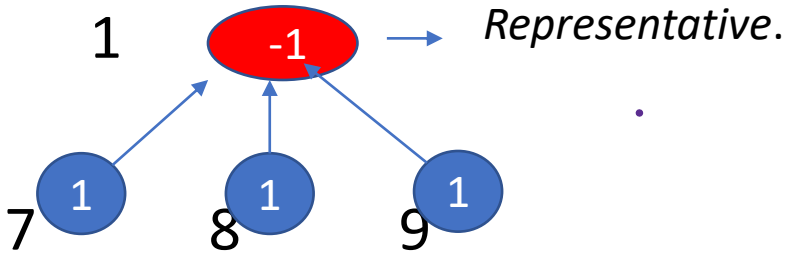
Data Structures for Disjoint Sets

Disjoint Sets Data Structure

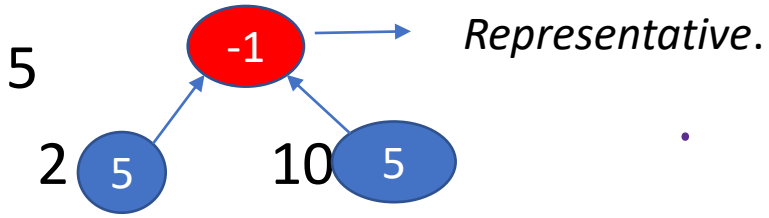
Example: S={S1,S2,S3}

S1={1,7,8,9} S2={2,5,10} S3={3,4,6}

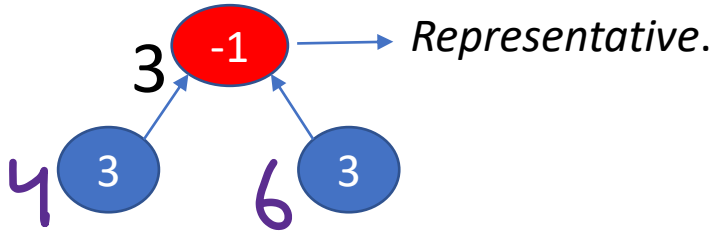
S1={1,7,8,9} --------- Represents as Tree



S2={2,5,10} --------- Represents as Tree



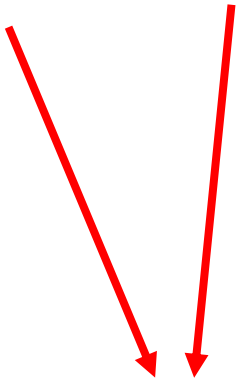
S3={3,4,6} } --------- Represents as Tree



Disjoint Sets Data Structure

A disjoint-set is a collection $S=\{S_1, S_2,\ldots, S_k\}$ of distinct dynamic sets.

What is disjoint set: S1={1,7,8,9} S2={2,5,10}



S = {S1, S2} where both S1 and S2 are disjoint sets

- Disjoint : if Si and Sj, i is not equal to j , are two sets which have no common elements ,mean there is no element that is in both Si and Sj
- Each set is identified by a member of the set, called Representative of that set
- Each set is represented as a tree
- Each set we have linked the nodes from children to the parent rather than from parent to the children.

Disjoint Sets Data Structure

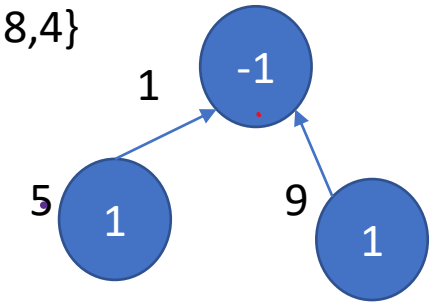
Disjoint set operations:

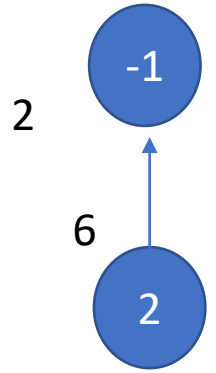
- MAKE_SET(x): Create a new set {x} containing the single element x. The object x must not appear in any other set in our collection. The leader of the new set is obviously x.
- FIND_SET(x): Find (the leader of) the set containing x.
- UNION(x, y): Replace two sets x and y in our collection with their union xU y. For example, Union(A,MakeSet(x)) adds a new element x to an existing set A. Union(x, y) has exactly the same behavior as Union(Find(x), Find(y)).

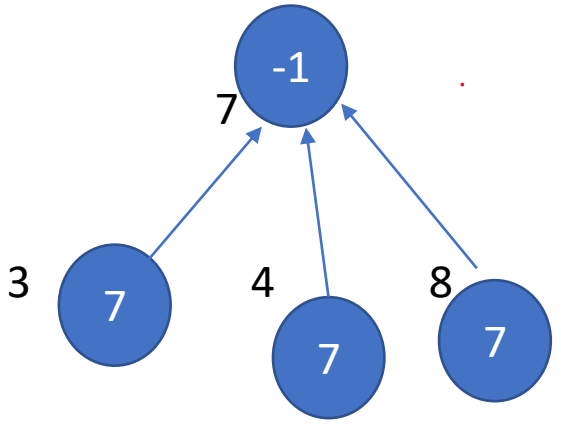
--The two sets are assumed to be disjoint prior to the operation

Disjoint Sets Data Structure

Example:
S1={1,5,9} S2={2,6} S3 ={3,7,8,4}

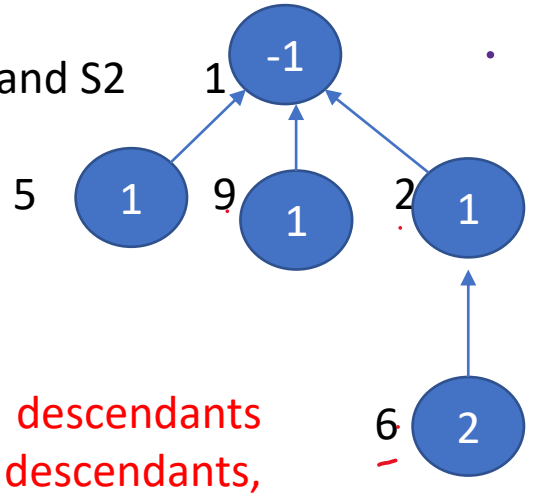


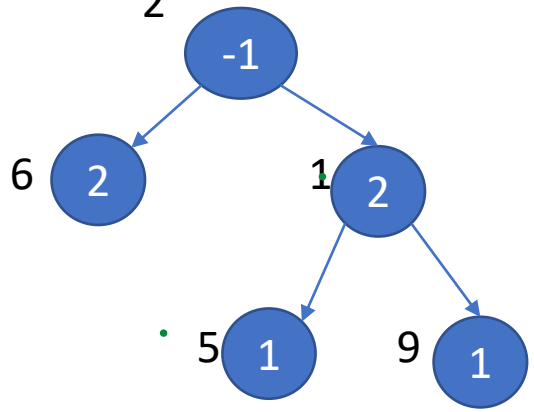





```
FIND_SET(x)
{
  while p[x]>0 do
     x=p[x]
 return x;
}
```

```
UNION(x,y)   //  x and y are representative  for two disjoint sets  S1 and S2
{
  p(y)←x;
  return x;
}
```



