11.3 Write a C++ program to find Minimum Spanning Tree for a given graph.

```
#include <iostream.h>
#include <conio.h>
#define ROW 7
#define COL 7
#define infi 9999 //infi for infinity

class prims
{
int graph[ROW][COL],nodes;
public:
	prims();
	void createGraph();
	void primsAlgo();
};

prims :: prims()
{
for(int i=0;i<ROW;i++)
	for(int j=0;j<COL;j++)
		graph[i][j]=0;
}

void prims :: createGraph()
{
int i,j;
cout<<"Enter Total Nodes : ";
cin>>nodes;
cout<<"\n\nEnter Adjacency Matrix : \n";
for(i=0;i<nodes;i++)
	for(j=0;j<nodes;j++)
		cin>>graph[i][j];

//Assigning infinity to all graph[i][j] where weight is 0
for(i=0;i<nodes;i++)
	for(j=0;j<nodes;j++)
		if(graph[i][j]==0)
			graph[i][j]=infi;

//Printing graph in matrix form
for(i=0;i<nodes;i++)
	for(j=0;j<nodes;j++)
	{
	   cout<<" "<<graph[i][j];
	   if ((j+1)%nodes==0)
	   cout<<endl;
	}
}

void prims :: primsAlgo()
{
int selected[ROW],i,j,ne;
```

```
int false=0,true=1,min,x,y;
for(i=0;i<nodes;i++)
        selected[i]=false;
selected[0]=true;
ne=0;
while(ne < nodes-1)
{
        min=infi;
        for(i=0;i<nodes;i++)
        {
                if(selected[i]==true)
                {
                        for(j=0;j<nodes;j++)
                        {
                                if(selected[j]==false)
                                {
                                        if(min > graph[i][j])
                                        {
                                          min=graph[i][j];
                                          x=i;
                                          y=j;
                                        }
                                }
                        }
                }
        }
        selected[y]=true;
        cout<<"\n"<<x+1<<" --> "<<y+1;
        ne=ne+1;
  }
}

void main(){
   prims MST;
   clrscr();
   cout<<"\nPrims Algorithm to find Minimum Spanning Tree\n";
   MST.createGraph();
   MST.primsAlgo();
   getch();
}
```

11.5 Explain theoretically, the applications of graphs used in networks.

Ans :- There are mainly 2 applications of graphs are :

- Minimum spanning tree,
- The shortest path through a network

1) **Minimum Spanning tree :-**

- A minimum spanning tree is a spanning tree in which the total weight of the edges are guaranteed to be minimum of all possible paths in graph.
- A spanning tree is a tree that contains all the vertices in the graph.
- While creating a spanning tree, following properties must be considered:-
  - Every vertex is included.
  - The total edge weight of spanning tree is the minimum possible weight that includes a path between any 2 vertices.
- The overall **cost** of a network is minimized.

2) **<u>The Shortest Path through a network :-</u>**

- If we are required to find shortest path between two nodes of a network (eg : say between 2 cities in an airline) , then we have to use the “shortest path algorithm”.
- The shortest path algorithm was first proposed by Dijkstra.
- Dijkstra’s algorithm gives the shortest path between any two nodes in a network.
- It can be used for finding costs of the shortest paths between 2 adjacent vertices.
- This algorithm is often used in routing as a subroutine in other graph algorithms , or in GPS Technology. Also it is widely used in network routing protocols (mostly in OSPF) to find shortest route between two places.

11.6 Write algorithms to (university):

d) insertArc

```
algorithm  insertArc      (val   graph         <graphHead pointer>.
                           val   fromKey            <keyType>,
                           val   toKey         <keyType>)
Adds an arc between two vertices.
   Pre       graph is a pointer to a graph head structure
             fromKey is the key of the originating vertex
             toKey is the key of the destination vertex
   Post      Arc added to adjacency list
   Return    +1 if successful
             -1 is memory overflow
             -2 if fromKey not found
             -3 if toKey not found
1 if  (memory full)
      return -1
Locate source vertex
2 fromPtr = graph->first
3 loop   (fromPtr not null AND fromKey >  fromPtr->data.key)
      1 fromPtr      =fromPtr->nextVertex
4 if (fromPtr null OR fromKey not equal fromPtr->data.key)
      1 return -2
Now locate to vertex
5 toPtr          = graph->first
6 loop  (toPtr not null AND toKey  >  toPtr->data.key)
      1  toPtr          = toPtr->nextVertex
7  if   (toPtr null OR toKey not equal toPtr ->data.key)
      1 return  -3
From and to vertices located.  Insert new arc
8 fromPtr->outDegree        = fromPtr->outDegree + 1
  9 toPtr->inDegree         = toPtr->inDegree  + 1
10 allocate  (newPtr)
```

First find source and destination vertex
Change indegree and outdegree
Allocate new and assign its dest^n to dest^{n}PTR
If sorce arc null insert first
Find location
PRE and LOCN
If pre null insert before first arc
Else assign it after PRE
NEW->NEXT = LOCN

11 newPtr->destination = toPtr
12 if (fromPtr->arc null)
Inserting first arc
1 fromPtr->arc = newptr
2 newPtr->nextarc = null
3 return 1
Find insertion point in adjacency (arc) list
13 predptr = null
14 locnPtr = fromPtr->arc
15 loop (locnPtr not null AND toKey >= locnPtr->destination->data.key)
1 predPtr = locnPtr
2 locnPtr = locnPtr->nextArc
16 if (predPtr null)
Insertion before first arc
1 fromPtr->arc = newPtr
17 else
1 predPtr->nextArc = newPtr
18 newPtr->nextArc =locnPtr
19 return 1
end insertArc

e) deleteArc

algorithm deleteArc (val graph <graphHead pointer>.
Val fromKey <keyType>.
Val toKey <keyType>)
Deletes an arc between two vertices.
Pre graph is a pointer to a head structure
fromKey is the key of the originating vertex
toKey is the key of the destination vertex
Post Vertex deleted
Return +1 if successful
-2 if fromKey not found
-3 if toKey not found
1 if (graph->first null)
return -2
Locate source vertex
2 fromVertex = graph->first
3 loop (fromVertex not null AND fromKey > fromVertex->data.key)
1 fromVertex = fromVertex->nextVertex
4 if (fromVertex null or fromKey < fromvertex-data.key)
1 return -2
Locate destination vertex in adjacency list
5 if (fromVertex->arc null)
1 return -3
6 prePtr = null
7 arcPtr = fromVertex->arc
8 loop (arcPtr not null AND toKey > arcPtr->destination->data.key)
1 prePtr = arcPtr
2 arcPtr = arcPtr->nextArc
9 if (arcPtr null or toKey < arcPtr->destination->data.key)

1 return – 3
10 toVertex = arcPtr->destination
fromVertex, toVertex, and arcPtr all located. Delete arc
11 fromVertex->outDegree = fromVertex->outDegree – 1
12 toVertex->inDegree = toVertex->inDegree – 1
13 if (prePtr null)
Deleting first arc
1 fromVertex->arc = arcPtr->nextArc
14 else
1 prePtr->nextArc =arcPtr->nextArc
15 recycle (arcPtr)
16 return 1
end deleteArc

g) shortestPath (Dijkstra)

algorithm shortestPath(graph <pointer to graphHead>)

Determines shortest path from a network vertex to other vertices.

Pre graph is pointer to network

Post Minimum path tree determined

1 if (graph->first null)

1 return

2 vertexPtr=graph->first

3 loop (vertexPtr not null)

Initialize inTree flags & path length.

1 vertexPtr->inTree=false

2 vertexPtr->pathLength=+8

3 edgePtr=vertexPtr->edge

4 loop(edgePtr not null)

1 edgePtr->inTree=false

2 edgePtr=edgePtr->nextEdge

5 vertexPtr=vertexPtr->nextVertex

Now derive minimum path tree

4 vertexPtr=graph->first

5 vertexPtr->inTree=true

```
6 vertexPtr->pathLength=0
7 treeComplete=false
8 loop(not treeComplete)
    1 treeComplete=true
    2 chkVertexPtr=vertexPtr->edge
    3 minEdgePtr=null
    4 pathPtr=null
    5 newPathLen=+8
    6 loop(chkVertexPtr not null)
        Walk through graph checking vertices in tree.
        1 if(chkVertexPtr->inTree true AND chkVertexPtr->outDegree > 0)
            1 edgePtr=chkVertexPtr->edge
            2 minPath=chkVertexPtr->pathLength
            3 minEdge=+8
            4 loop(edgePtr not null)
                Locate smallest path from this vertex
                1 if(edgePtr->destination->inTree false)
                    1treeComplete=false
                    2 if(edge->weight < minEdge)
                        1 minEdge = edgePtr->weight
                        2 minEdgePtr = edgePtr
                2 edgePtr = edge->nextEdge
            Test for shortest path
            5 if(minPath + minEdge < newPathLen)
                1 newPathLen = minPath + minEdge
                2 pathPtr = minEdgePtr
        2 chkVertexPtr = chkVertexPtr->nextVertex
    7 if(pathPtr not null)
        Found edge to insert into tree.
```

1 pathPtr->inTree=true

2 pathPtr->destination->inTree=true

3 pathPtr->destination->pathLength=newPathLen

9 return

end shortestPath

h) BreadthFirstTraversal

algorithm breadthfirst (val graph <graphHead pointer>)

Process the keys of the graph in breadth-first order.

