1. Linear DS - Array, ...

2. Non-Linear DS - Tree, graph ...de in linear fashion finally

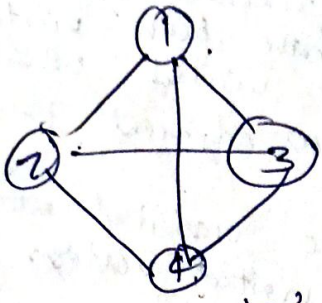
(A tree is a graph but a graph is not necessarily a tree

Graph

Basic Terminology :- Directed Graph / Digraph, Multigraph

Graph Representation :-

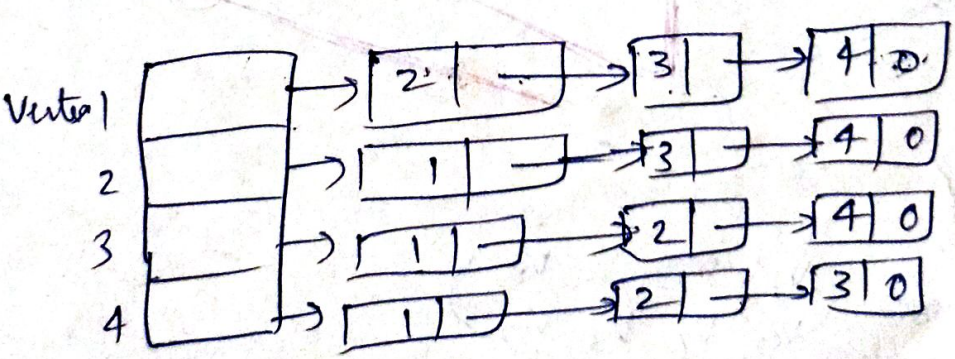
Adjacency Matrix & Adjacency List



Graph G1

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

Adjacency matrix of G1



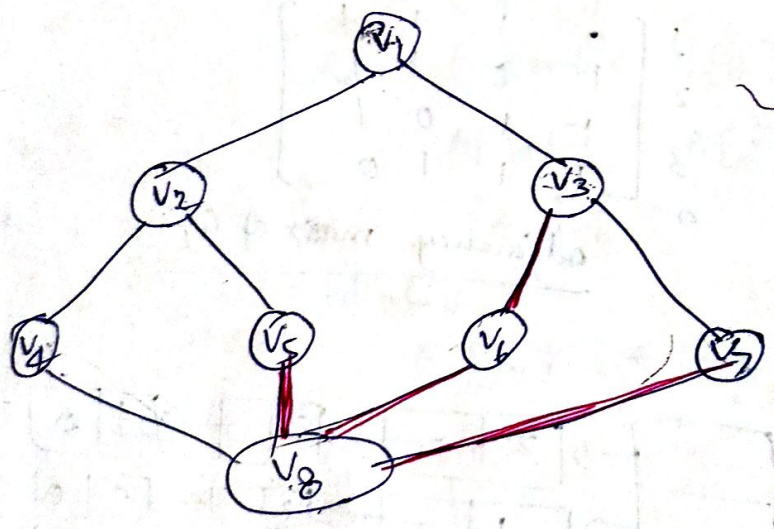
Adjacency List of G1

Traversal

Tree — Preorder, Inorder, Postorder
Graph (undirected) — Depth first search, Breadth first search

DFS:-

→ First of all, the start vertex v is visited.
→ Then an unvisited vertex w adjacent to v is selected & a DFS from w is initiated.
→ When a vertex u is reached such that all its adjacent vertices have been visited, back up to the last vertex visited which has an unvisited vertex w adjacent to it & initiate a DFS from w.
→ search terminates when no unvisited vertex is reached from any of the visited vertices