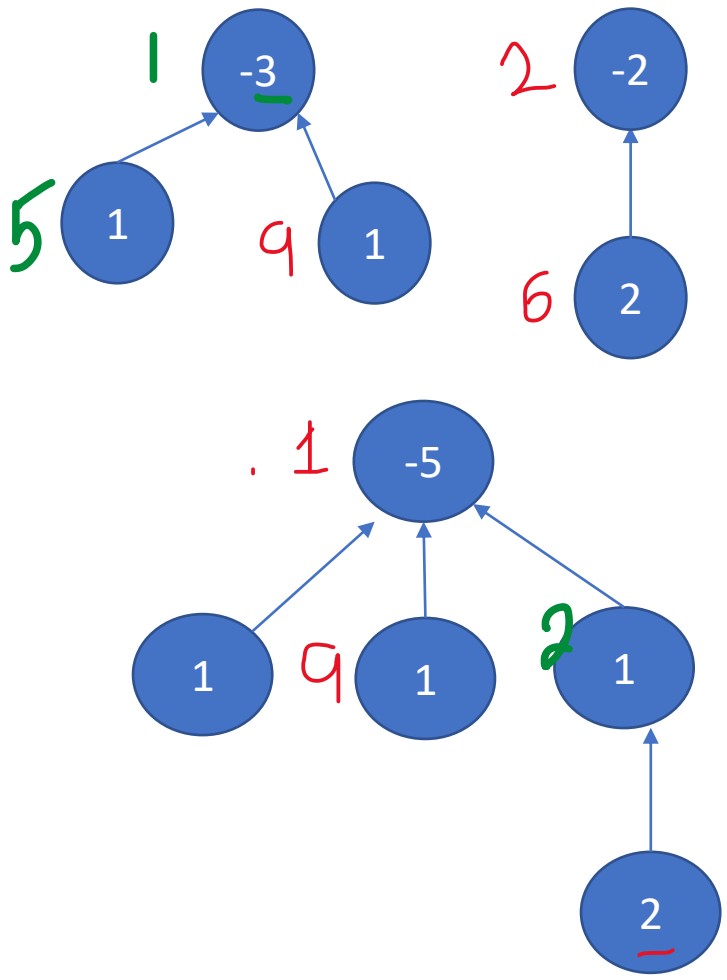


When the trees with roots *x* and *y* are merged, the node with more descendants becomes the parent. If the two nodes have the same number of descendants, then either one can become the parent. In both cases, the size of the new parent node is set to its new total number of descendants.

Total no of Comparison=10

Disjoint Sets Data Structure

```
weighted_union(x , y)
{
    temp ←p[x]+p[y]
     if p[x] > p[y] then
            p[x]=y;
            p[y]=temp;
     else
             p[y]=x;
             p[x]=temp;
}
```



An application of disjoint-set data structures

1. A disjoint-set data structure, also called a union–find data structure or merge–find set, is a data structure that stores a collection of disjoint sets
2. One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph for example, shows a graph with four connected components.
3. Used to determine whether two vertices belong to the same component, or whether adding an edge between them would result in a cycle.
4. It is also a key component in implementing Kruskal's algorithm to find the minimum spanning tree of a graph.

Disjoint Sets Data Structure

An application of disjoint-set data structures: To find the connected components

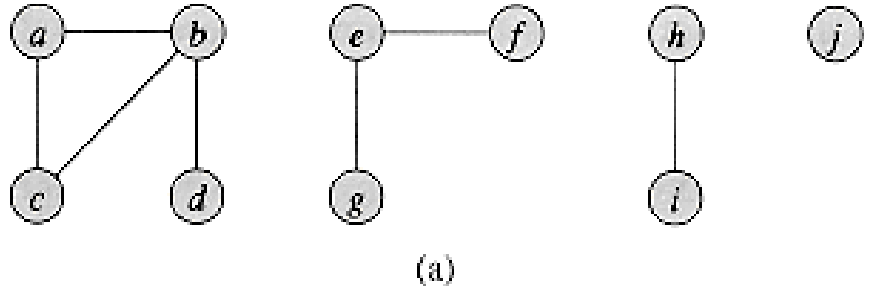
Example:

G=(V,E)=({ a ,b ,c ,d ,e ,f ,g ,h ,i ,j },{(b , d),(e ,g),(a , c),(h ,i),(a, b),(e , f),(b , c)}

CONNECTED-COMPONENTS(*G*)
1 **for** each vertex *v* ∈ *V*[*G*] **do**
2 MAKE-SET(*v*)
3 **for** each edge (*u*,*v*) ∈ *E*[*G*] **do**
4 **if** FIND-SET(*u*) ≠ *FIND-SET(*v*)* **then**
5 UNION(*u*,*v*)

Edge processed	Collection of disjoint sets
initial sets	{*a*} {*b*} {*c*} {*d*} {*e*} {*f*} {*g*} {*h*} {*i*} {*j*}
(*b*, *d*)	{*a*} { *b*, *d*} {*c*} {*e*} {*f*} {*g*} {*h*} {*i*} {*j*}
(*e* , *g*)	{*a*} { *b*, *d*} {*c*} { *e*, *g*} {*f*} {*h*} {*i*} {*j*}
(*a* , *c*)	{ *a*, *c*} { *b* ,*d*} { *e*, *g*} {*f*} {*h*} {*i*} {*j*}
(*h* , *i*)	{*a*, *c*} { *b* ,*d* } { *e* ,*g* } {*f*} {*h* ,*i*} {*j*}
(*a*, *b*)	{*a* ,*b* ,*c* ,*d* },{*e* , *g*} ,{*f*} {*h* ,*i*} {*j*}
(*e* , *f*)	{*a* ,*b* , *c*, *d*} {*e* ,*f* , *g*} { *h*, *i*} {*j*}
(*b*, *c*)	**{*a* ,*b* ,*c* , *d*} {*e* ,*f* , *g*} { *h* , *i*} {*j*}**

The collection of disjoint sets after each edge is processed



(a)

1.{a ,b ,c ,d} 2.{e ,f , g} 3.{h ,i } 4.{j}

A graph with four connected components: {a, b, c, d}, {e, f, g}, {h, i}, and {j}.

Disjoint Sets Data Structure

Once CONNECTED-COMPONENTS has been run as a preprocessing step, the procedure SAME-COMPONENT answers queries about whether two vertices are in the same connected component or not.

```
SAME-COMPONENT(u,v)
1 if FIND-SET(u) = FIND SET(v) then
2 return TRUE
3 else return FALSE
```

Minimum Spanning Tree

Minimum Spanning Tree

What is Spanning Tree?

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.
- Let G=(V,E) be an undirected connected graph. A sub-graph t=(V,E') of G is a spanning tree of G iff t is a tree.

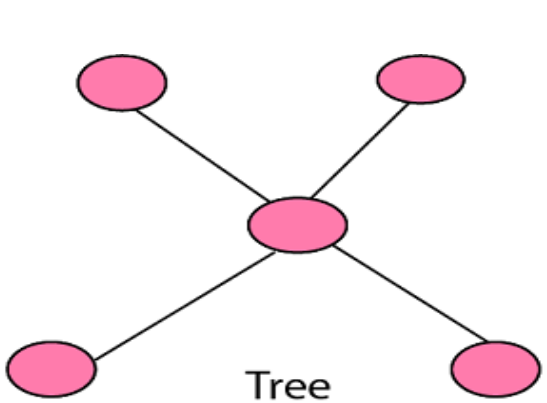
.

