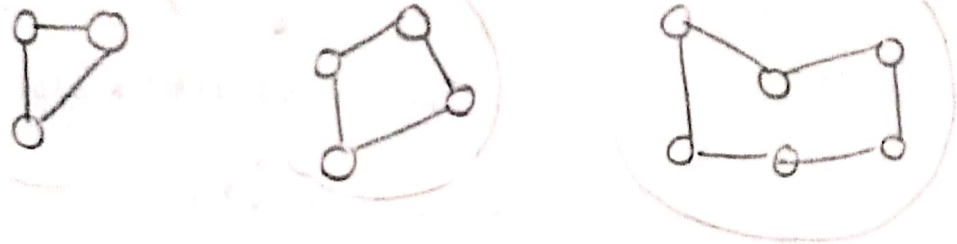
Day 40 : MST and Disjoint sets.



Undirected graph

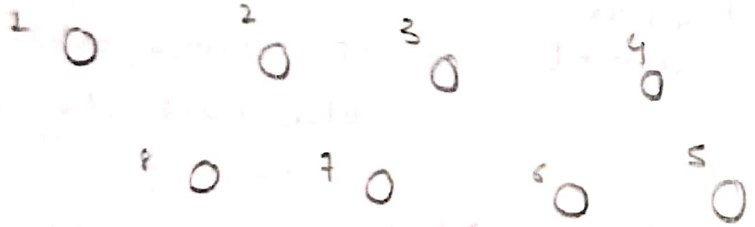
Disjoint sets can be used to segregate different connected components

↓

can be used when edges are encountered one by one and not at once. (online)

Disjoint set union

Disjoint sets is an online data structure ←



i) edge $N_A N_B$

ii) query : number of connected components / clusters

one way could be to calculate the number of connected components at each query using BFS or DFS, which can be a little inefficient, where disjoint set union can help

Edge : $N_A - N_B$

```
c1 = findCluster(NA)
c2 = findCluster(NB)
if(c1 != c2){
    union(c1, c2);
}
```

Disjoint set union functions in the way that it finds the cluster of each candidate node and if they belong to different clusters, merge or join them and make them one cluster.

when all nodes are Dint, to 1.

↑

starting at n

↑

by just maintaining a count of clusters ← (in other language a connected component)

A unique identifier for each cluster can be its root.
one of the nodes which can be regarded as root
of the tree. (which can act as the identity of the cluster)

↓

Each cluster with N nodes can be thought to have
N-1 edges (since when we get nodes in the same cluster, we don't operate on them.)
(tree like structure)

```
int findCluster(node A){
    if(parent[A] == -1) return A;
    return findCluster(parent[A]);
}
```

→ the root node would have no parent

→ won't work for nodes other than the roots.

```
void union(int c1, int c2){
    parent[c2] = c1;
    return;
}
```

findCluster()
findCluster()

↓

make c_1 parent of c_2

findCluster

Time complexity: $O(n) * (n^2$ max queries$) \rightarrow O(n^3)$

↓ max possible edges in a connected graph

↓ need to be optimised.

↓

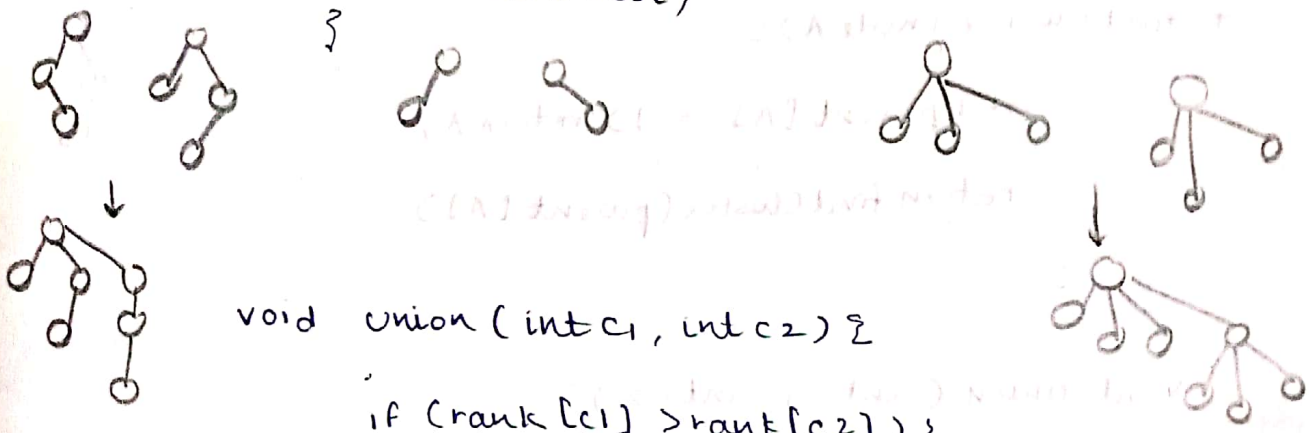
Such a cluster is also possible, in this case findCluster will become $O(n)$.