Pre graph is a pointer to a graph head structure

Post vertices processed

1.if(graph->first null)

1.return

First set all processed flags to not processed

Flag: 0---not processed, 1---enqueued, 2---processed

2.queue=createQueue

3.walkPtr=graph->first

4.loop (walkPtr not null)

1.walkPtr->processed=0

2.walkPtr=walkPtr->nextVertex

Process each vertex in vertex List

5.walkPtr=graph->first

6.loop(walkPtr not null)

1.If(walkPtr->processed <2)

1.If(walkPtr->processed <1)

Enqueue and set processed flag to queued(1)

1.enqueue(queue,walkPtr)

2.walkPtr->processed=1

Now Process descendents of vertex at queue front

2. loop(not emptyQueue(queue))

1.dequeue(queue,vertexPtr)

Process vertex and flag as processed

2.process(vertexPtr)

3.vertexPtr->processed=2

Enqueue all vertices from adjacency list

4.arcPtr=vertexPtr->arc

5.loop(arcPtr not null)

1.toPtr=arcPtr->destination

2.if(toPtr->processed is 0)

1.enqueue(queue,toPtr)

2.toPtr->processed=1

4.arcPtr=arcPtr->nextArc

2. walkPtr=walkPtr->nextVertex

7. destroyQueue(queue)

8. return

end breadthFirst

i) DepthFirstTraversal

algorithm depth-first (val grah<metadata>)
Process the keys of the graph in depth-first order.
Pre: grah os a pointer to a graph head structure.
Post: vertices " processed"
1. If (empty graph)
   1. Return
   Set processedflag to not processed

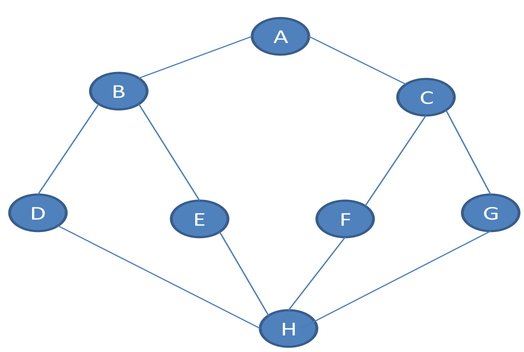
   2. Walker = graph-first
   3. Loop (walker)
      1. Walker ->processed =0
      2. walker = walker ->nextvertvertex
   4. End loop
      Process each each vertex in list
   5. Create stack(stack)
   6. Walkptr = graph-first
   7. Loop (walker not NULL)
      1. If (walkptr -> processed <2)

```
                    Push and set flag to stack
                    1.  pushstack(stack, walkptr)
                    2. walkptr ->processed = 1
              2.  End if
              Process vertex at stack top
          7.1. 3 loop (not empty(stack))
              1. popstack (stack, vertexptr)
              2. process(vertexptr -> dataptr)
              3. vertexptr -> processed =2
          Push all vertices from adjacency list
              4. arcwalker = vertex -> arc
              5. loop(arcwalkptr not NULL)
                    1. vertToptr  = arcwalkptr -> destination
                    2. if (vertTo ptr -> processed is 0)
                          1. pushstack(stack, vertexToptr)
                          2. vertToptr -> processed =1
                    3. End if
                    4. arcwalker = arcwalkptr -> nextarc
              6. End loop
              7.1.4. End loop
            7.2 End if
          7.3. walkptr = walkptr ->nextvertex
          8. End loop
          9. destroyStack(stack)
          10. return
end depth-first
```

11.vii. Solve the following university questions :

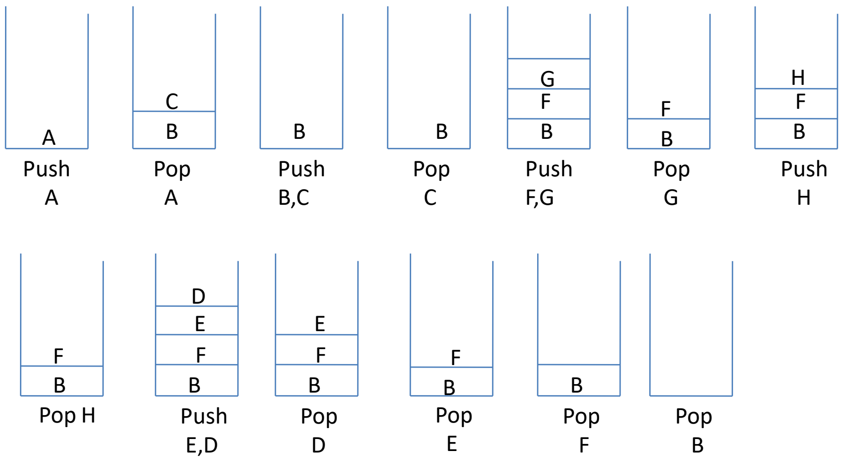
h) D2009-Q6 b)

Give DFS and BFS traversal of the graph shown below. Dec. 2009



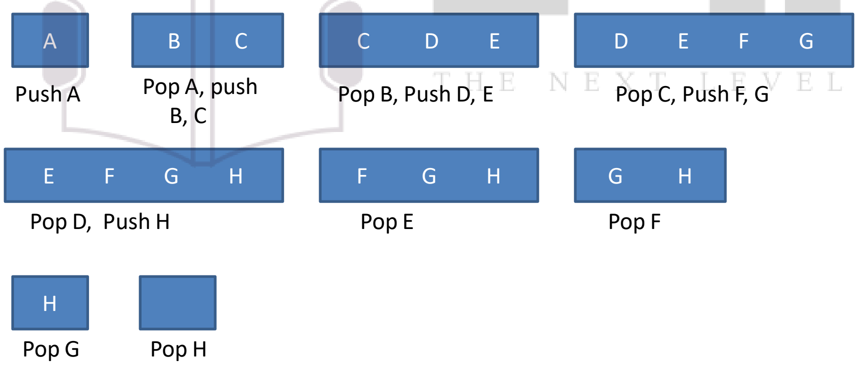
Solution:

DFS:



Depth-first Traversal.
A C G H D E F B

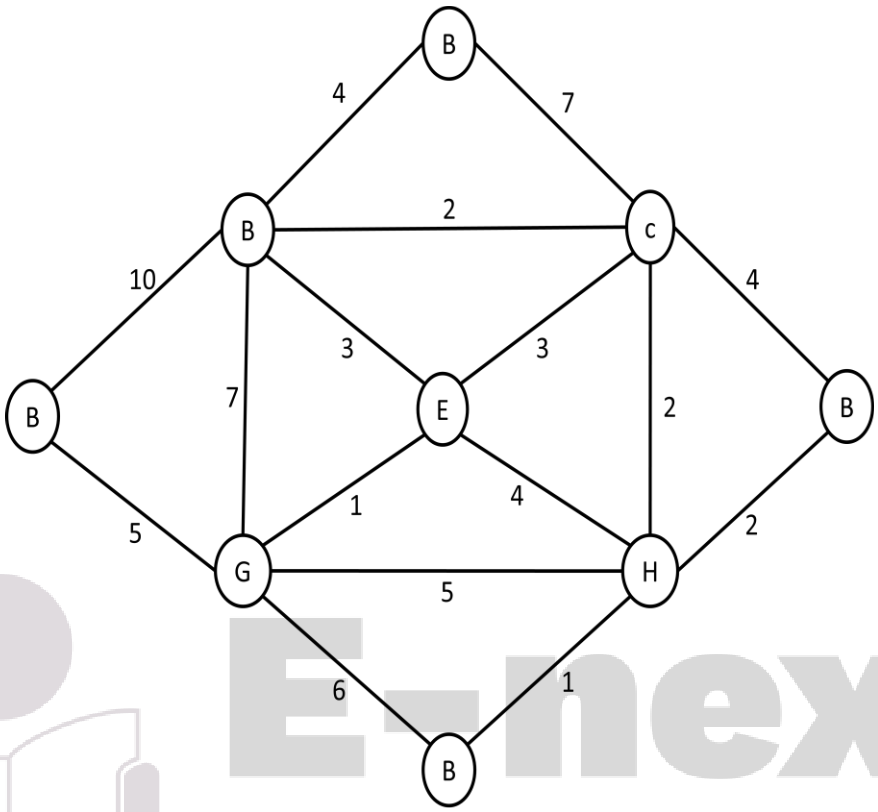
BFS:



Breadth-first trversal:
A B C D E F G H

i) M2009-Q6 b) ii) and j) D2008-Q6 b) ii)