DFS: $(V_1\ V_2\ V_4\ V_8\ V_5\ V_6\ V_3\ V_7)$

$V_1\ V_2\ V_4\ V_8\ V_5\ V_6\ V_3\ V_7$

ansive Algo:-

```
Procedure DFS(v)
// Global variables [ G = (V, E)
                      VISITED(n)
VISITED(v) ← 1
for each vertex w adjacent to v do
    if VISITED(w) = 0 then call DFS(w)
end
end DFS
```

an undirected graph with n vertices
an array initially set to zero

BFS :—

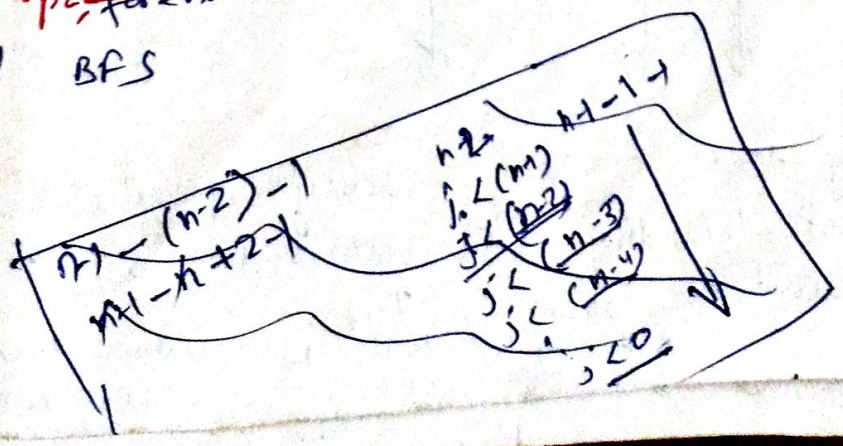
* differs from DFS in that
  → all unvisited vertices adjacent to v are visited next.
  → Then unvisited vertices adjacent to these vertices are visited & so-on.

BFS :

v_1
v_2 v_3
v_4 v_5 v_6 v_7
v_8

~~Algo~~
Algo :

```
Procedure BFS (V)
// - mark VISITED (i)=1   if the vertex is visited
   Q is a Queue
   Q → visited and checked
VISITED (V) ← 1
initialize Q to be empty
Call ADDQ (V, Q)
Loop
    for all vertices w adjacent to V do
        if VISITED (W)=0
            then [call ADDQ (W,Q)
                  VISITED (W) ← 1]
end Loop1;
Loop until Q is empty  if Q is empty then return
                       Call DELETEQ (W, Q)
End Loop2; forever
end BFS
```

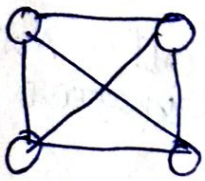


Connected components :—

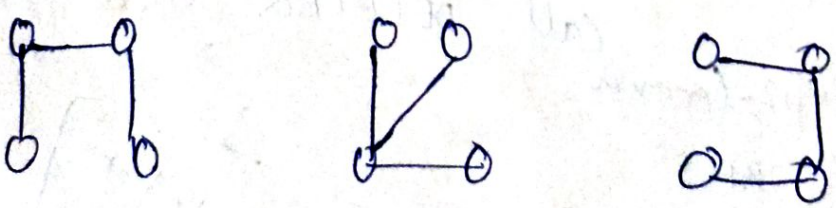
```
Procedure COMPONENT (G, n)
    // G has n ≥ 1 vertices
    // VISITED is a local array

    for i ← 1 to n do
        VISITED (i) ← 0    // initialize all vertices as unvisited
    end
    for i ← 1 to n do
        if VISITED(i) = 0 then
            Call DFS (i);    // find a component
            Output all newly visited vertices
              together with all edges incident
              to them.
    end
end COMPONENT
```

// Spanning Tree :—

Let us a graph is —



its 3 spanning trees are —

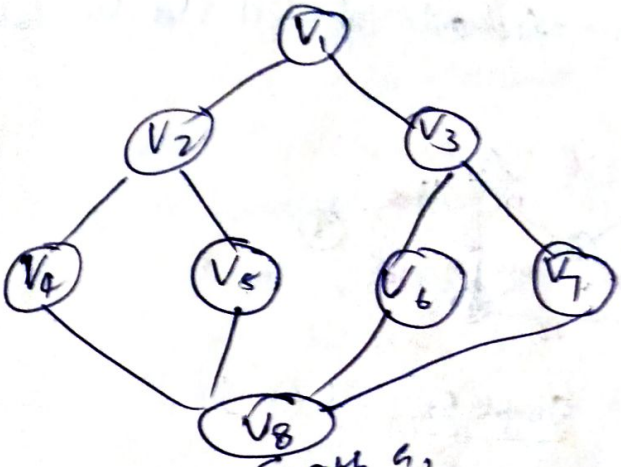


Any tree consisting solely of edges in G and including all vertices in G is called a spanning Tree.

