MEDIUM

Disjoint Set

Intuition

Why is a disjoint set used ?

Suppose that there are 2 components in a graph and we have to check if the given node belongs to component1 or not, then we can find the search for connected components. It is the brute force approach and it will take O(N+E) time complexity.

Using a Disjoint set we can do the O(1) complexity

It gives us two options

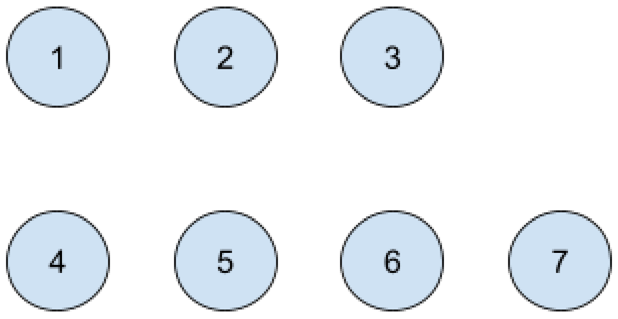
findParent()

Union() -> rank | size

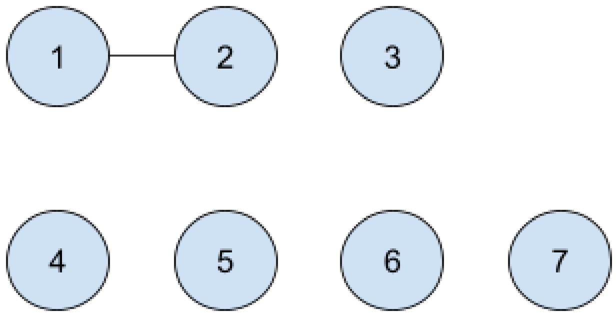
Dynamic Graphs : Initially we assume all the nodes are along components before the edge formation

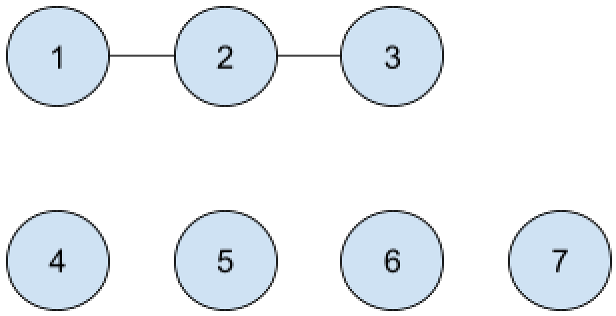
Eg.

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



What union() does is to build the graph ie. connect the edges of the graph





….

Until completely connected.

But at any stage we might end up with a query then in that case the disjoint set is useful. Eg.
Does 1,4 are from the same component ?

UNION( ) :

The union can be implemented in two ways as per the algorithm, by rank or by size

UNION BY RANK ( ) :

Requirements are :
- Rank vector
- Parent vector

Rank is initialized with 0 and parent is initialized with same node

Eg.
Rank Vector

1	2	3	4	5	6	7
1	0	0	2	0	1	0

Parent vector

1	2	3	4	5	6	7
1	1	4	4	4	4	6

Ie. all are their own parents

union (u,v) :
- Find ultimate parent of u and v
- Find rank of ultimate parent
- Connect smaller rank to larger rank always

Eg.

1,2

Pu = 1 pv = 2
Ru = 0 rv = 0

Connect anyone to anyone
u—->v : 1 — 2 is now a connected edge

Change parent(2) = 1
Increment rank(1) +=1

2,3

```
1         3
1         0
```
parent(3) = 2

4,5

```
4         5
0         0
```
parent(5) = 4
rank(4)+=1

6,7

```
6         7
0         0
```

parent(7) = 6
rank(6)+=1

5	6
4	6
1	1

parent(6) = 4
rank(4)+=1

3	7
1	4
1	2

parent(3) = 4

FindParent( ) :

- Return the ultimate parent // log(n)

We will apply something as **path compression** as eg.

4-6-7 : 4 is parent(6) ,6 is parent(7) ;we can say that ultimate_parent(7) is 4 // we can change parent(7) : 4

5-4-3-2-1 : returns 1-1-1-1-1 // 1 as parent this is path compression.

But the rank cannot be decreased during the path compression, because suppose we compressed one section of tree while the other is remaining as it is.

```
findParent(u):
    If u == parent(u) :
        Return u
    Return parent[u]=findParent(parent(u))
```

On attaching to the smaller component we run into problem for the situation as the height increases and thus we need to travel a longer distance in order to traverse

UNION BY SIZE ( ) :

We will be taking a size vector instead of the rank vector here

Initial Configuration :

Size :

1	2	3	4	5	6	7
1	1	1	1	1	1	1

Parent :

1	2	3	4	5	6	7
1	2	3	4	5	6	7

Now we will be performing the Union by rank over the given edges

1,2 2,3 4,5 6,7 5,6 3,7

Performing Union By rank :

Size :

1	2	3	4	5	6	7
3	1	1	7	1	2	1

Parent :