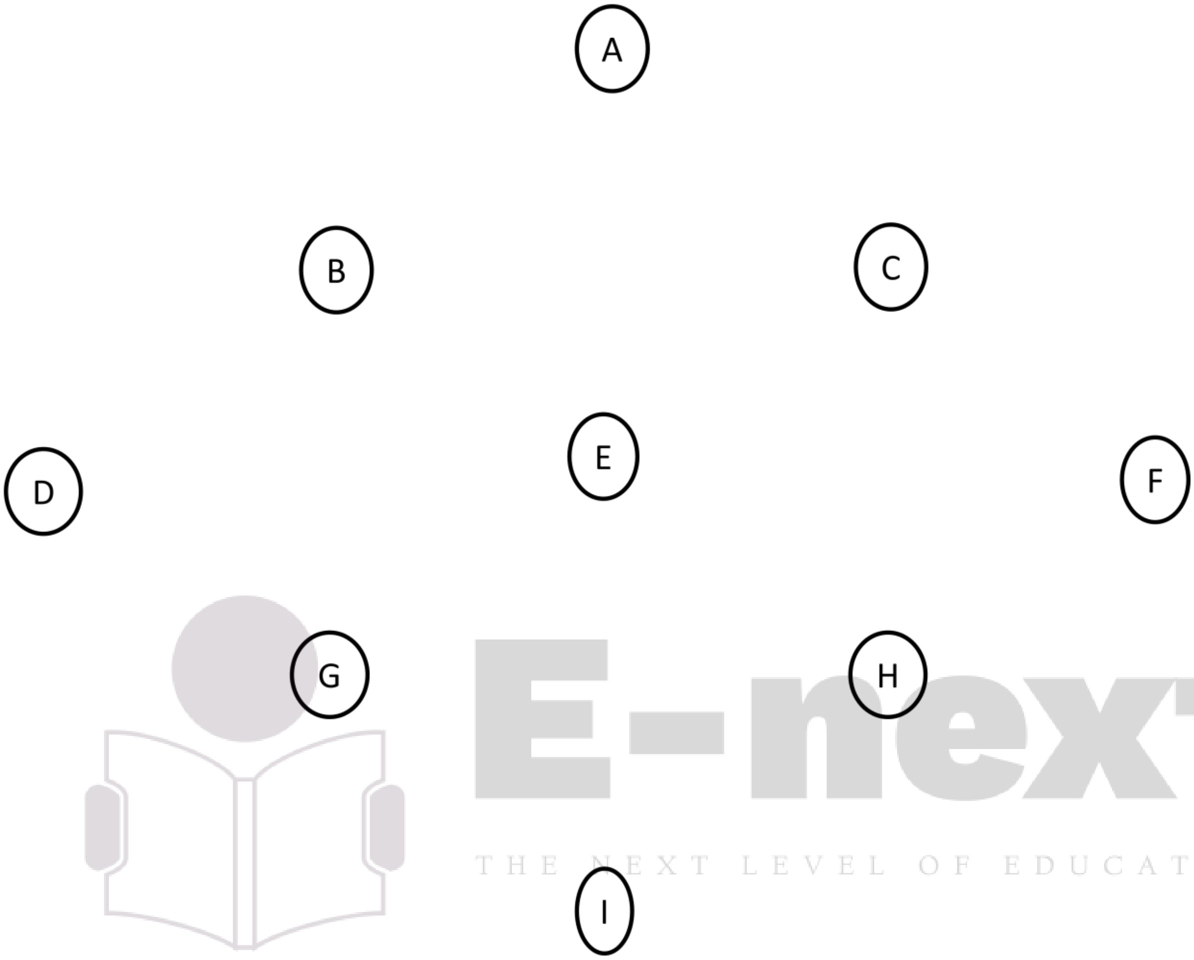
Determine the minimum spanning tree of the following graph using Kruskal's algorithm.



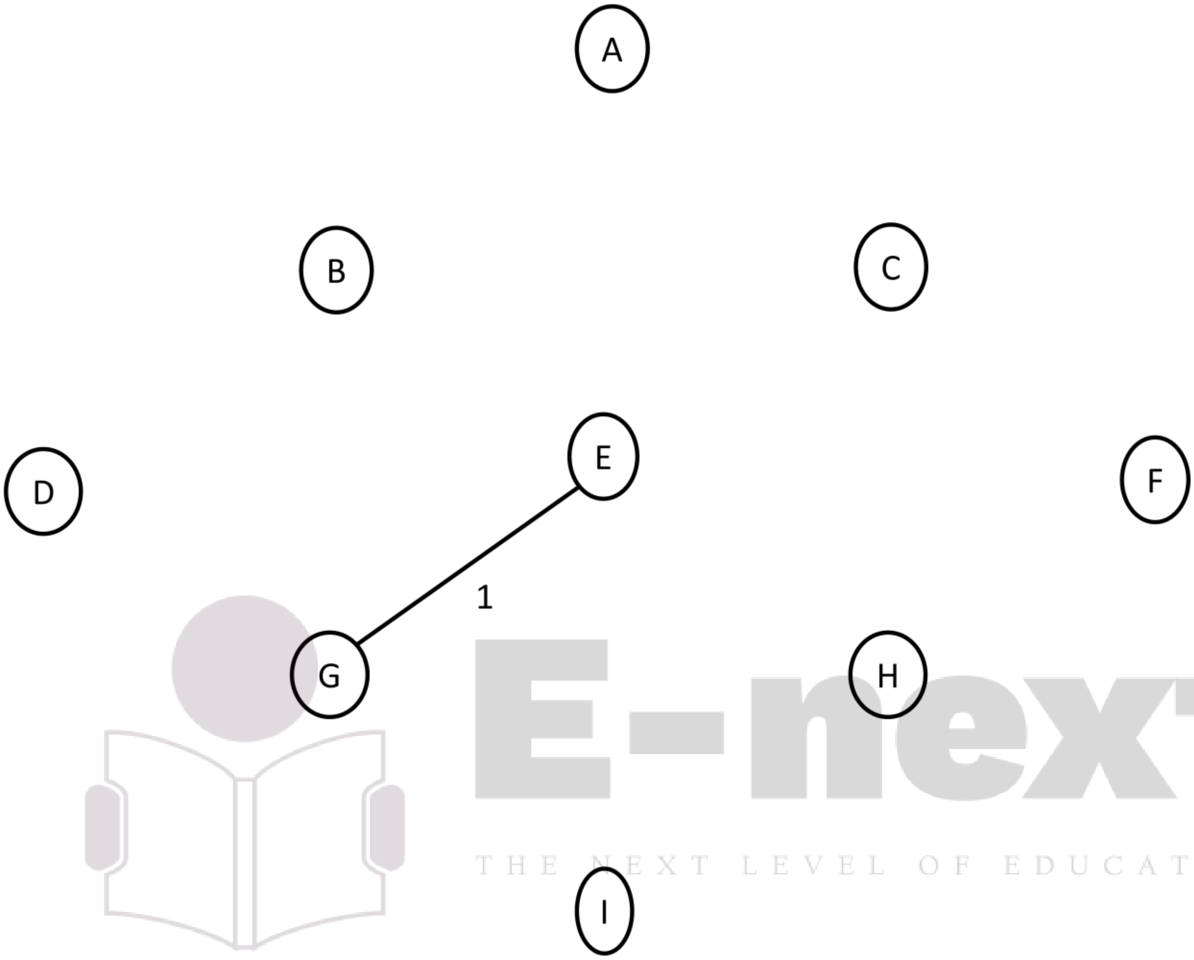
List of Edges in Ascending of Their Weights		
EDGE	WEIGHT	Allowed / Disallowed
E-G	1	Yes
H-I	1	Yes
B-C	2	Yes
C-H	2	Yes
F-H	2	Yes
B-E	3	Yes

C-E	3	No
A-B	4	Yes
C-F	4	No
E-H	4	No
G-H	5	No
D-G	5	Yes
G-I	6	No
A-C	7	No
B-G	7	No
B-D	10	No