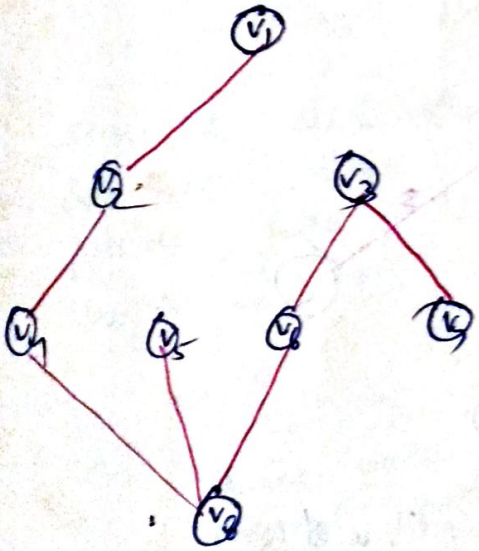
(A spanning tree will not contain fewer than (n-1) edges when n is total no of vertices

→ When DFS is called to result a spanning tree, that tree is known as Depth first spanning tree.

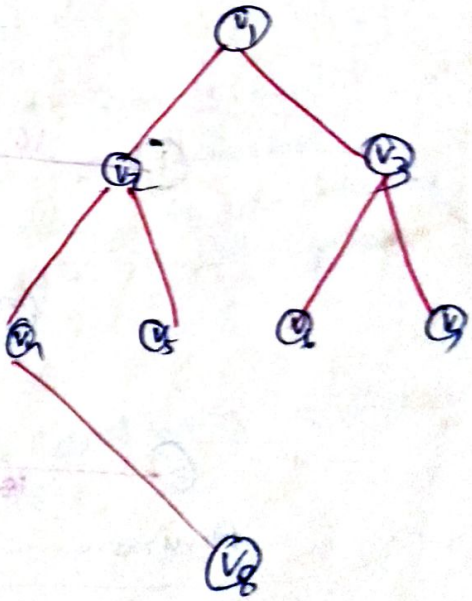
→ when BFS is used, the resulting tree is Breadth first spanning tree.