Tree:

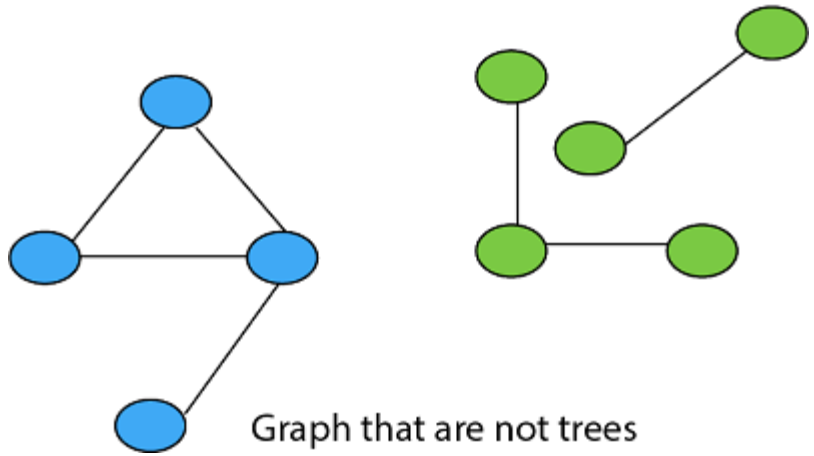
A tree is a graph with the following properties:

1.The graph is connected (can go from anywhere to anywhere)

2.There are no cyclic (Acyclic)

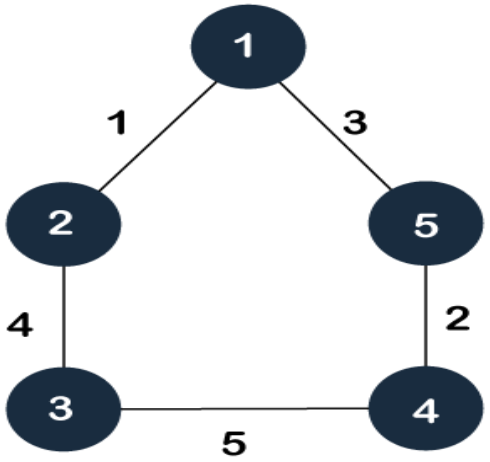


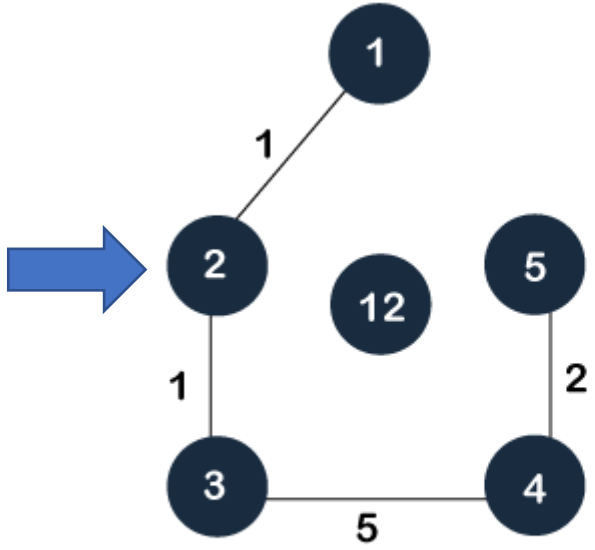
Tree



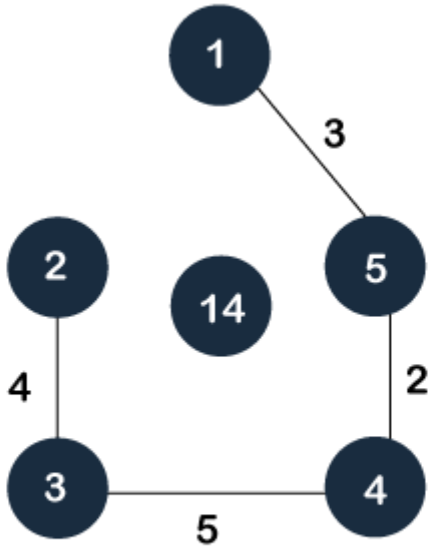
Graph that are not trees

Minimum Spanning Tree

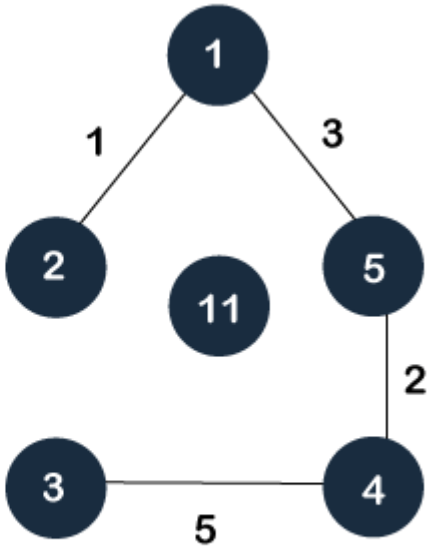




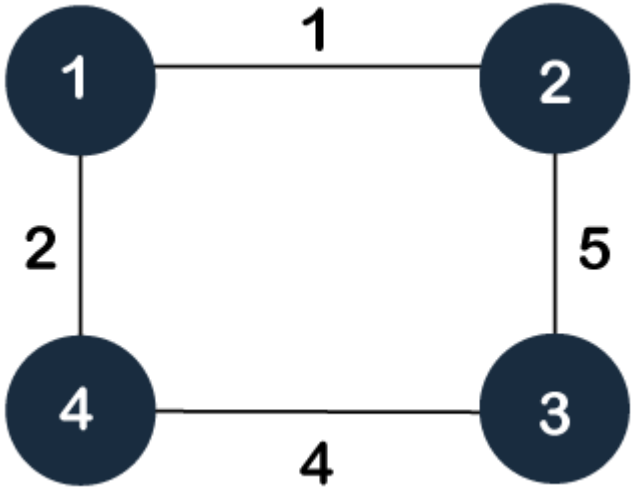
Spanning tree 1



Spanning tree 2



Spanning tree 3



Spanning Tree-1

Spanning Tree-2

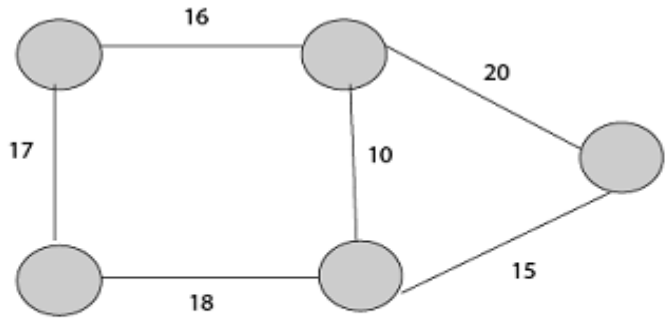
Spanning Tree-3

Spanning Tree-4

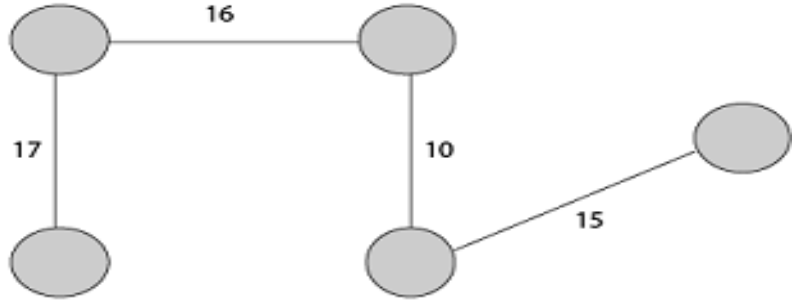
Minimum Spanning Tree

What is Spanning Tree?

Minimum Spanning Tree is a Spanning Tree which has minimum total cost. If we have a linked undirected graph with a weight (or cost) combine with each edge. Then the cost of spanning tree would be the sum of the cost of its edges.