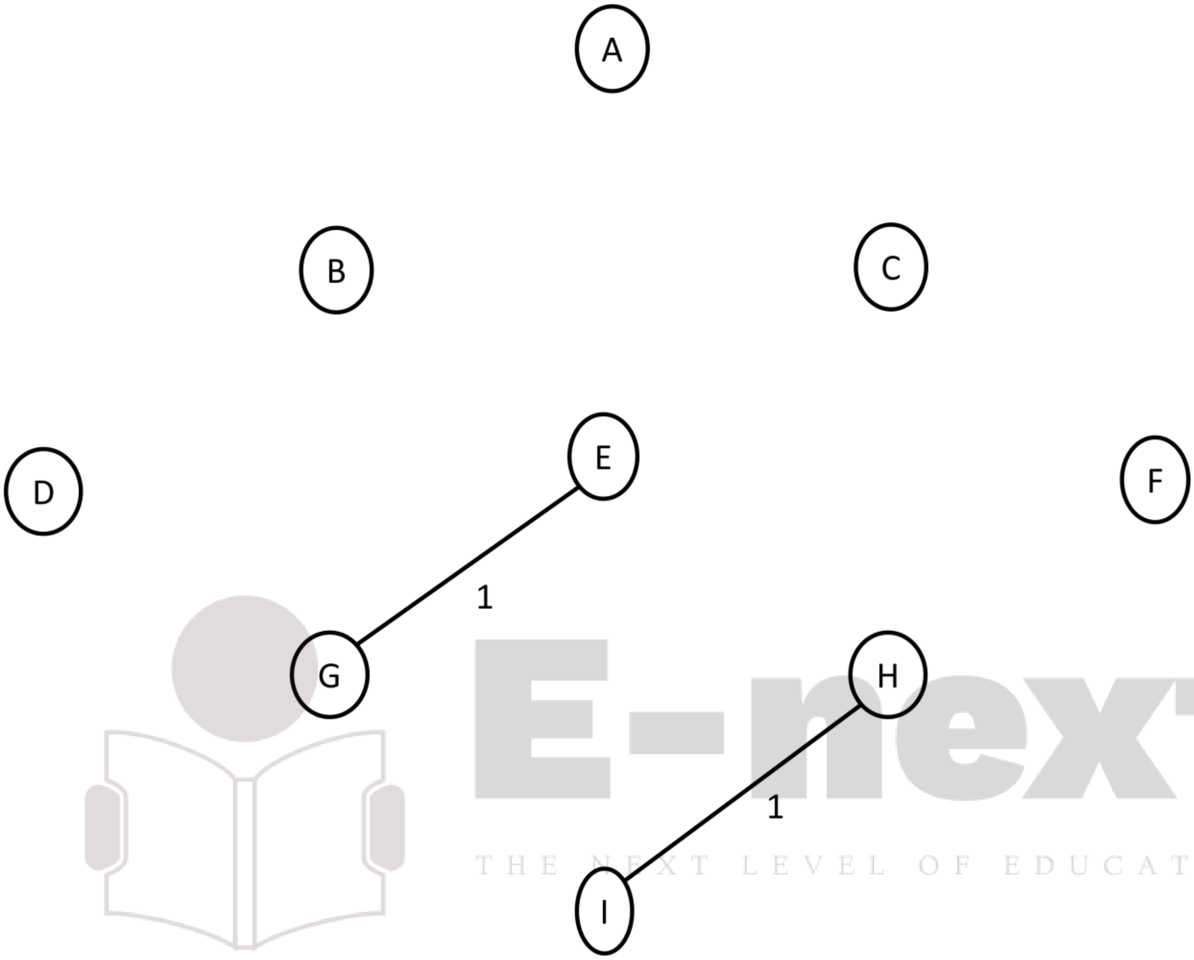
Step-1 : Adding all vertices



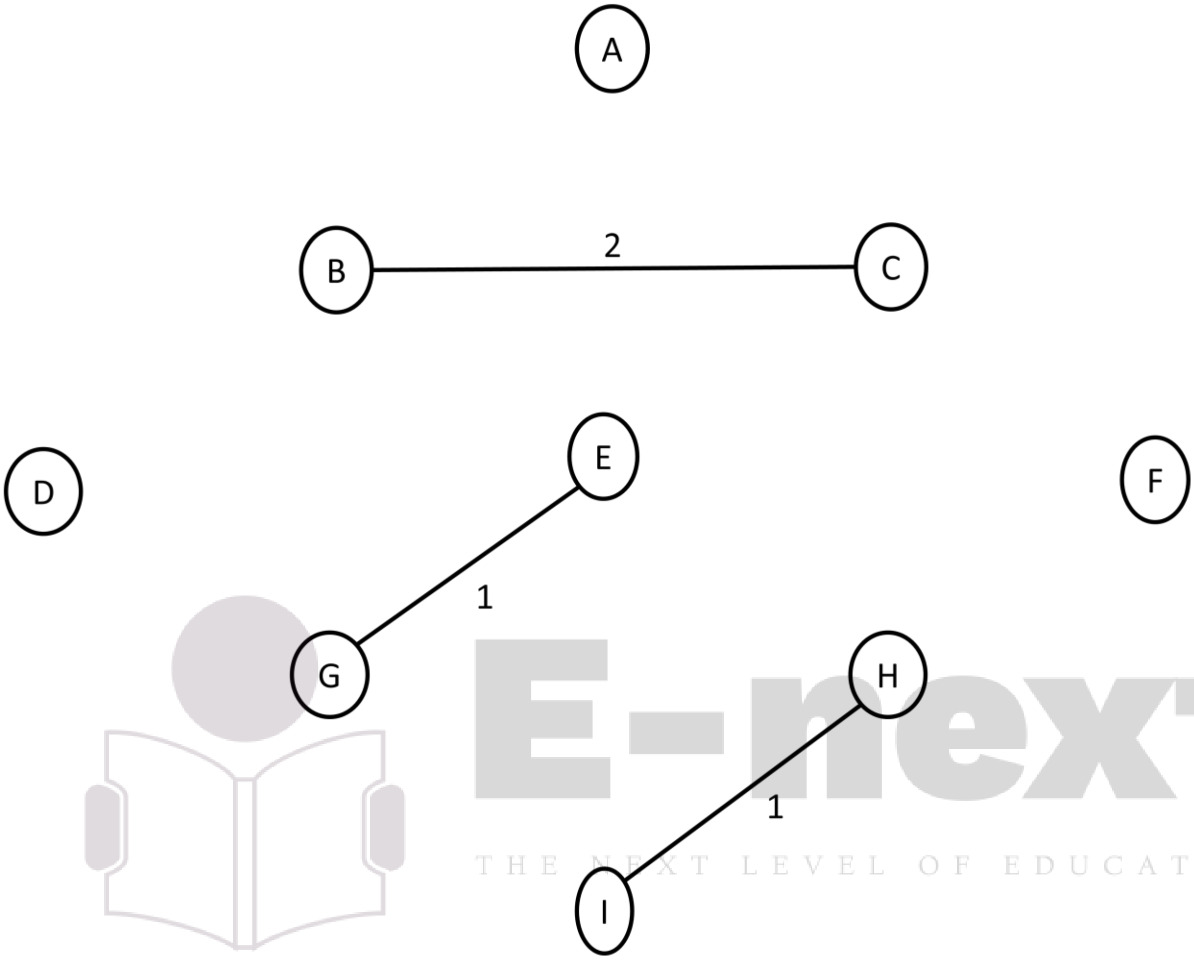
Step-2 : Adding edge E-G



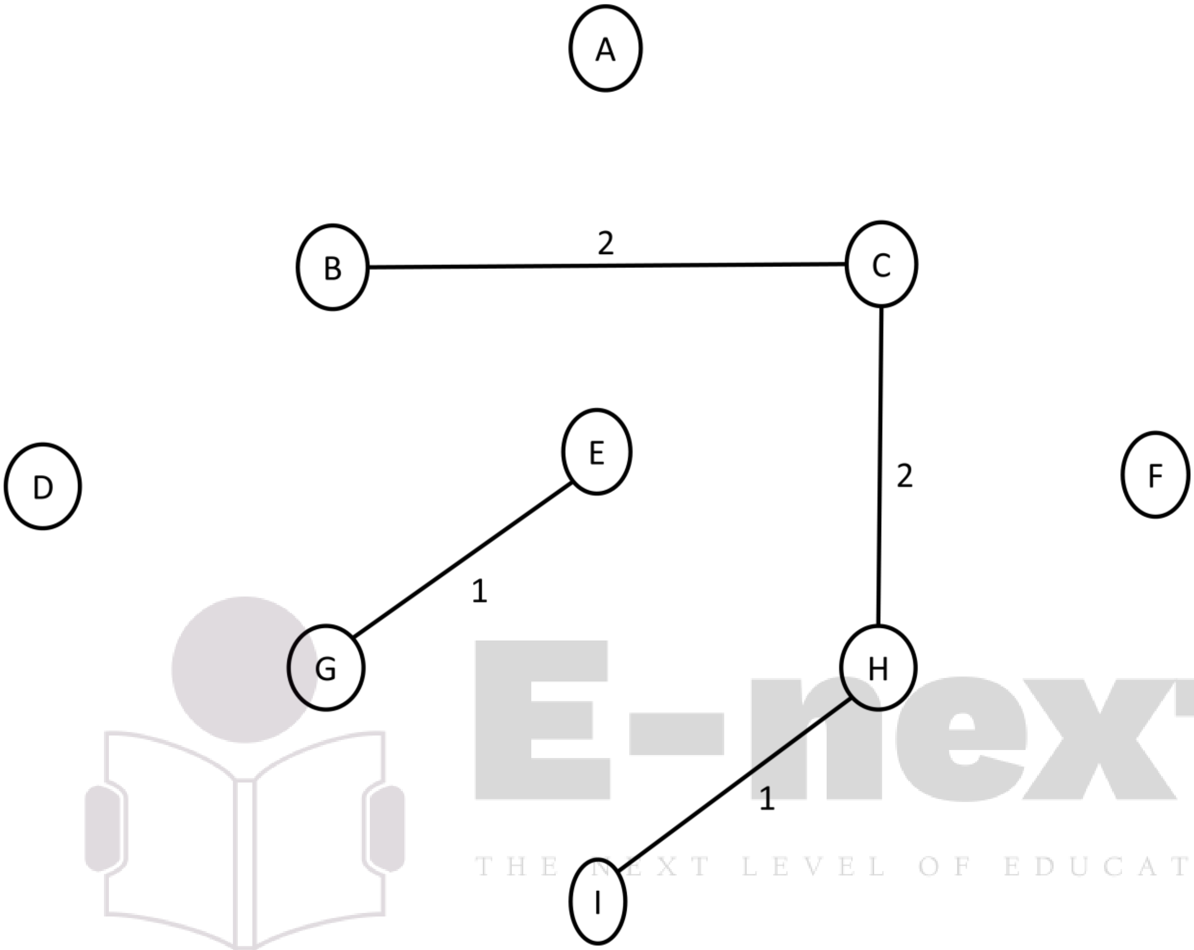
Step-3 : Adding Edge H-I



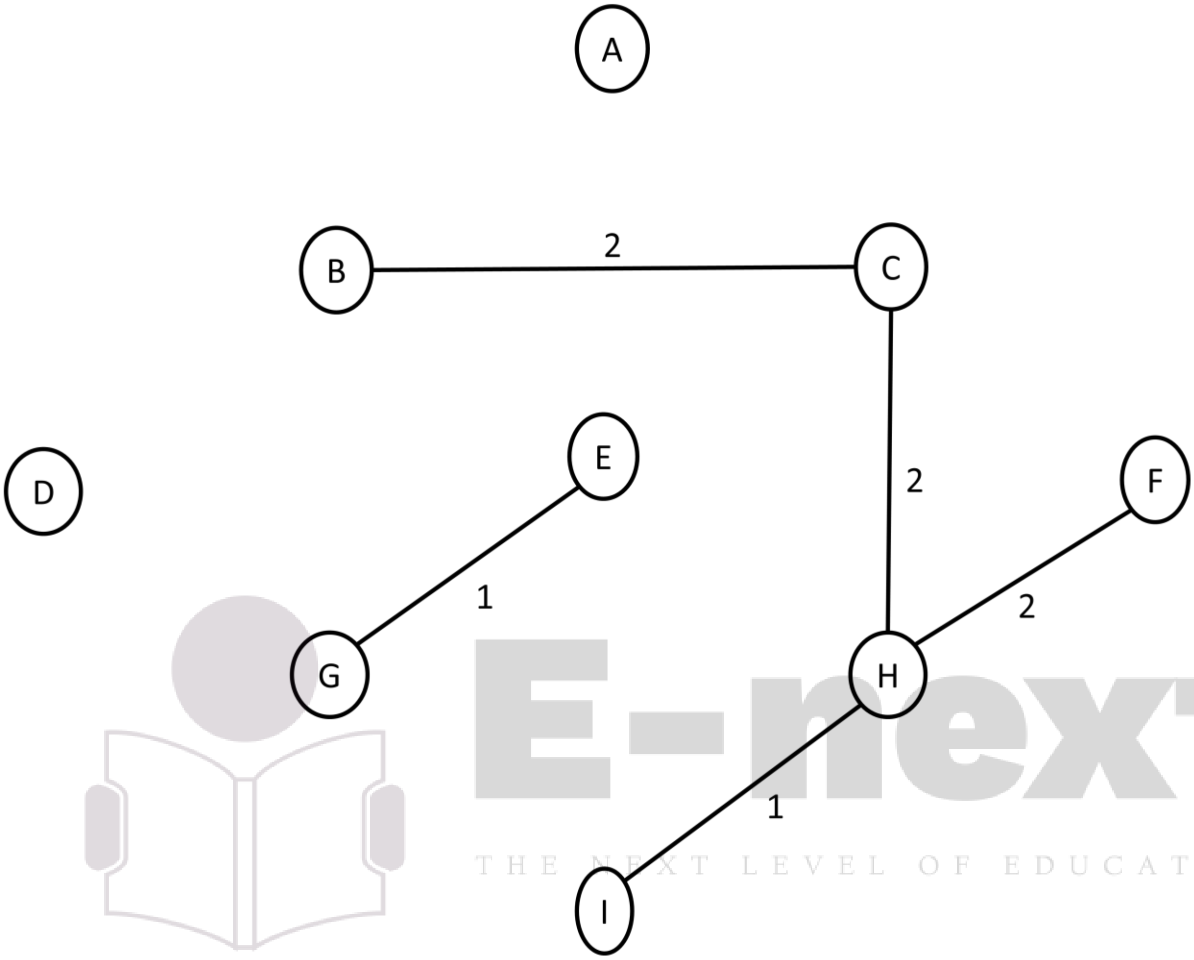
Step-4 : Adding Edge B-C



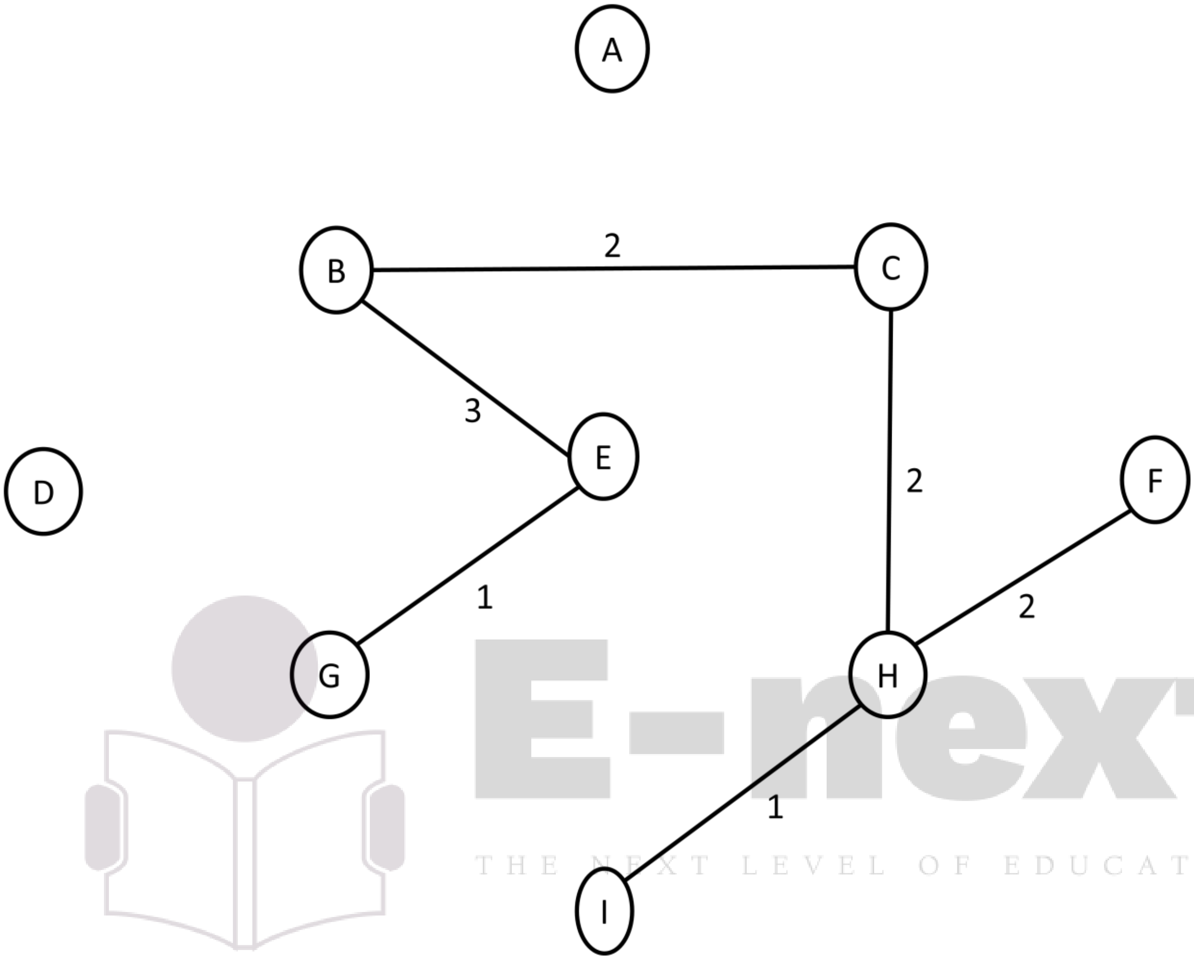