Space complexity: parent array $\rightarrow O(n)$.

path compression: the parent for each node in the cluster can be the root of the cluster. That will reduce the complexity for find cluster to O(1).

O(1) ←

```
int findCluster(int A){
    if(parent[A] == -1) return A;
    int root = findCluster(parent[A]);
    parent[A] = root;
    return root;
}
```



```
void union(int c1, int c2){
    if(rank[c1] > rank[c2]){
        parent[c2] = c1;
        rank[c1] += rank[c2];
    }
    else{
        parent[c1] = c2;
        rank[c2] += rank[c1];
    }
}
```

(best possible tree here)

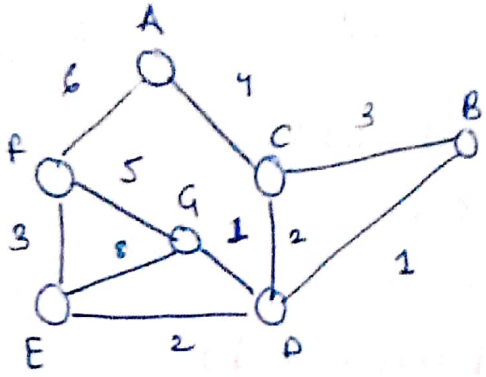
① maintain a rank array which indicates the no of nodes in the cluster headed by this node.

② rank array can be the measure of height of the cluster headed by that vertex (height will increase only when the height of the cluster is same)

amortized O(1) per update on edge → n^2 operations

space: 2n space → O(n) (for 2 arrays utilized)

d.



Minimum spanning tree.

↓

all vertices connected,

n-1 edges, with minimum sum weight.

i) sort edges as per weights

ii) for each edge till compiled number of edges != n-1

check if both points are in different clusters or same. If different, take union. otherwise ignore.

