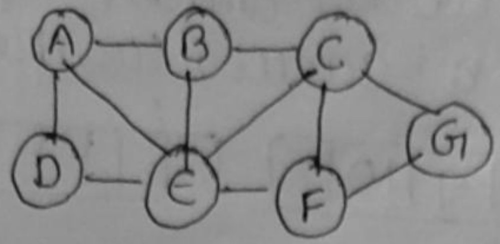
BFS is 1, 4, 2, 3, 5, 7, 8, 9, 10, 6



Spanning tree :-



Example: 2

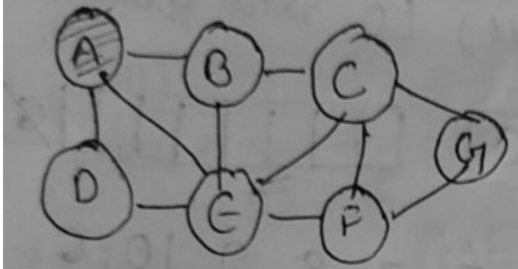


Step 1:

→ Select the vertex A as Starting poin.

→ Insert A in to queue

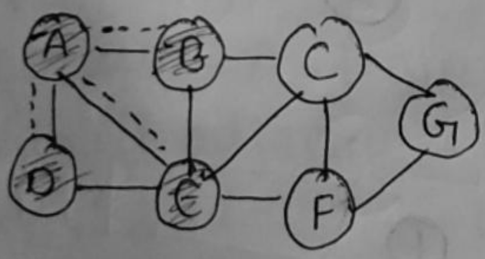
A					



Step 2 :-

→ Visit all the adjacent vertices of A which are not visited (D,E,B)

→ Insert newly visited vertices in to queue & delete A from the queue.

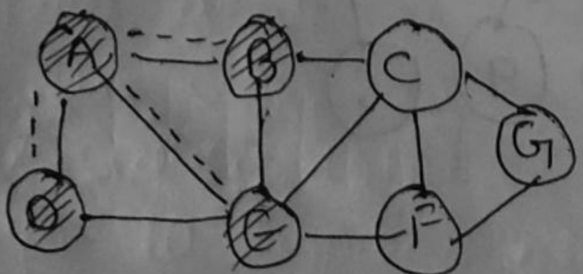


	D	E	B		

Step 3 :-

→ visit all adjacent vertices of ~~E~~ D which are not visited (there is no vertex)

→ Delete D from the queue.

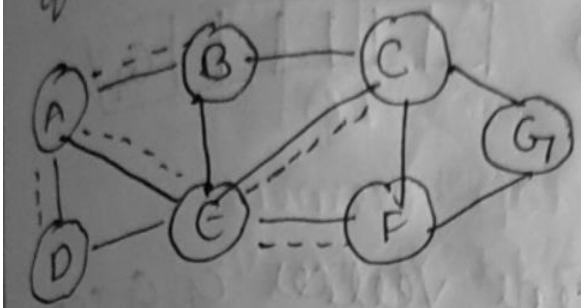


		E	B		

step 4 :-

-> visit all adjacent vertices of E which are not visited (C, F)

-> Insert newly visited vertices in to the queue & Delete E from the queue.

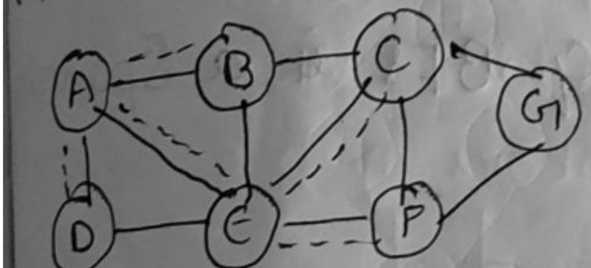


			B	C	F

step 5 :-

visit all adjacent vertices of B which are not visited (there is no vertex)

-> Delete B from the queue.

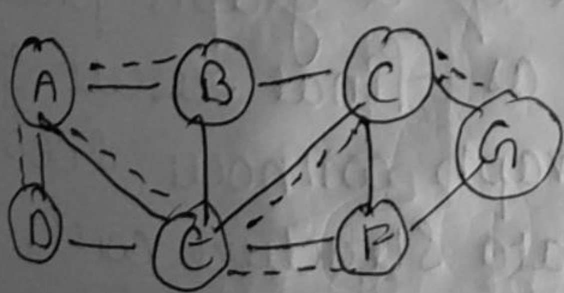


				C	F

step 6 :-

visit all adjacent vertices of C which are not visited (G)

-> Insert newly visited vertex in to the queue & delete C from the queue.

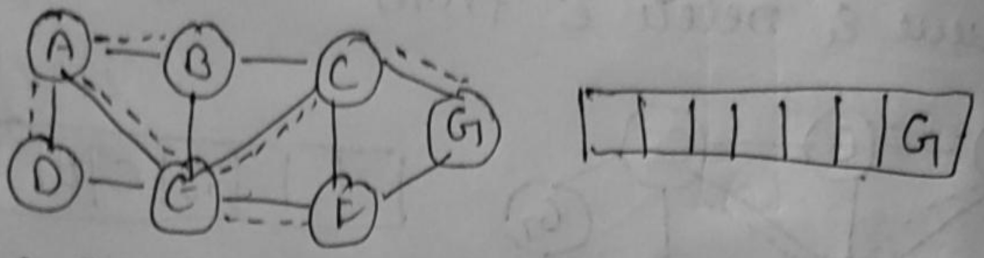


				F	G

Step 7:

→ visit all adjacent vertices of F which are not visited (there is no vertex)

→ Delete F from the queue.