Step-5 : Adding Edge C-H



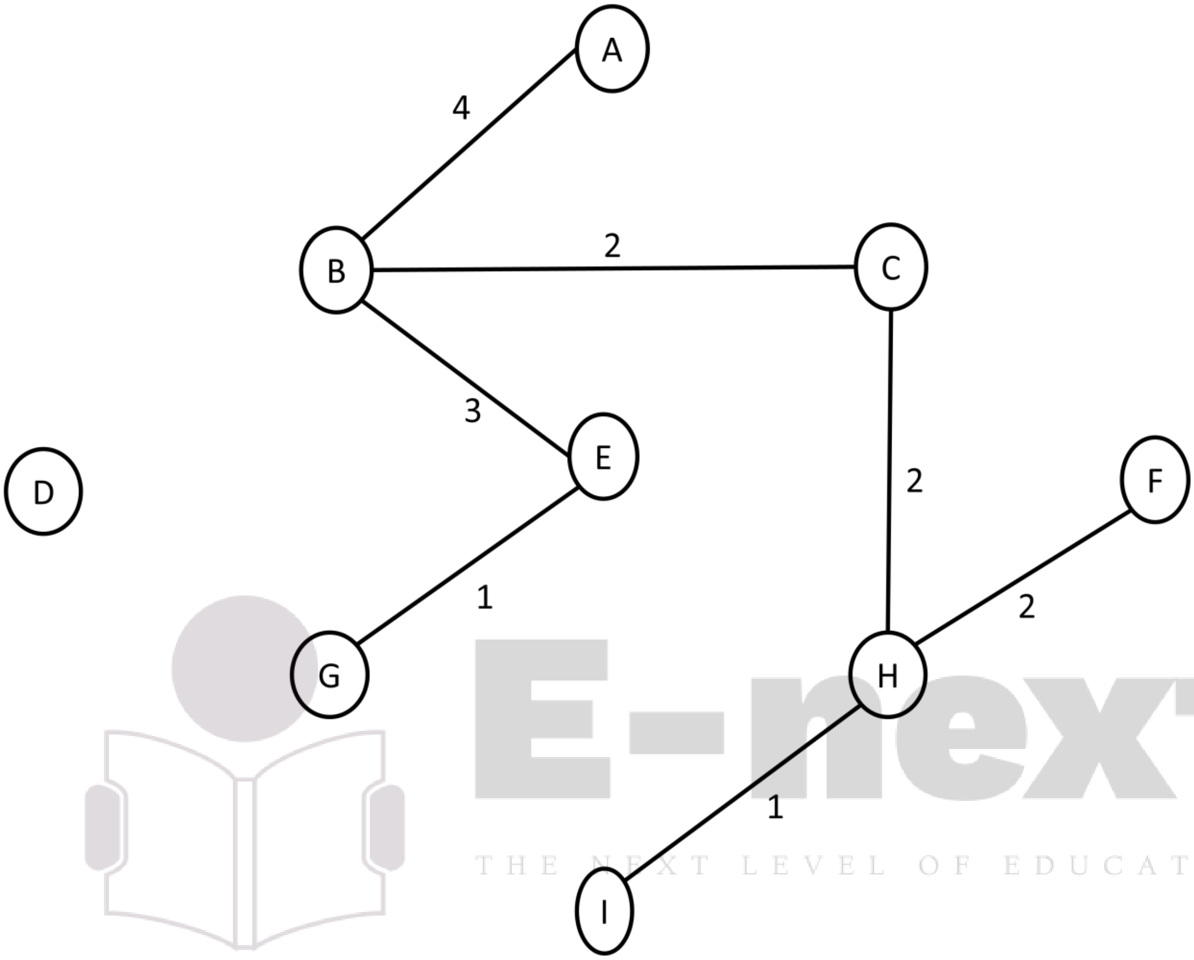
Step-6 : Adding Edge F-H



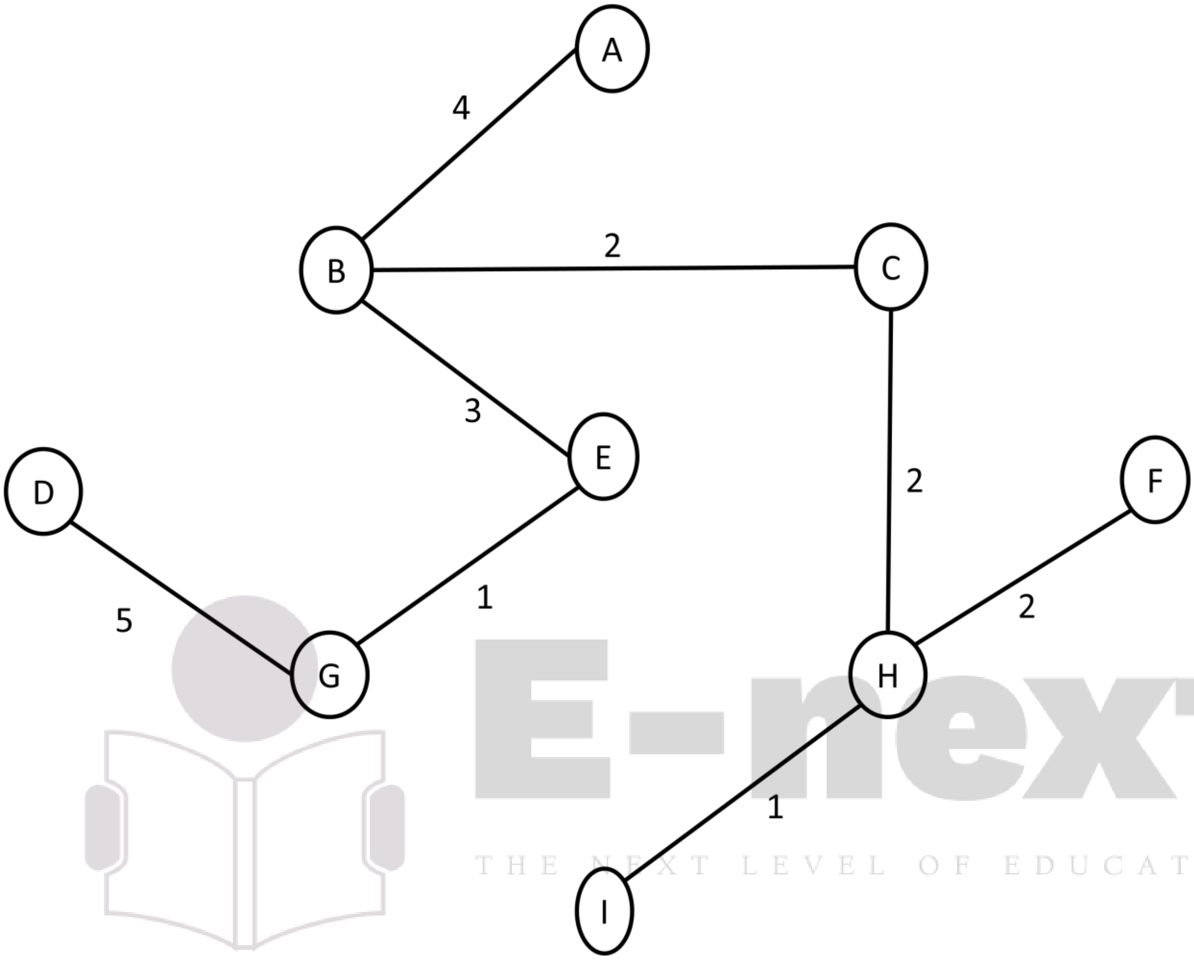
Step-7 : Adding Edge B-E



Step-8 : Adding Edge A-B

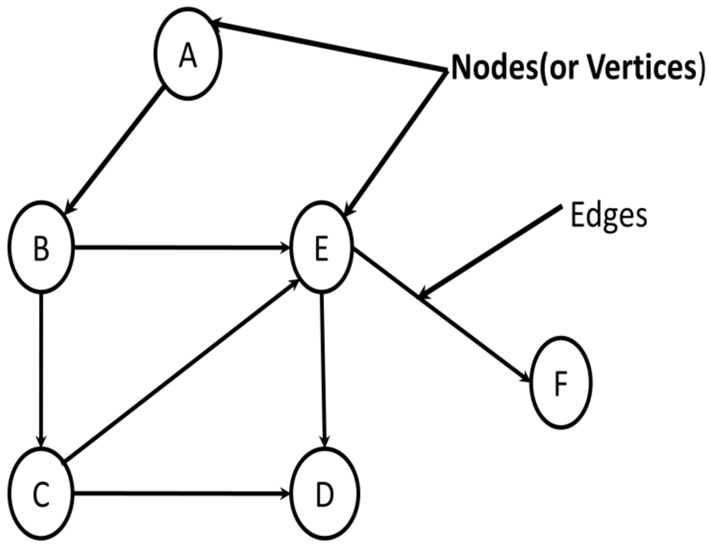


Step-9 : Adding Edge D-G



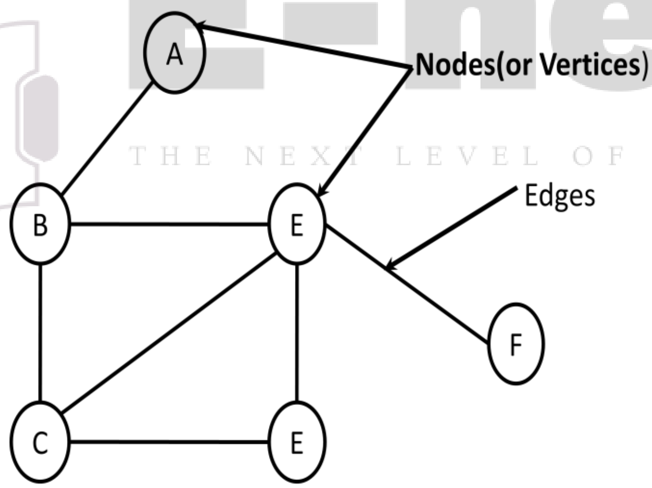
k) Year-2008 (May) Q1 (B) 5 Marks
(i) Directed and Undirected graph

Ans:-
Directed graph
A directed graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph is sometimes called a digraph or a directed network. In contrast, a graph where the edges are bidirectional is called an undirected graph.