Graph G1



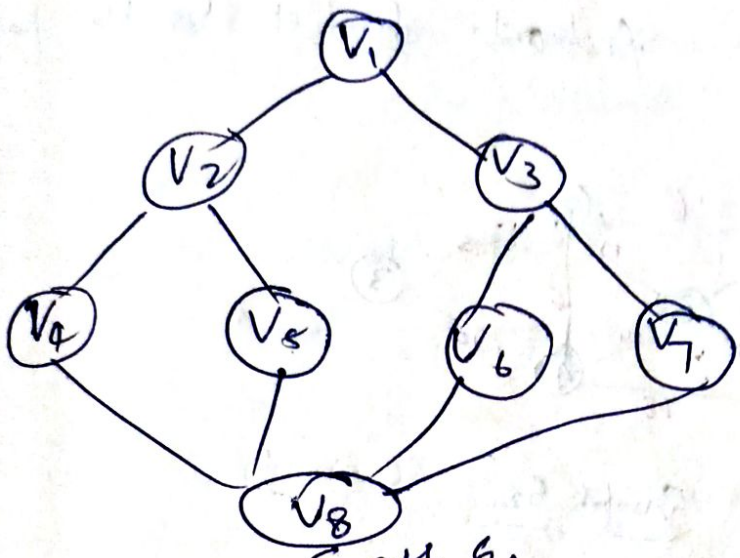
DFS spanning Tree



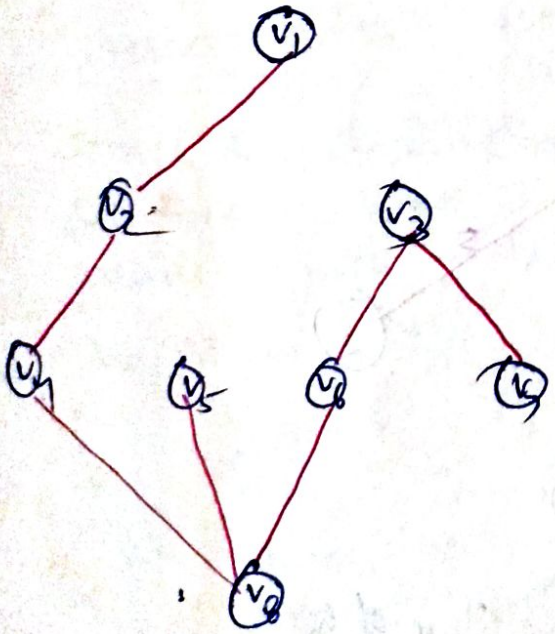
BFS spanning Tree

→ When DFS is called to result a spanning tree, that tree is known as Depth first spanning tree.

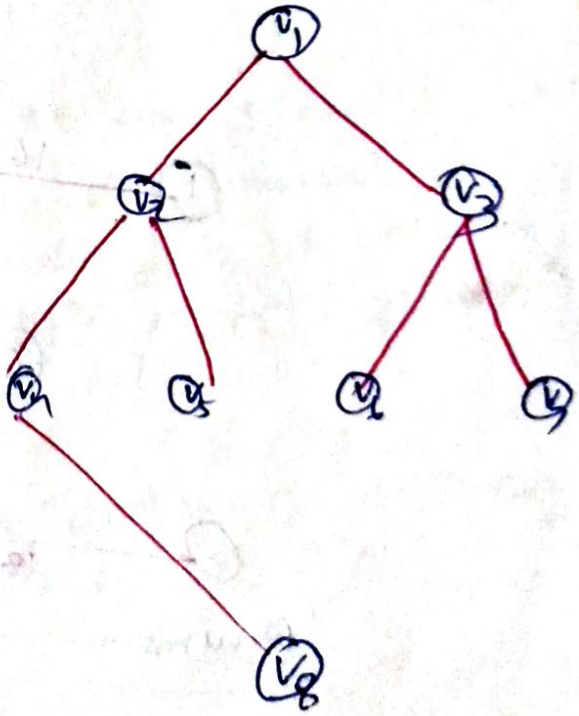
→ when BFS is used, the resulting tree is Breadth first spanning tree.