Connected , Undirected Graph



Minimum Cost Spanning Tree

Total Cost = 17+16+10+15=58

Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

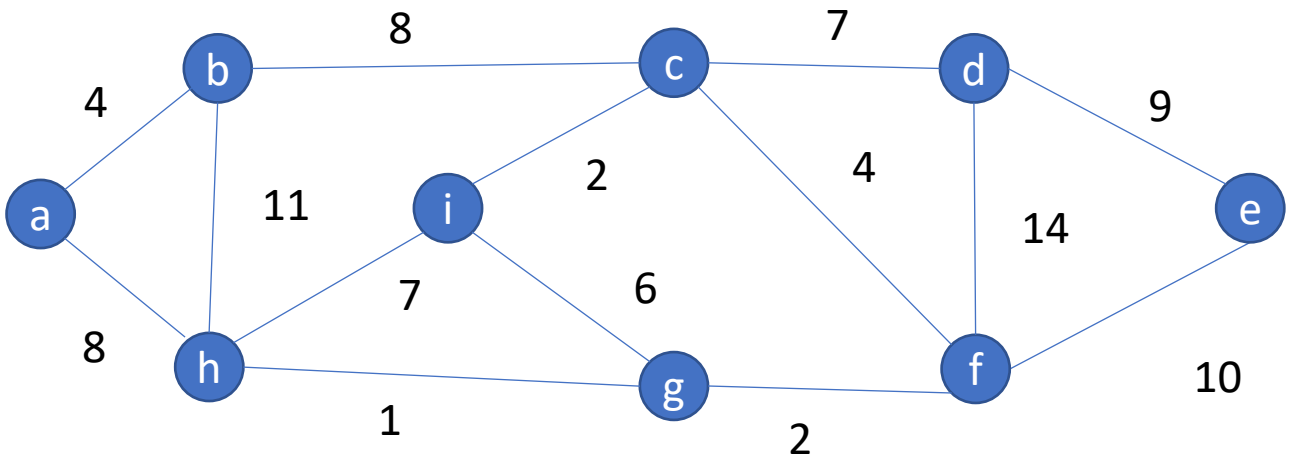
1.Kruskal's Algorithm

2.Prim's Algorithm

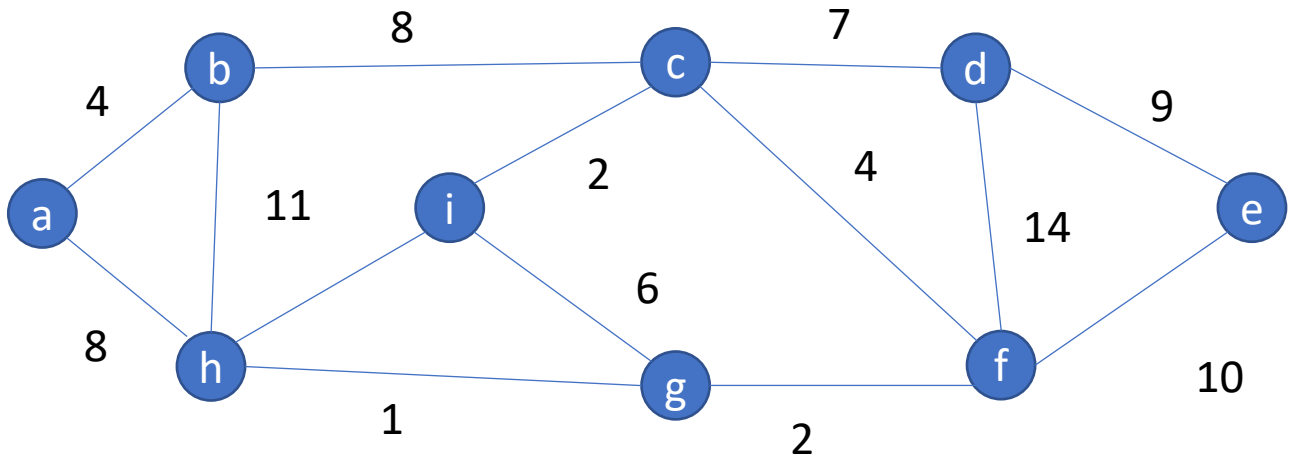
Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph.

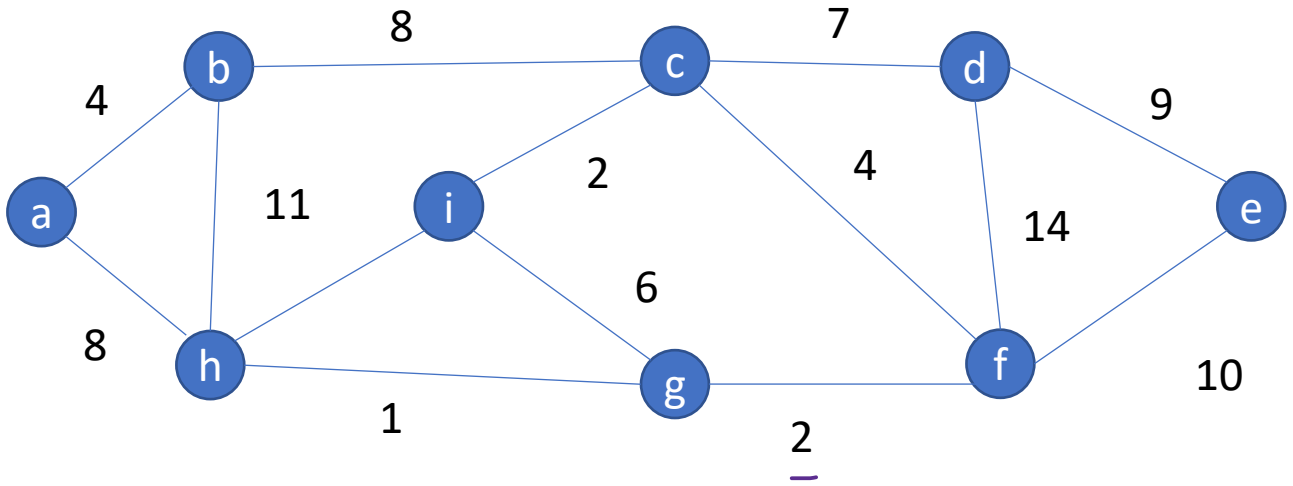
To understand Kruskal's algorithm let take example:



Minimum Spanning Tree



Step:1 Remove all loop and parallel edges given graph (if parallel edges present then keep one which has the lowest cost)



Minimum Spanning Tree

Step:-2 Arrange all edges in their increasing order of weights

Weights	Source	Destination
1	g	h
2	f	g
2	i	c
4	a	b
4	c	f
6	i	g
7	h	i
7	c	d
8	a	h
8	b	c
9	d	e
10	e	f
11	b	h
14	d	f