A directed graph with 6 vertices (or nodes) and 7 edges.

Undirected graph
An undirected graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are bidirectional. An undirected graph is sometimes called an undirected network. In contrast, a graph where the edges point in a direction is called a directed graph.



An undirected graph with 6 vertices (or nodes) and 7 edges.

I) Year-2008 (May) Q7 (A) 10 Marks
Explain Warshall's algoritham with a suitable example.

Warshall's Algorithm :-

Warshall's algorithm is an efficient method for computing the transitive closure of a relation.Warshall's algorithm takes as input the matrix M_R. representing the relation *R*.**;** and outputs the matrix $M_R.^*$ of the relation $R.^*$ **;** the transitive closure of *R.*

```
Warshall(M_R: n * n 0-1 matrix)
W = M_R (W = [w_ij] )
 for(k=1 to n ) {
     for(i=1 to n) {
          for(j=1 to n) {
             w_ij = w_ij V (w_ik ^ w_kj)
         }
       }
    }
   Return W
```

Let's examine the **algorithm** closely. When the inner *for* loop is being executed, the only value which is changing is j . Notice that the value of w_{kj} does not depend on j.Thus, during each iteration of the inner loop, w_{kj} is constant. If $w_{kj}=0$, then

$$w_{ij} = w_{ij} \vee (w_{ik} \wedge w_{kj}) = w_{ij} \vee (0 \wedge w_{kj})\ w_{ij} \vee (0) = w_{ij},$$

and if $w_{ik} =1$, then

$$w_{ij} = w_{ij} \vee (w_{ik} \wedge w_{kj}) = w_{ij} \vee (1 \wedge w_{kj}) = w_{ij} \vee w_{kj},$$

for each value of J.Thus, the values of w_{ij} remain unchanged if $w_{ik} =0$,and become $w_{ij} \vee w_{ik}$ if $w_{ik} = 1$. It is this observation which leads to the second algorithm. Notice that when $w_{ik} = 1$, we are simply replacing the i th row of the matrix with OR of the i th and k th rows of the matrix.

Example: The matrix $W_0 = M_R$ below is the matrix representation for a relation R. Find the matrix representation M_R* of R* ,the transitive closure of R.

$$M_R = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Solution: We know that $W_0 = MR.$ To compute W_0 , we notice that in the first column of W_0 ,there are "1" s in rows 1 and 4. Thus, we replace rows 1 and 4 with the OR of themselves and row 1.

We obtain

$$W_1 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

To compute W_2, we notice that in the second column of W_1, there is a "1" in row 3. Thus, we replace row 3 with the OR of rows 3 and 2, obtaining.

$$W_2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

To compute W_3, we notice that in the third column of W_2, there is a "1" in every row. Thus, we replace each row with the OR of itself and row 3, obtaining.

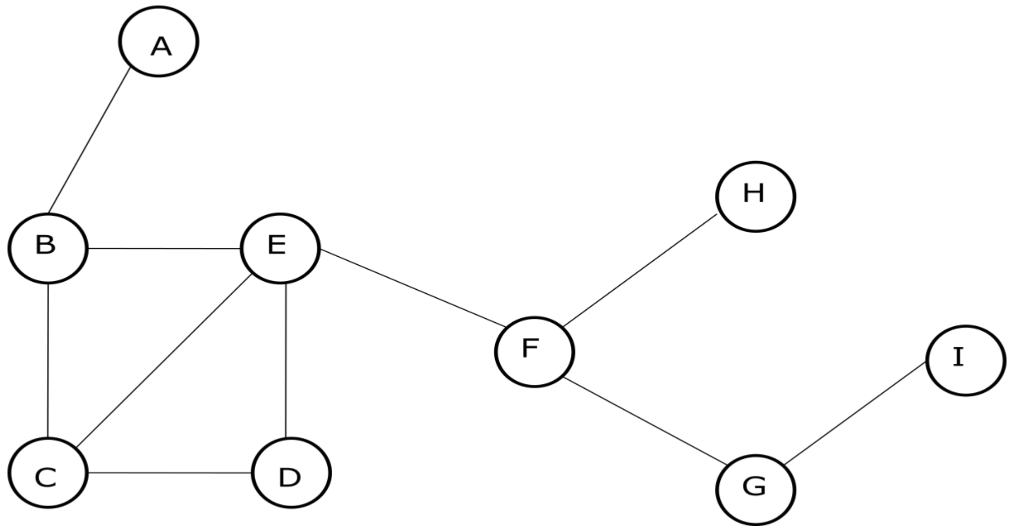
$$W_3 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

To compute W_4, we notice that in the fourth column of W_3, there is a "1" in every row. Thus, we replace each row with the OR of itself and row 4, obtaining.

$$W_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} = M_{R^*}$$

m) M2007.Q6.a.)

Write the algorithm for breadth first traversal of graph & give the Breadth First traversal of graph in figure given below : (10 marks)



Ans : **BREADTH-FIRST TRAVARSAL**

In the breadth-first traversal of a graph, all adjacent vertices of a vertex before going to next level.

Algorithm:

algorithm breadthfirst (val graph <graphHead pointer>)

Process the keys of the graph in breadth-first order.

Pre graph is a pointer to a graph head structure

Post vertices processed

1.if(graph->first null)

1.return

First set all processed flags to not processed

Flag: 0---not processed, 1---enqueued, 2---processed

2.queue=createQueue

3.walkPtr=graph->first

4.loop (walkPtr not null)