Graph G1



DFS Spanning Tree

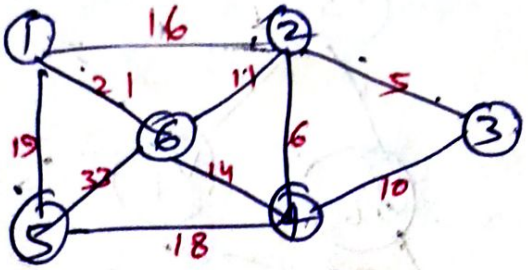


BFS Spanning Tree

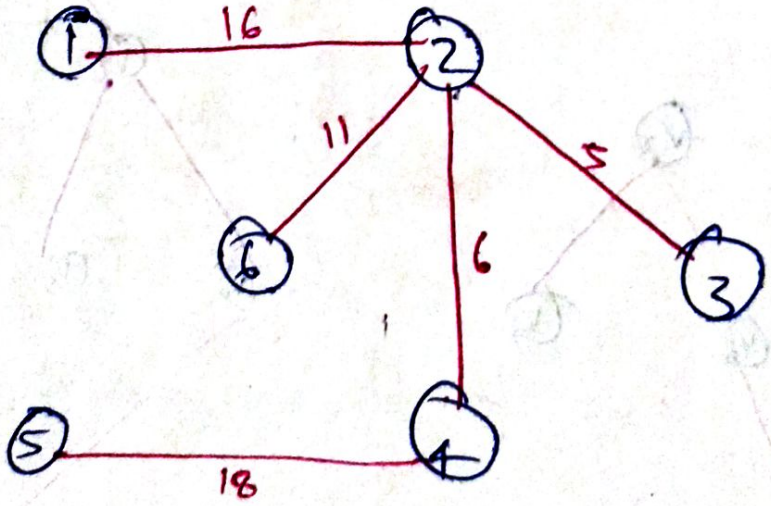
Minimum cost spanning Tree :-

The cost of a spanning tree is the sum of the costs (weights) of the edges in that tree. Thus a spanning tree of G with minimum cost is known as minimum cost spanning tree.

~~for this we find out shortest possible path~~



Graph G_2



Minimum cost spanning Tree of G_2

2. Non-linear DS - Tree, graph

~~Linear~~ data items is possible in linear fashion ~~read~~ one by one sequentially not

Kruskal - Algo

(T = mST
E = ~~Given~~ set of all edges in G)

~~Prim~~ mST
T ← ∅

while T contains less than (n-1) edges and E not empty do

Choose an edge (v, w) from E of lowest cost

delete (v, w) from E

if (v, w) does not create a cycle in T
then add (v, w) to T
else discard (v, w)

end

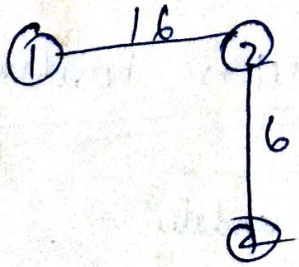
if T contains fewer than (n-1) edges then
Print ('no spanning tree')

Prim algo

Shortest Path :-

source & destination vertices are given & we have to go on shortest possible path from source to destination. Weight of the each edge is also given.

if in G_2
source node is ① & destination is ④ then shortest path will be :

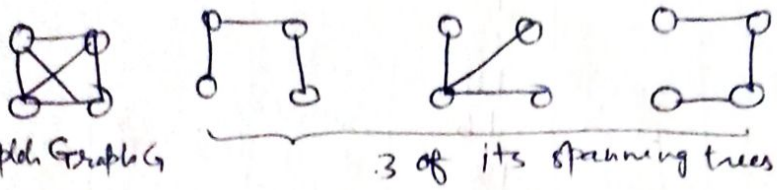


Path length = 22

Dijkstra Algo

Spanning Tree

Any tree that consists of edges of the connected graph G and all the vertices of G.



A complete Graph G 3 of its spanning trees

Min. cost spanning Tree

The cost of a spanning tree of a weighted undirected connected graph is the sum of the costs (weights) of the edges in the spanning tree. A MST is a spanning tree of least cost. Following algo are used for constructing the MST (T) —

1. Kruskal's Algo
2. Prim's Algo

These algo's use the Greedy Method.

For spanning trees, we use a least cost criterion. Following constraints must be satisfied for MST —

1. we must use only edges within the graph
2. ——— exactly (n-1) edges
3. we may not use edges that would produce a cycle.

Kruskal's Algo

```
T = { };                          //E is the set of all edges in G
while (T contains less than n-1 edges && E is not empty)
  {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w);
  }
if (T contains fewer than n-1 edges)
        print 'No Spanning Tree'
```

1. Construct the MST for following graph using Kruskal's & Prim's Algo. Also Prove that Prim's Algo generate a MST for this graph
2. Write down the algo of traversals of a graph. Which data structure are used in these traversal carry out these traversals for following graph
3. What is AVL tree. Write down the rules to be followed for constructing the AVL tree. Make a balanced tree for the months of the year.