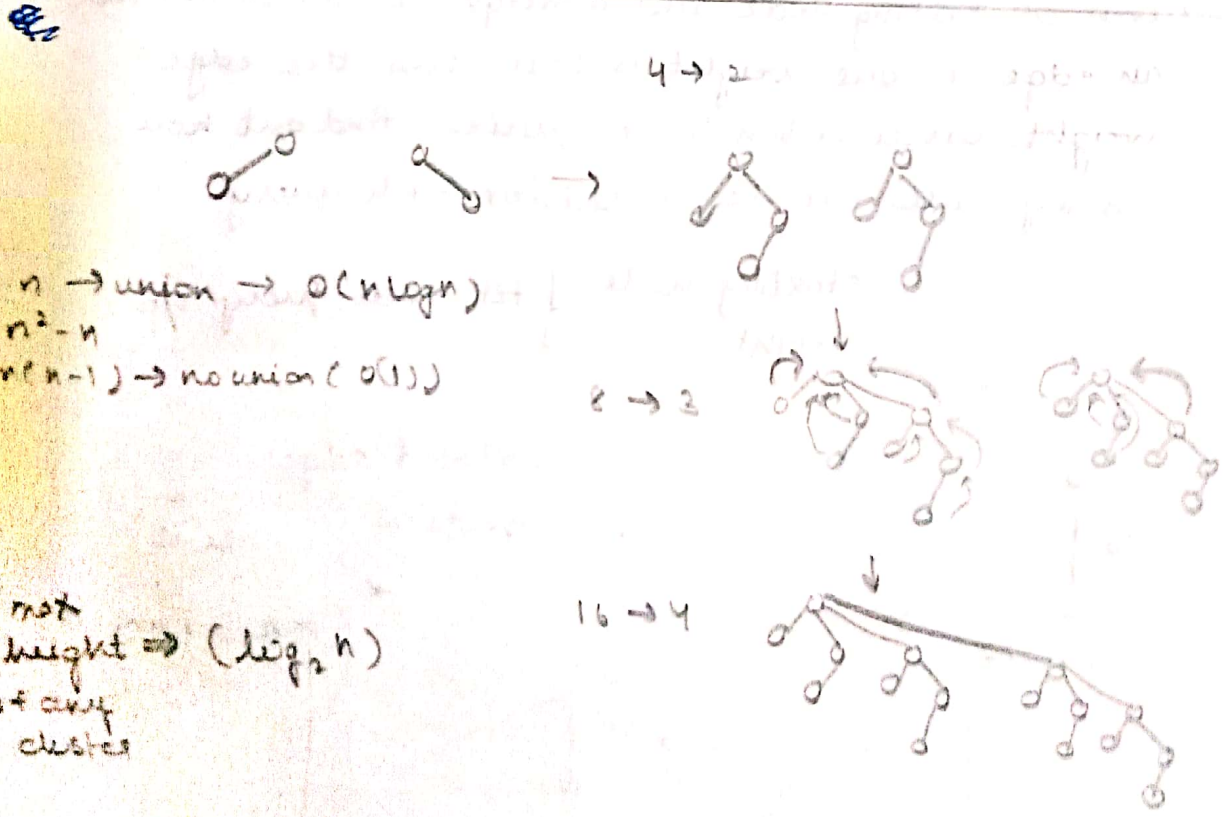
$O(E \log E + E)$

(Kruskal's algorithm)

↓

if edges are already sorted, then $O(E)$

since find cluster & union are ammortized $O(1)$



next height ⇒ $(\log_2 n)$ of any cluster

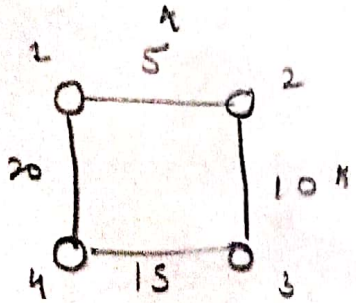
* union based on height as rank

```
void union(int c1, int c2){
    c1 = findCluster(c1);
    c2 = findCluster(c2);
    if (rank[c1] > rank[c2])
        parent[c2] = c1;
    else if (rank[c2] > rank[c1])
        parent[c1] = c2;
    else {
        parent[c2] = c1;
        rank[c1]++;
    }
}
```

limits the height to logn. for each cluster

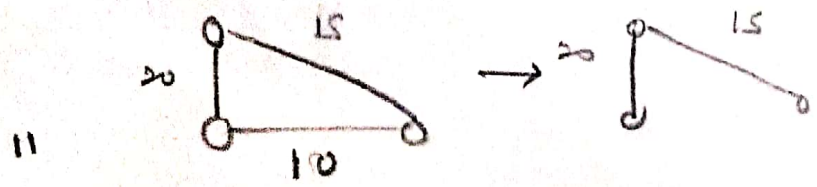
Q. Given a graph (bush), may or maynot be connected. Given a starting node and a weight W. we can traverse an edge if our weight is less than the edge's weight. Given a bunch of queries, find out how many nodes can be visited for each query

starting node
weight
} for each query i/p.



10^5 nodes + edges
10^5 queries

*



(queries are offline)

merge the query weights & edge weights in descending order.

↓

for any query, we only want to consider the edges with weight more than query weight

↓

edge: union → rank can be size of the cluster

(**) query: cluster which start node is part of.

since I know that the clusters formed are using weights more than the query weight.

↓

return the size of the cluster of the start node