Edges that does not leads to cycle

(g, h) --- FIND_SET(h) ≠ FIND_SET(g) {g,h}

(f, g) --- FIND_SET(f) ≠ FIND_SET(g) {f,g,h}

(i, c) --- FIND_SET(i) ≠ FIND_SET(c) {i,c}

(a, b) --- FIND_SET(a) ≠ FIND_SET(b) {a,b}

(c , f) --- FIND_SET(c) ≠ FIND_SET(f) {f,g,h,i,c}

(i, g) --- FIND_SET(i) = = FIND_SET(g) Hence discarded

(i, h) --- FIND_SET(h) = = FIND_SET(g) Hence discarded

(c, d) --- FIND_SET(c) ≠ FIND_SET(d) {f,g,h,i,c,d}

(a, h) --- FIND_SET(a) ≠ FIND_SET(h) {f,g,h,i,c,d,a,b}

(b, c) --- FIND_SET(b) = = FIND_SET(c) Hence discarded

(d, e) --- FIND_SET(d) ≠ FIND_SET(e) {f,g,h,i,c,d,a,b,e}

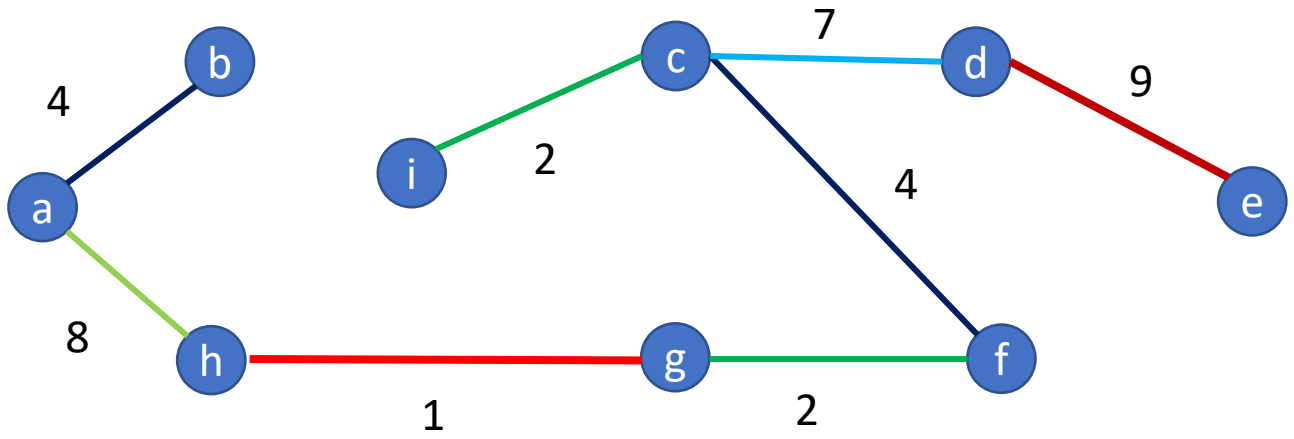
(e, f) --- FIND_SET(e) = = FIND_SET(f)) Hence discarded

Hence discarded

Hence discarded

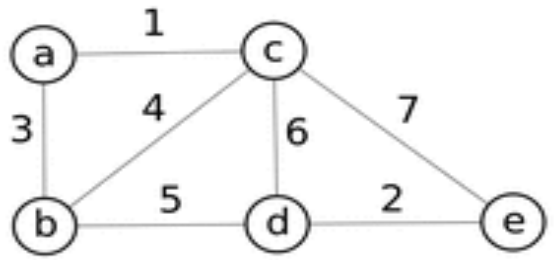
Minimum Spanning Tree

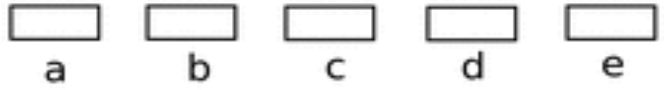
Step-3: Add the edge which has least weightage



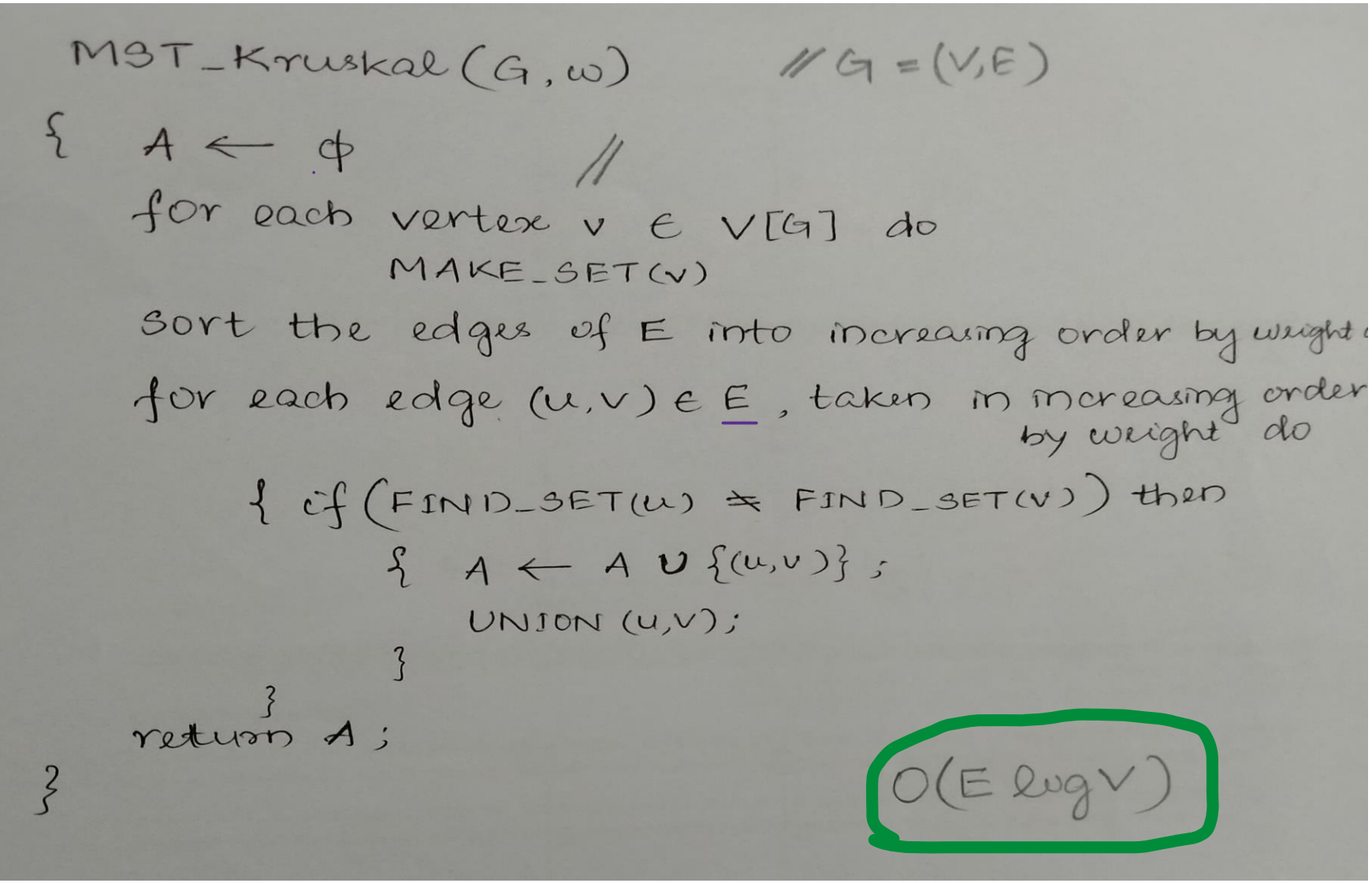
Total edge cost= 1+2+2+4+4+7+8+9=37

2.





