- **Disjoint Set**

↳ To find if 2 nodes are in the same component, this problem can be easily solved by Disjoint Set. There are more concepts in DS which are Union, finding parent, Rank & Path compression.

↳ Example: Suppose n is 7, where nodes are 1 to 7.

Just for an example

- Union(1,2)
- Union(2,3)
- Union(4,5)
- Union(6,7)
- Union(5,6)
- Union(3,7)

Initially, All the nodes are in diff. components

1 2 3 4 5

6 7

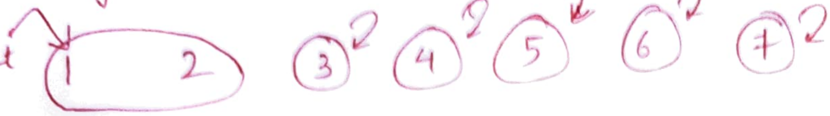
All of the nodes ~~are~~ are the parents of itself.

↳ These are the operations, we have to perform and after that, we can find any 2 nodes are in the same component or not.

The main idea behind this (for finding if 2 nodes are in component or not). To find the parent of both nodes and if the parents are same, we can say they are in same component otherwise not

After doing union(1,2)

parent



union(2,3)

union(4,5)

union(6,7)

union(5,6)

Here, all the union operations are completed and we can find if 2 nodes are in the same comp. or not.

Ques: 2 and 6

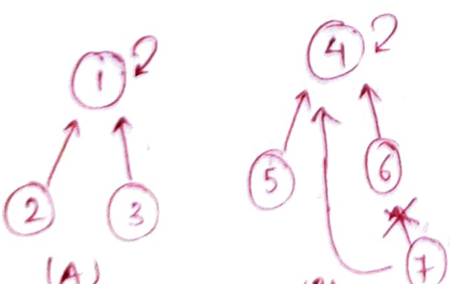
parent of 2 → 1
parent of 6 → 4

} Different parent, which ~~mean~~ means they are not in the same component.

Ques: 5 and 7

parent ~~of~~ 5 → 4
" " 7 → 4

} They are in the same component.



(A) (B)

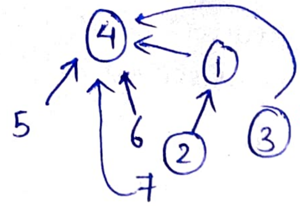
Here, 7 is pointing to 6 and then 4. In case we need to find the parent of 7, then it has to be traverse from 7 to 6 to 4. But we can make direct parent of 7 by Path compression.

Here, if we want to union both of these trees, we have to find rank of both of the trees,

A rank → 1
B rank → 2

} B > A ranks.

Here smaller rank tree will be attached to the greater one. and if the ranks are same, just attached anyone and increase the rank of attached tree.



← (By using Path compression)

This is the resultant tree of doing final union.

Code:

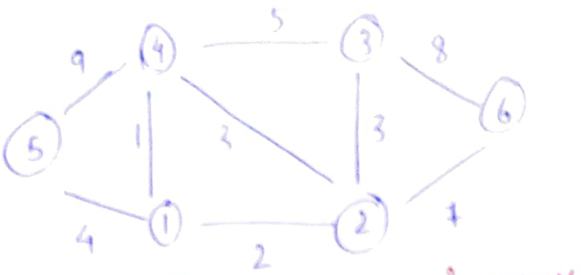
$$\begin{cases} T.C \rightarrow O(4\alpha) \approx O(4) \leftarrow \text{constant} \\ S.C \rightarrow O(n) \end{cases}$$

```
int parent [10000];
int rank [10000];

void makSet () {
    for (i=1; i<=n; i++) {
        parent [i] = i;
        rank[i] = 0;
    }
}
int findPar (int node) {
    if (node = parent [node])
        return node
    else
        return parent [node] = findPar (parent [node]);
}
void union (int u, int v) {
    u = findPar (u);
    v = find Par (v);
    if (rank [u] < rank [v])
        parent [u] = v;
    else if (rank [u] > rank [v])
        parent [v] = u;
    else {
        parent [v] = u;
        rank[u]++;
    }
}
int main() {
    makeSet();
    int m; cin >>m;
    while (m--) {
        int u, v;
        cin >> u>>v;
        union (u,v);
    }
    if (FindPar(2) != find Par (3)) → Not in same comp.
    else ← In the Same comp.
```

• Kruskal's Algorithm

Here, we will not use Adj list but a linear data structure and sort all the edges acc. to their weights



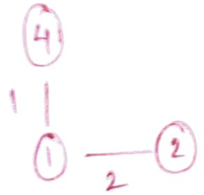
wt	u	v
1	1	4
2	1	2
3	2	3
3	2	4
4	1	5
5	3	4
7	2	6
8	3	6
9	4	5

(i) check if 1 and 4 are in same component or not

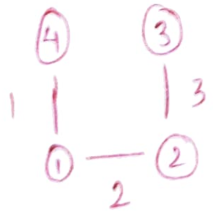
so, Its not in same comp.

Approach: Do this for all the edges, check for the same component or not, if they are, then do not add that edge and node in MST, move forward to next nodes else add that edge and node to MST.

(ii) 1 and 2



(iii) 2 and 3



(iv) 2 and 4

Now, they have same parent (1)

Hence, they are in the same component, so dont add this in MST.

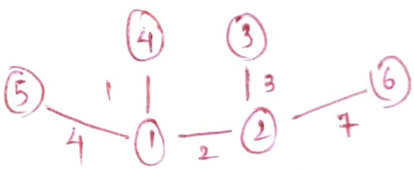
(v) 1 and 5

~~Dont Add~~

(vi) 3 and 4

Dont Add.

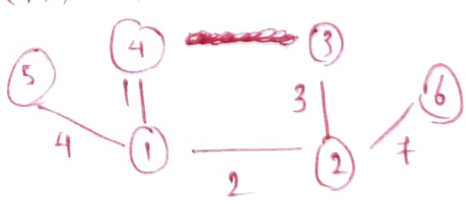
(vii) 2 and 6



(viii) 3 and 6

Don't Add

(ix) 4 and 5 → Don't Add



Total Min Cost = 20.

```
struct node {
    int u, v, wt;
    node (int x, int y, int z) {
        u = x;
        v = y;
        wt = z;
    }
};

int main() {
    vector<node> edges;
    int n, m;   <- n = nodes
                   m = edges
    for (i -> m) {
        int u, v, wt;
        cin >> u, >> v >> wt;
        edges.push_back(node(u, v, wt));
    }
    sort(edges.begin(), edges.end(), comp);
    vector<int> parent[N];
    for (i -> N)
        parent[i] = i;
    vector<int> rank (N, 0);
    int cost = 0;
    vector<pair<int, int>> mst;
    for (auto it : edges) {
        if (findPar(it.v, parent) != findPar(it.u, parent)) {
            cost += it.wt;
            mst.push_back({it.u, it.v});
            union(it.u, it.v, parent, rank);
        }
    }
    cout << cost;
    for (auto it : mst) {
        cout << it.first << " - " << it.second;
    }
}

bool comp(node a, node b) {
    return a.wt < b.wt;
}
```

$T.C \rightarrow O(M \log M) + O(M \times O(4\alpha)) \simeq O(M \log M)$

$S.C \rightarrow O(M) + O(N) + O(N) \simeq O(N).$