1	2	3	4	5	6	7
1	1	4	4	4	4	6

1 2
1 1
1<-2
size(1)+=1
par(2) = 1

2 3
2 1

```
1<-3
size(1)+=1
par(3)=1

4       5
1       1
4<-5
size(4)+=1
par(5)=4

6       7
1       1
6<-7
size(6)+=1
par(7) = 6

5       6
2       2
5<-6
size(4)+=size(6)
par(6)=4
3       7
3       4
4<-3
size(4)+=size(1)
par(3) = 4
```

Now path compression will take place during the findParent( ) similarly to union by rank( )

Approach

DisjointSet( ) // constructor :
- Resizing the rank vector according to the given size and initializing with 0
- Resizing the parent vector and initialize them with node itself as parents

FindParent( ) :
- If the parent[node] == node : return node
- Assign parent of the node to node
- Return the findParent recursive call to find the ultimate parent as return findParent(parent[node])

UnionByRank( ) :
- Calculate the ultimate parents of the node
- Check if both the ultimate parents are same :

        - Return // as both are the parts of the same components
    - Check if the u component parent rank is smaller :
        - Make parent[u] = v
    - Check if the v component parent rank is smaller :
        - Make parent[v] = u
    - Check if both parents are same :
        - Make u as parent[v]
        - Update rank of u by 1

UnionBySize( ):

```
- Calculate the ultimate parents of the nodes
- Check if both parents are same :
    - Return // as they belong to same components
- Check if u is smaller in size :
    - Update parent of u as v
    - Update size of v as v+=size(u)
- Check if v is smaller of equal in size :
    - Update parent of v as u
    - Update size of u as v+size(u)
```

Function Code

UNION BY RANK( ) :

```
#include<bits/stdc++.h>
using namespace std;
class DisjointSet
{
    vector<int> rank,parent;
    public:
        DisjointSet(int n)
        {   // assigning the rank as 0 to all elements
            rank.resize(n+1,0);
            // resizing the parent vector to store the n+1 nodes
            parent.resize(n+1);
            // traversing the store the itself components into the parent
vector
            for(int i=0;i<=n ;i++)
            {
                parent[i] = i;
            }
        }
        int findParent(int node)
        {
            if(parent[node]==node)
            {
                return node;
            }
            return parent[node] = findParent(parent[node]);
        }
        void unionByRank(int u,int v)
        {
            // finding the ultimate parents of the nodes
            int up = findParent(u);
            int vp = findParent(v);
            // if the nodes already belong to the same component
            if(up==vp)
            {
                return;
            }
            // checking if u rank is greater
            if(rank[up]<rank[vp])
            {
                parent[up] = vp;
            }
            else if(rank[vp]>rank[up])
            {
                parent[vp] = rank[up];
            }
            else
            {
                parent[up] = parent[vp];
                rank[up]+=1;
            }
        }
};

int main()
{
    DisjointSet ds(7);
    ds.unionByRank(1,2);
    ds.unionByRank(2,3);
    ds.unionByRank(4,5);
    ds.unionByRank(6,7);
    ds.unionByRank(5,6);
    // checking if 3 7 are from same component
    if(ds.findParent(3)==ds.findParent(7))
    {
        cout<<"same"<<'\n';
    }
    else
    {
        cout<<"not same"<<'\n';
    }
    ds.unionByRank(3,7);
    if(ds.findParent(3)==ds.findParent(7))
    {
        cout<<"same"<<'\n';
    }
    else
    {
        cout<<"not same"<<'\n';
    }
    return 0;

}
```

UNION BY SIZE ( ) :

```
#include <bits/stdc++.h>
using namespace std;
class DisjointSet {
    // creating parent vector and size vector
    vector<int> parent, size;
public:
    DisjointSet(int n) {
        // resizing parent and size vector
        parent.resize(n+1);
        size.resize(n+1);
        // making parent to itself and assigning size initially 1
        for(int i = 0;i<=n;i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }
// find parent using path compression
```

```cpp
    int findUPar(int node) {
        if(node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionBySize(int u, int v) {
        // finding ultimate parents
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        // checking if the ultimate parents are same ie. same components
        if(ulp_u == ulp_v) return;
        // checking whose size is smaller
        if(size[ulp_u] < size[ulp_v]) {
            // updating parent and size
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        // else updating the parent and size
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
int main() {
    DisjointSet ds(7);
    ds.unionBySize(1, 2);
    ds.unionBySize(2, 3);
    ds.unionBySize(4, 5);
    ds.unionBySize(6, 7);
    ds.unionBySize(5, 6);
    // if 3 and 7 same or not
    if(ds.findUPar(3) == ds.findUPar(7)) {
        cout << "Same\n";
    }
    else cout << "Not same\n";

    ds.unionBySize(3, 7);

    if(ds.findUPar(3) == ds.findUPar(7)) {
        cout << "Same\n";
    }
    else cout << "Not same\n";
      return 0;
}
```

Time Complexity

O(1) // both by using rank as well as size