1.walkPtr->processed=0

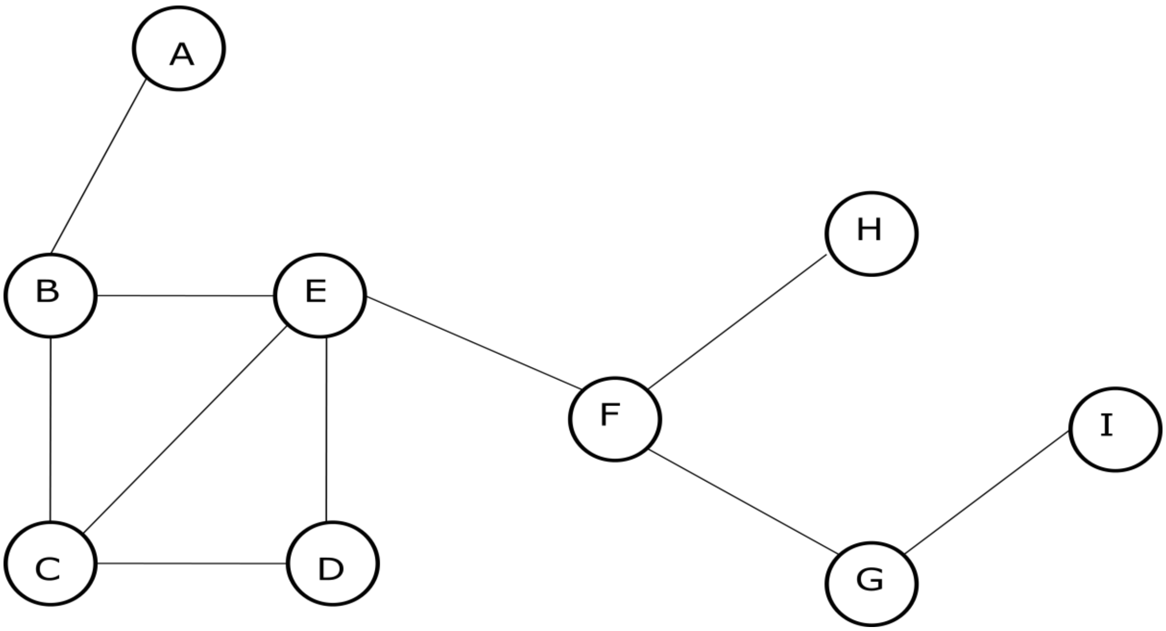
2.walkPtr=walkPtr->nextVertex

Process each vertex in vertex List

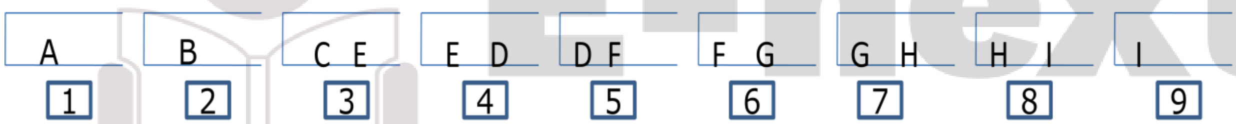
5.walkPtr=graph->first

```
6.loop(walkPtr not null)
    1.If(walkPtr->processed <2)
        1.If(walkPtr->processed <1)
            Enqueue and set processed flag to queued(1)
            1.enqueue(queue,walkPtr)
            2.walkPtr->processed=1
        Now Process descendents of vertex at queue front
        2. loop(not emptyQueue(queue))
            1.dequeue(queue,vertexPtr)
            Process vertex and flag as processed
            2.process(vertexPtr)
            3.vertexPtr->processed=2
            Enqueue all vertices from adjacency list
            4.arcPtr=vertexPtr->arc
            5.loop(arcPtr not null)
                1.toPtr=arcPtr->destination
                2.if(toPtr->processed is 0)
                    1.enqueue(queue,toPtr)
                    2.toPtr->processed=1
                4.arcPtr=arcPtr->nextArc
    2. walkPtr=walkPtr->nextVertex
7. destroyQueue(queue)
8. return
end breadthFirst
```

Now, in the example,



GRAPH



Queue

The breadth first traversal for above graph is **ABCEDFGHI**.

n) Year-2006 (Nov) Q4 (B) 8 Marks
Find the Minimum Spanning Tree of the graph given below.

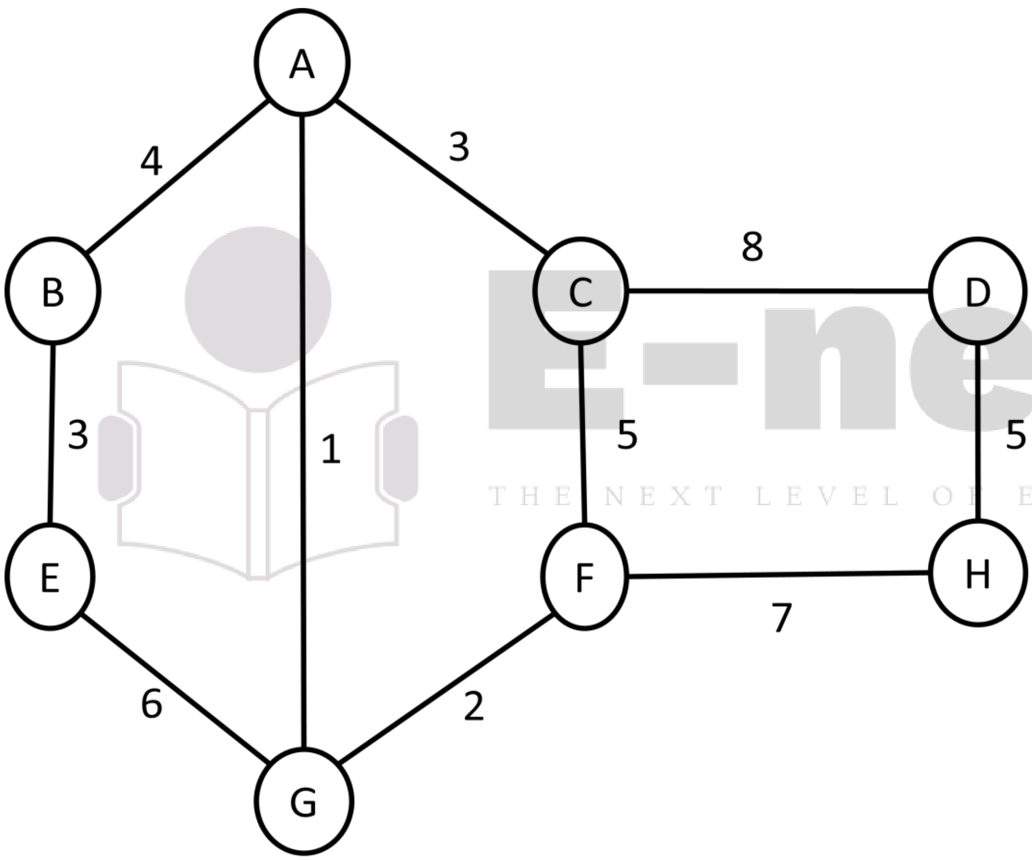
Ans:
Minimum Spanning Tree :-
Definitions:-

- A tree is a connected graph without cycles.
- A subgraph that spans (reaches out to) all vertices of a graph is called a spanning subgraph.

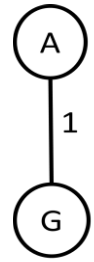
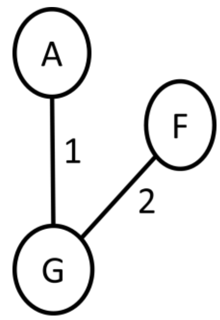
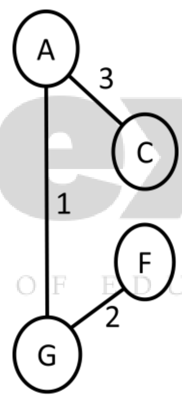
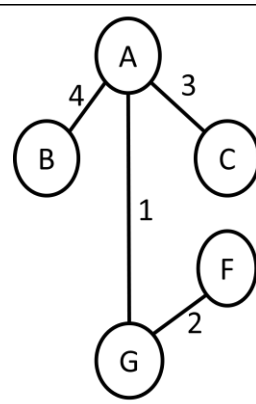
- A subgraph that is a tree and that spans (reaches out to) all vertices of the original graph is called a spanning tree.
- Among all the spanning trees of a weighted and connected graph, the one (possibly more) with the least total weight is called a minimum spanning tree (MST).

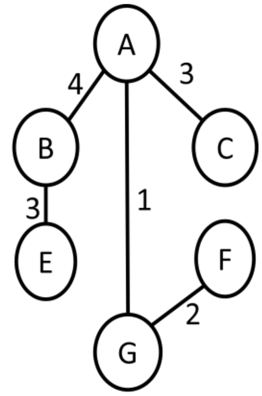
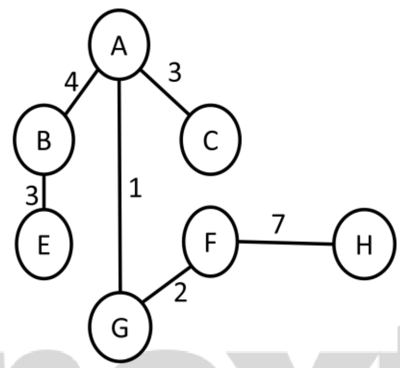
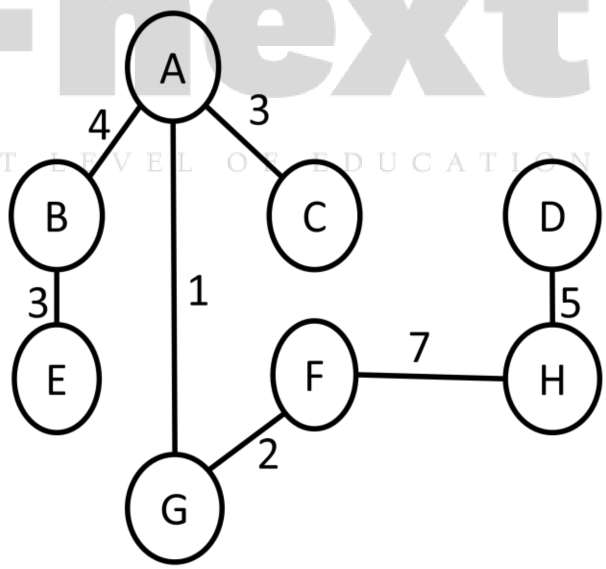
Properties of Trees:-

- A graph is a tree if and only if there is one and only one path joining any two of its vertices.
- A connected graph is a tree if and only if every one of its edges is a bridge.
- A connected graph is a tree if and only if it has N vertices and N; 1 edges.



Set Of Vertices in the Minimal Spanning Tree	Edges which have exactly one end belonging to the partial Minimal	The Edge Chosen	The new partial Minimal Spanning Tree

	Spanning Tree		
A	(A,B) (A,C) (A,G)	(A,G)	
(A,G)	(A,B) (A,C) (G,F) (G,E)	(G,F)	
(A,G,F)	(A,B) (A,C) (G,E) (F,C) (F,H)	(A,C)	
(A,G,F,C)	(A,B) (G,E) (F,H) (C,D)	(A,B)	

(A,G,F,C,B)	(G,E) (F,H) (C,D) (B,E)	(B,E)	
(A,G,F,C,B,E)	(F,H) (C,D)	(F,H)	
(A,G,F,C,B,E,H)	(C,D)(D,H)	(D,H)	
TOTAL MINIMAL COST = 25			