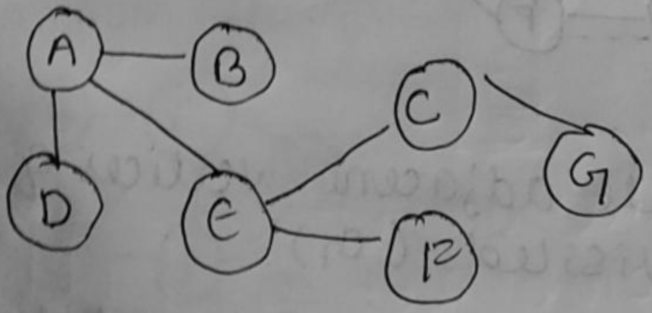
Step 8:

visit all adjacent vertices of G which are not visited (there is no vertex).

→ Delete G from the queue

→ queue become Empty so stop BFS procry

→ Final result of BFS is a Spanning tree.



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 max size of total no. 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 no. of vertices in the graph

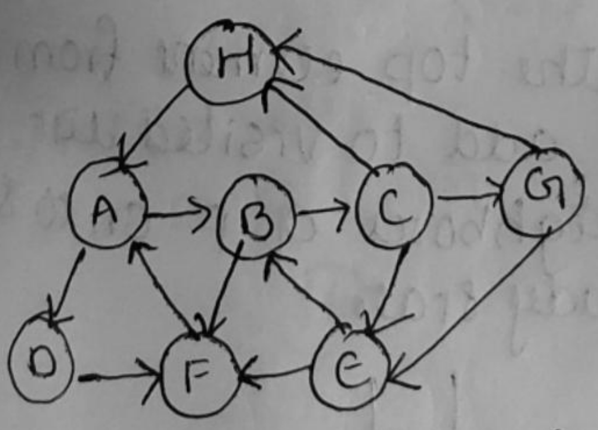
step 2 :- Select any vertex as starting point for traversal. visit that vertex & push it on to the stack.

step 3 - visit any ~~one~~ of the non-visited adjacent vertex & push in to top of the stack.

step 4 :- Now repeat steps 3 & 4 until no vertices are left to visit from the vertex on the stacks top

step 5 : If no vertex is left, go back & pop a vertex from the stack.

step 6 : repeat steps 2, 3, & 4 until the stack is Empty.



Adjacency list

A : B, D

B : C, F

C : E, G, H

G : E, H

E : B, F

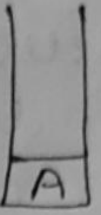
F : A

D : F

H : A.

step 1 → If considering H as a starting vertex. push H on to stack

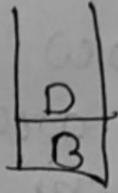
Step 2 :- pop the top element from the stack i-e H, & add it to visited list & push all the neighbors of H onto the stack ~~that~~ are in ready 'B'



visited list

H

Step 3 :- pop the top element from the stack i-e A & add it to visited list & push all the neighbors q A on to stack

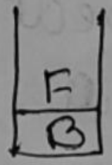


visited list

H A

Step 4 :- pop the top element from the stack i-e D & add to visited list & push all the neighbors of D on to stack

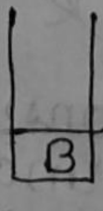
visited list



H A D

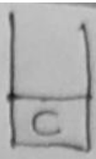
Step 5 :- pop the top element from the stack i.e F & add to visited list & push all the neighbors q F on to stack that are in ready state

visited list



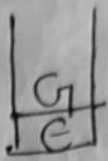
H A D F

Step 6 :- pop the top element from the stack i-e B & ~~print~~ add it to visited list & push all neighbors q B on to the stack.



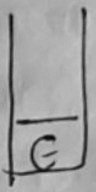
HADFB

step 7 :-
pop the top element from the stack
i.e C & add it to visited list & push
all neigbors of C on to stack



visited list.
HADFC

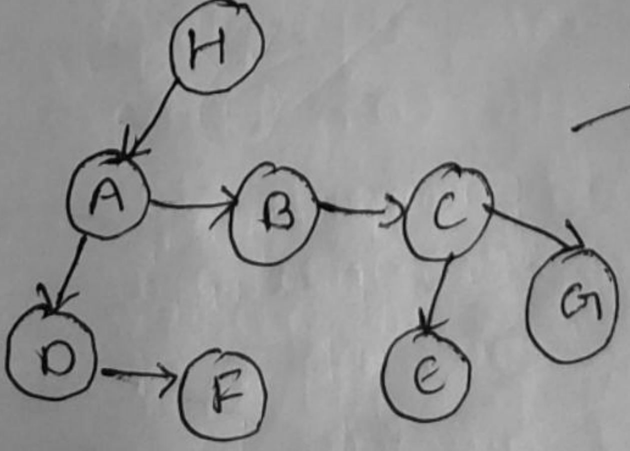
step 8 :- pop the top element from
the stack i.e G & push all the
neighbors of G in to stack



visited List.
H A D F C G

step : 9 :- pop the top element ~~fro~~ & add it
from the stack i.e E
to visited list & push adjacent vertices
of E in to stack.

visited list :-
H A D F C G E



→ Spanning tree