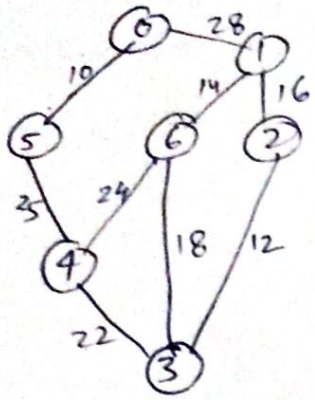


fig-01

Edge	Weight	Result	fig
—	—	initial	2
(0,5)	10	added to tree	3
(2,3)	12	added	4
(1,6)	14	added	5
(1,2)	16	added	6
(3,6)	18	discarded (cycle)	—
(3,4)	22	added	7
(4,6)	24	discarded (cycle)	—
(4,5)	25	added	8
(0,1)	28	not considered (not edge)	—

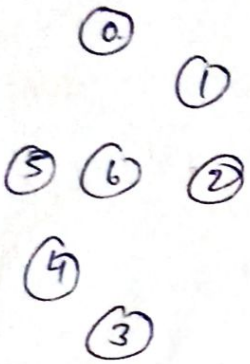


fig-2

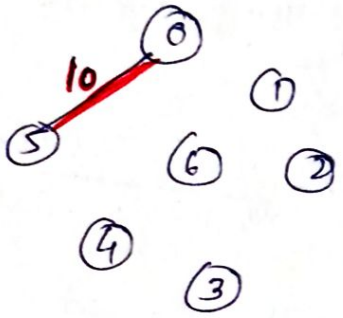


fig-3

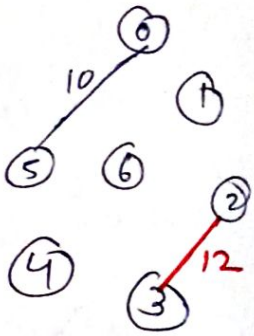


fig-4

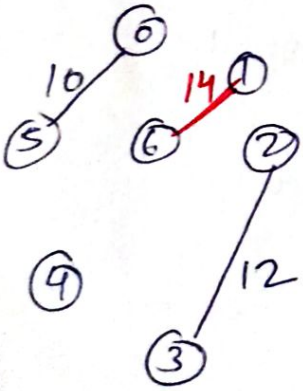


fig 5

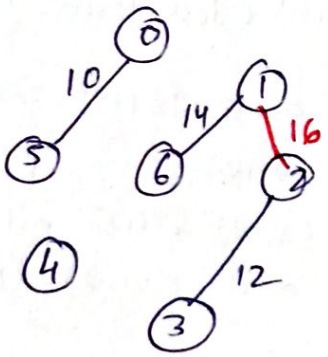


fig 6

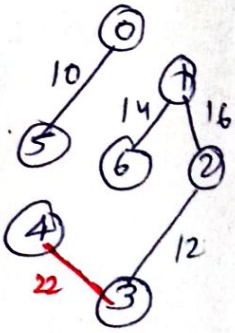
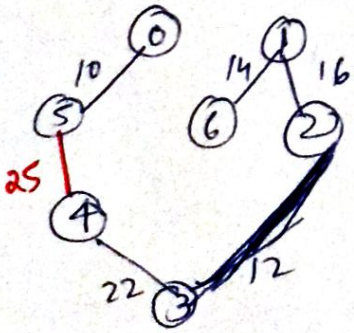


fig 7



Prove that Kruskal's algo generates a MST for an undirected connected graph

Show that —

(a) Kruskal's algo produces a spanning tree whenever a spanning tree exists.

(b) The spanning tree generated is of minimum cost.

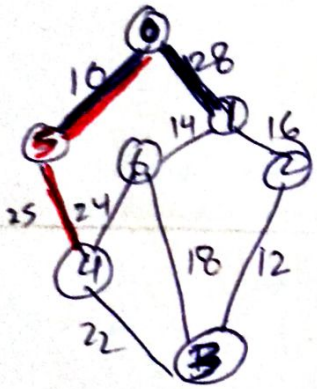
Prims Algo

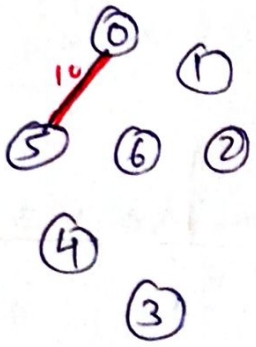
// T – set of Tree edges
// TV – vertices, that vertices that are currently in the tree