Minimum Spanning Tree



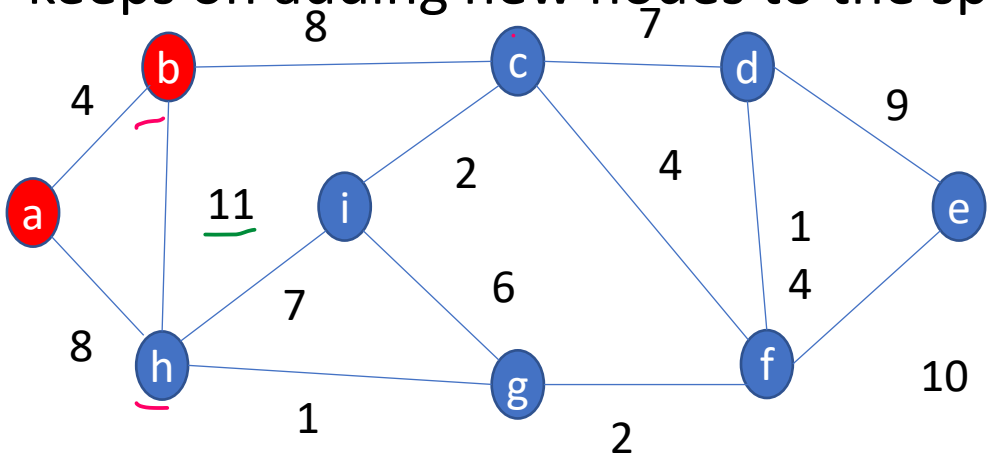
1. First we initialize the set A to the empty set
2. create |v| trees, one containing each vertex with MAKE-SET procedure.
3. Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.
4. Only add edges which doesn't form a cycle , edges which connect only disconnected components.

Time Complexity:

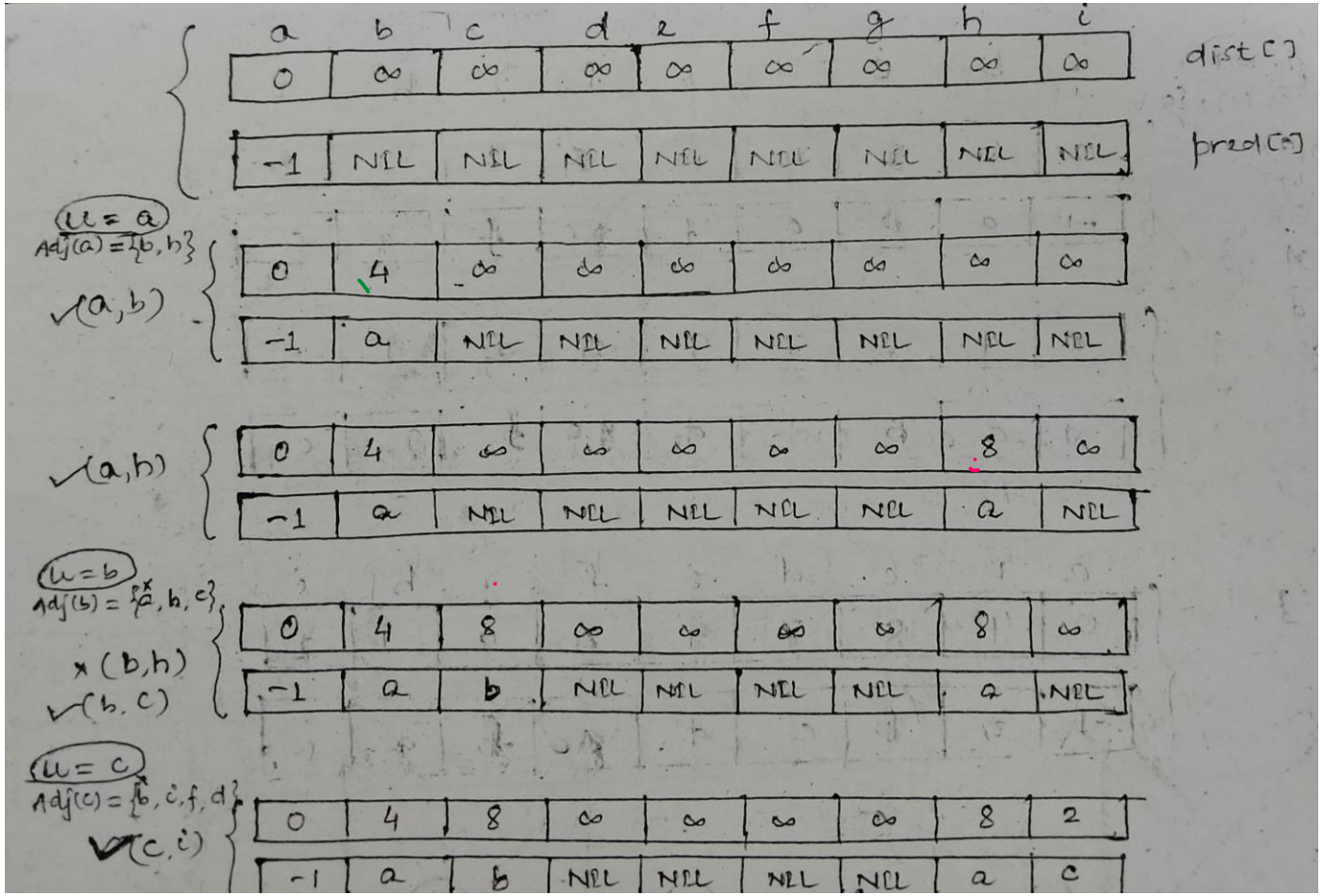
In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(ElogV), which is the overall Time Complexity of the algorithm.

Minimum Spanning Tree

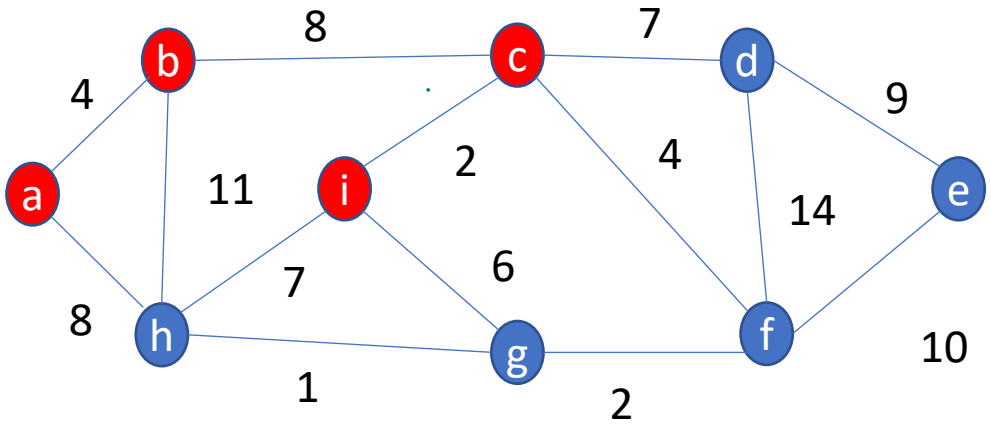
Prim's algorithm: in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

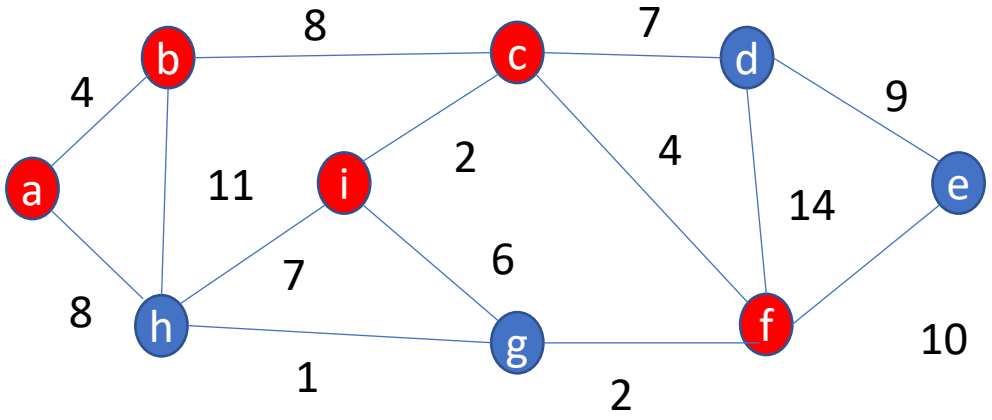


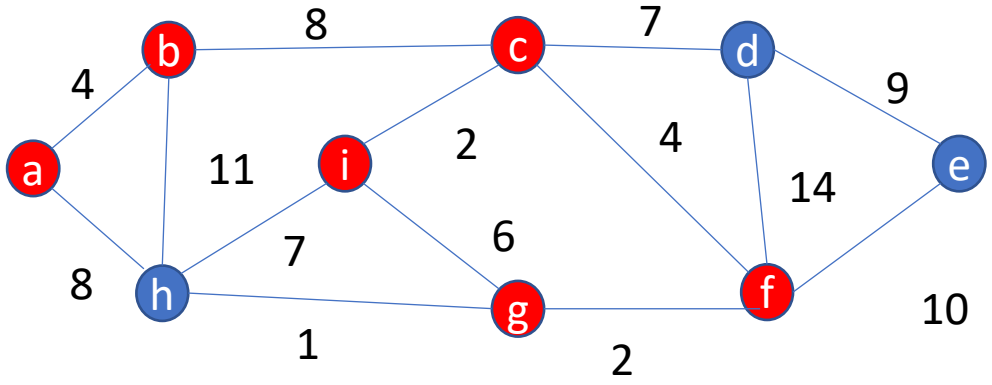
1. Taking Two array dist[ ] and pred[ ]
2. Let start from node a. u=a
3. dist[ ] of all node is ∞ except a and pred[ ] is NIL for all nodes except a
4. Adj(a)={b , h}
5. Next node u=b
6. Adj(b)={a, c , h} ab---is already considered
   (b,c)----updated
   (b, h) ------weight(h) > weight(b,h) -- yes then update else remain same