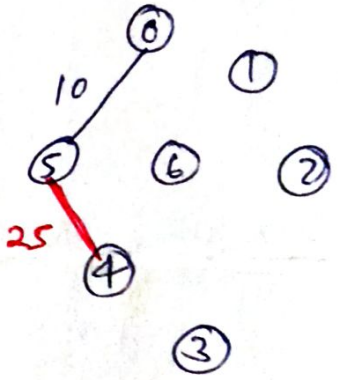
```
T = {};
TV = {0};          // starts with vertex 0 and no edges
while (T contains fewer than n-1 edges) // Repeat until T contains n-1 edges
    {
      let (u,v) be a least cost edge such that u ∈ TV and v ∉ TV
      if (there is no such edge)
            break;
      add v to TV;
      add (u,v) to T; // add a least cost edge (u,v) to T such that T ∪ {(u,v)} is also a Tree.
    }
if (T contains fewer than n-1 edges)
      print "No spanning Tree"
```

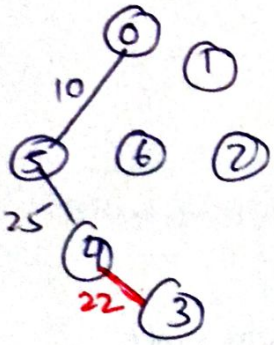
(edge repeat नहीं होनी चाहिए)

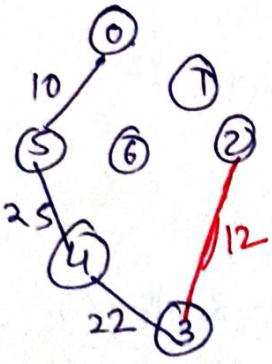
(current node से शुरू करना है और edge नहीं होनी चाहिए)

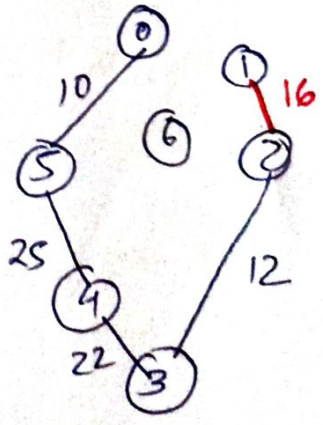


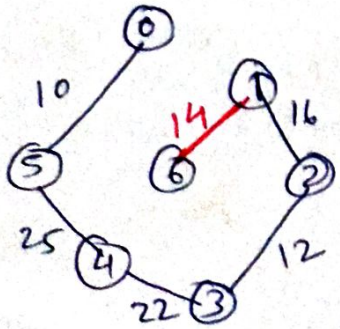










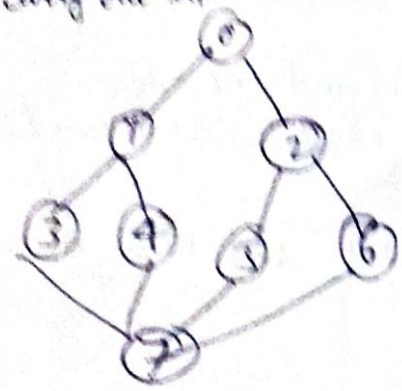


Prove that Prim's Algo finds a MST for every undirected connected graph

Traversal in Graph

① DFS (Stack is used)

carry out depth first search for following graph.



0 1 3 7 4 5 2 6

DFS (~~Beginning Vertex~~ V)

// V : Bag of vertex

nodeptr w

visited [V] ← True

Print V

~~Loop~~ (w ← ~~V graph[V]~~ to last node)

if (! visited [w → vertex])

DFS (w → vertex)

w = w → link

end loop;

end;

vertex link

② BFS (Queue is used)

0 1 3 7 4 2 5 6

0
1 2
3 4 5 6
7

BFS (V)

nodeptr w;

front ← NULL

Rear ← NULL

Print V

visited [V] ← True

addq (V);

Loop~~patrole~~ (front)

V = deleteq().

loop w ← V to last node

if (! visited [w → vertex])

Print (w → vertex)

addq (w → vertex)

visited [w → vertex] ← True

w = w → link

end loop

end loop