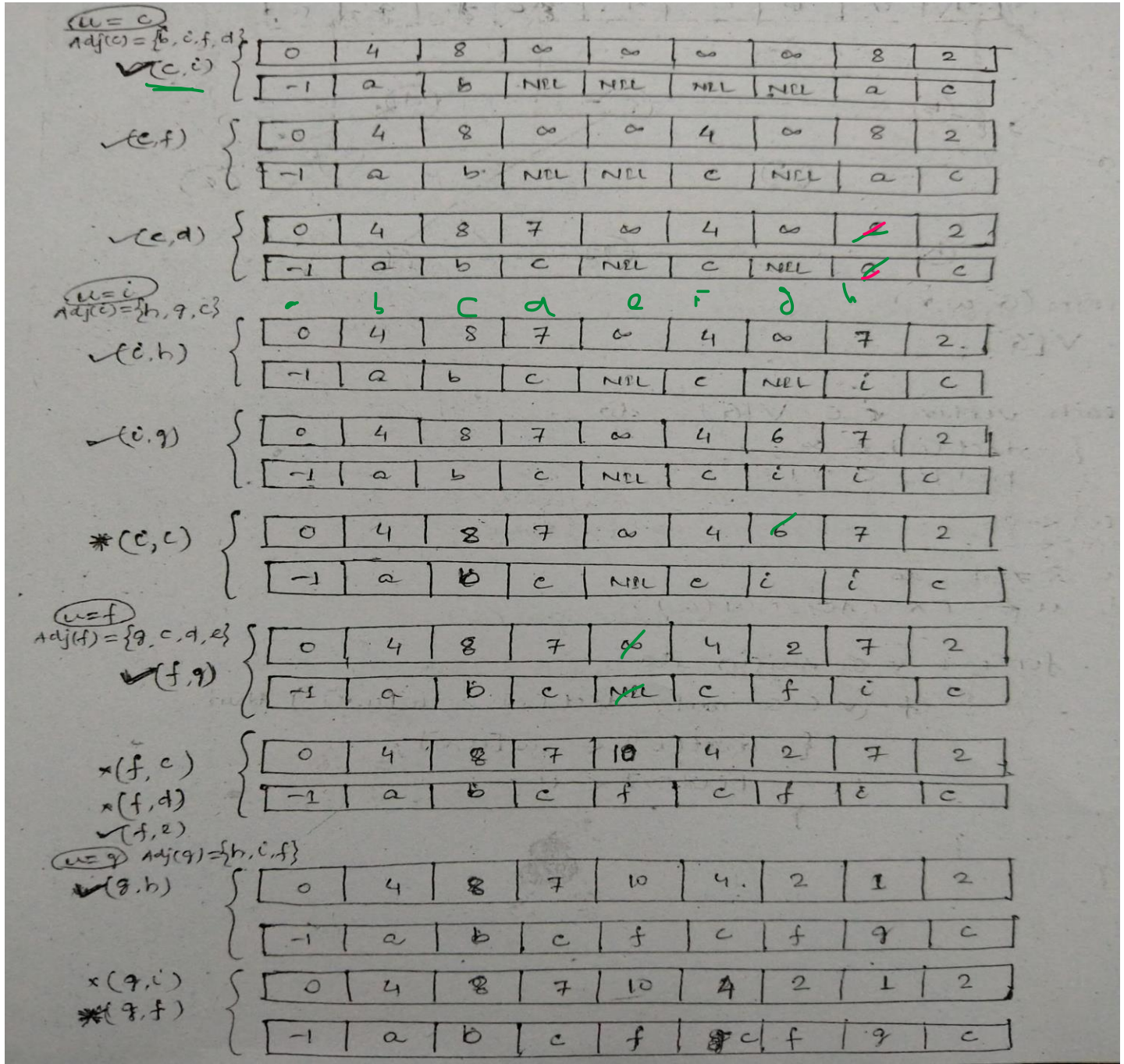


Minimum Spanning Tree

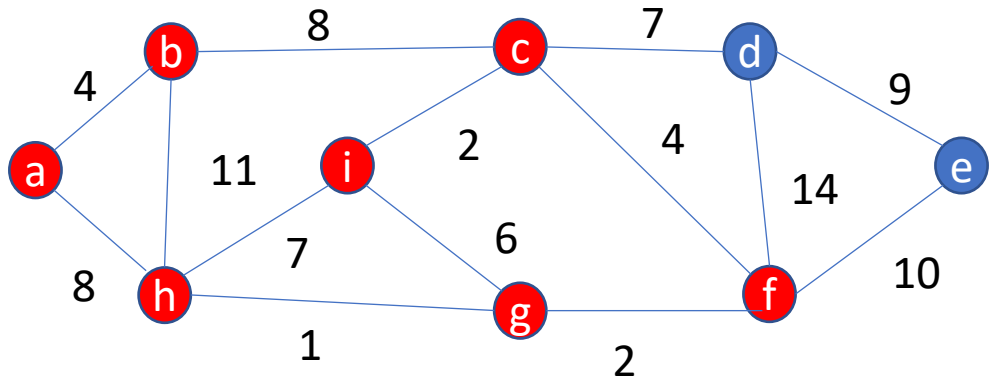


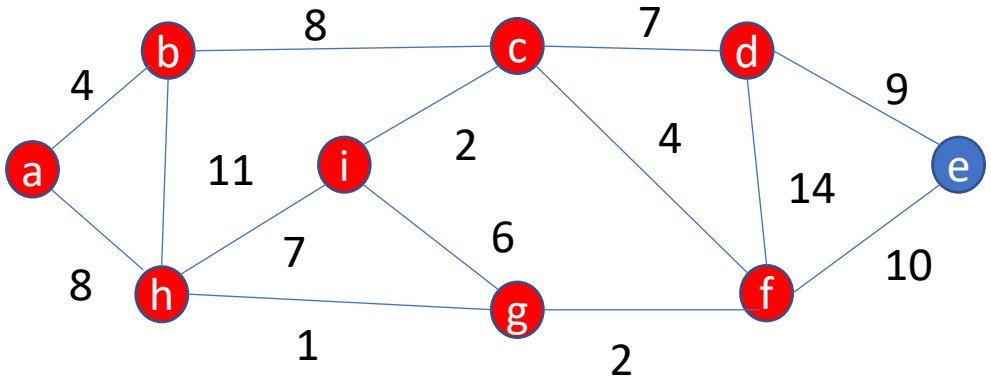


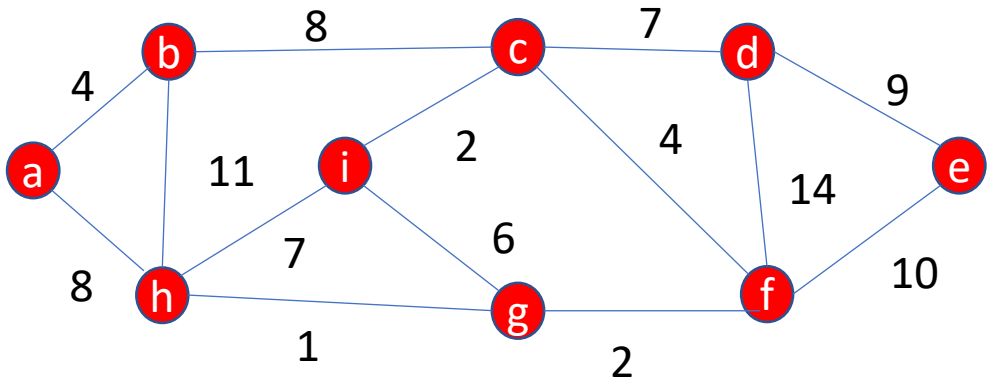


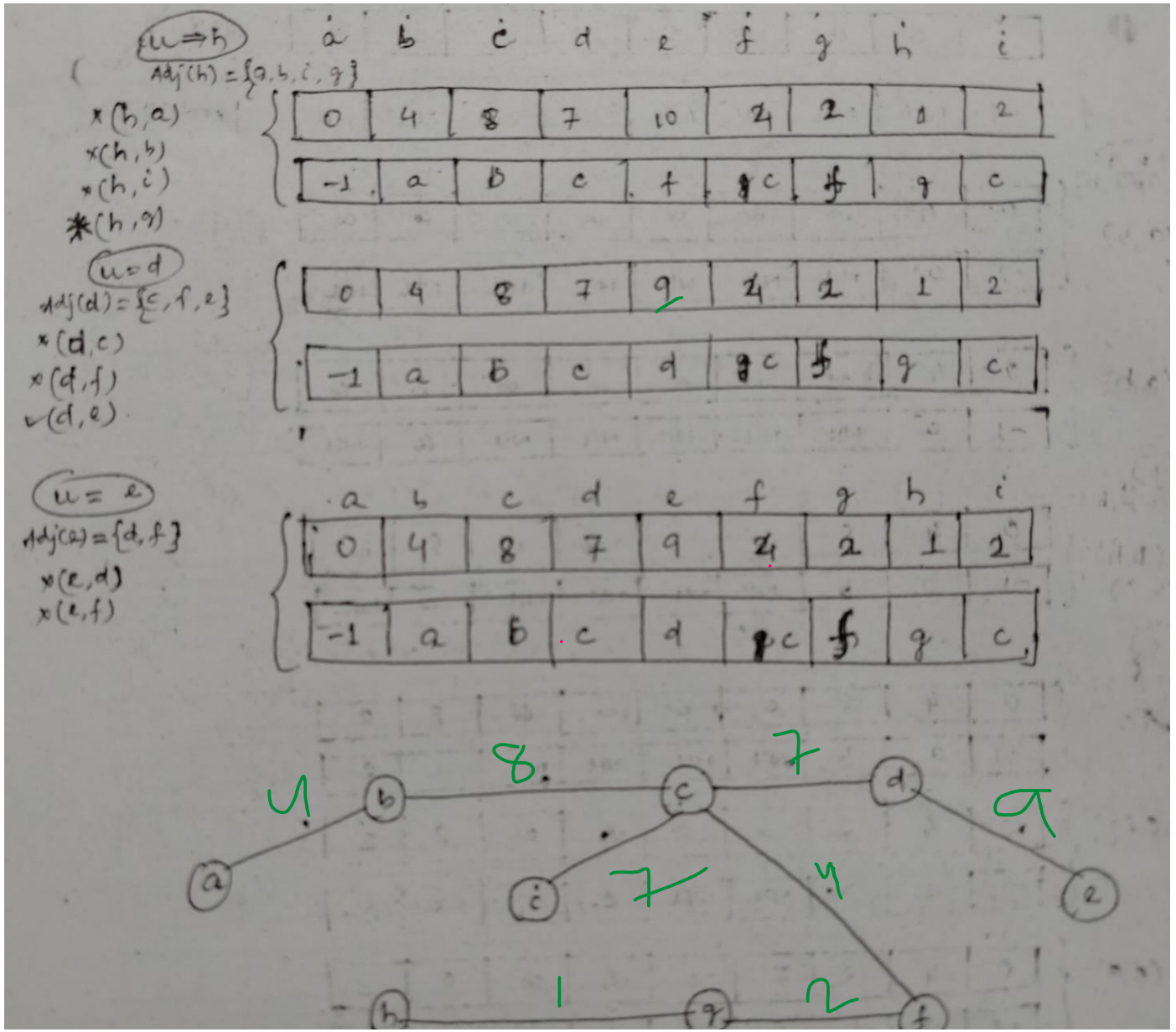


Minimum Spanning Tree









Minimum Spanning Tree

Prim's Algorithm

Steps:

1. Remove all loops and parallel edges
2. Choose any arbitrary node as root node :

In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

3. Check outgoing edges and select the one with less cost

Example: After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.

Minimum Spanning Tree

Prim's Algorithm

```
MST_PRIM (G, w, r)
{  Q ← V[G] ;

   for each vertex i ∈ V[G]  do
        {  dist[i] ← ∞ ;
           pred[i] ← NIL ;
        }
   dist[r] ← 0 ;
   pred[r] ← -1 ;
   while Q ≠ ϕ  do
      {  u ← EXTRACT_MIN(Q) ;

         for each v ∈ Adj(u)  do
            {  if (v ∈ Q and  dist[v] > w[u,v] ) then
                  {  dist[v] ← w[u,v] ;
                     pred[v] ← u ;
                  }
            }
      }
}
```

w is the weight matrix
r is the starting vertex

Time Complexity Analysis

- If adjacency list is used to represent the graph, then using breadth first search, all the vertices can be traversed in O(V + E) time.
- We traverse all the vertices of graph using breadth first search and use a min heap for storing the vertices not yet included in the MST.
- To get the minimum weight edge, we use min heap as a priority queue.
- Min heap operations like extracting minimum element and decreasing key value takes O(logV) time.

So, overall time complexity
= O(E + V) x O(logV)
= O((E + V)logV)
= O(ElogV)