Disjoint Set Data Structure

Why we need a Disjoint Set data structure ?

Let’s understand the need of disjoint set data structure with an example.

Suppose there is a graph having multiple components (given below) and as we know that a graph may contains multiple components.

The above graph contains 2 components.

Now, if I ask you a question that tell me whether node ‘1’ and node ‘5’ are in the same component or not.

Now, in order to solve this question we can use either the DFS or BFS traversal technique like if we traverse the components of the graph we can find that node 1 and node 5 are not in the same component. This is actually the ***brute force*** approach whose time complexity is **O(N+E)***(N = no. of nodes, E = no. of edges).*

But ***using a Disjoint Set data structure we can solve this same problem in constant time i.e., O(1) time complexity***.

💡 The entire concept of disjoint set is used to find whether 2 arbitrary node belongs to the same component of graph or not in constant i.e., O(1) time complexity.

💡 The disjoint Set data structure is generally used for ***dynamic graphs***.

💡 The disjoint-set data structure is also known as the **Union-Find data structure.**

What is Dynamic Graphs ?

A dynamic graph is a graph that changes over time. It is a graph whose edges and/or vertices are added, removed or modified over time.

Basically, A dynamic graph is a graph that changes its configuration over time. It is a graph where edges and/or vertices can be added, removed, or modified frequently.

Example:

- Let's consider the edge information for the given graph as: {{1,2}, {2,3}, {4,5}, {6,7}, {5,6}, {3,7}}. Now if we start adding the edges one by one, in each step the structure of the graph will change. So, after each step, if we perform the same operation on the graph while updating the edges, the result might be different. In this case, the graph will be considered a dynamic graph.
- For example, after adding the first 4 edges if we look at the graph, we will find that node 4 and node 1 belong to different components but after adding all 6 edges if we search for the same we will figure out that node 4 and node 1 belong to the same component.

So, ***after any step, if we try to figure out whether two arbitrary nodes u and v belong to the same component or not, Disjoint Set will be able to answer this query in constant time.***

Functionalities of Disjoint Set data structure:

The disjoint set data structure generally provides two types of functionalities:

1. Finding the parent for a particular node (***findParent() or findPar() or findSet()***).
2. **Union**
    a. **Task of Union Method: If 2 arbitrary nodes belong to different components of a graph, then performing a Union operation on those nodes will merge the two components into a single component and this merge operation can be done in 2 ways, first is union by *rank* and second is union by *size*.**
    b. This Union method/functionality can be implemented in 2 ways,
        i. Union by **Rank.**
        ii. Union by **Size.**

So, Primarily Disjoint Set gives you two methods, find parent and union.

First, we will be discussing Union by rank and then Union by size.

Union by rank:

Before discussing Union by Rank, first, we need to discuss some terminologies that are used in Union By Rank.

Rank:

The rank of a node generally refers to the distance (the number of nodes including the leaf node) between the furthest leaf node and the current node. Basically rank includes all the nodes beneath the current node.

Here the rank of node 1 is 2 as the distance between node 1 and the furthest leaf node 4 is 2.

Ultimate parent:

The parent of a node generally refers to the node right above that particular node. But the ultimate parent refers to the topmost node or the root node.

In this graph, the parent of 8 is 5 but the ultimate parent of 8 is 4

<u>Now let's discuss the implementation of the union by rank function:</u>

In order to implement Union by rank, we basically need two arrays of size N (where N is the number of nodes).

1. One is the ***rank array:***
    a. The rank array basically stores the rank of each node and Initially the rank of all nodes is 0.
2. The other one is the ***parent array:***
    a. The parent array stores the ultimate parent for each node and initially, each node is the ultimate parent of itself.

Algorithm for Union by Rank:

So, the entire concept of disjoint set is used to find whether 2 arbitrary node belongs to the same component of the graph or not, in constant i.e., O(1) time complexity.

So, in input, we will be given the number of nodes in the graph (V) and the edges vector, now there is no need to create the entire graph at once because we will be building the graph through the union method so that at any point of time if someone asks you to tell whether 2 arbitrary nodes belongs to the same component of the graph or not, so can answer this question in constant time.

Example: Let's understand Union by rank with the help of an example and then we write the pseudocode of algorithm.

Assume we have a graph with 7 vertices and the edges vector is given below,

```
vector<vector<int>> edges = { {1, 2}, {2, 3}, {4, 5}, {6, 7}, {5, 6}, {3, 7} };
```

Initially, what we assume is, all the nodes is alone, it means edges between node are not created yet, and I already said that, in Disjoint Set Data structure, there is no need to create the entire graph at once because in Disjoint Set Data structure the union method is used for connecting disconnected components (Initially when edges are not created between node so we can say the all the vertices are alone and forming a individual component).

Let's perform Union by Rank,

We have seen the Initial configurations, Now let's quickly see the pseudocode for the Union operation and then we move toward performing the union**.**

Pseudocode for union(u, v):

Firstly, the Union function requires two nodes(***let's say u and v***) as arguments.

1. Find the ultimate parent of node 'u' and node 'v' and store the ultimate parents of both the node in a separate variable.

2. Find ranks of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent and we will increase the rank by one for the ultimate parent node to whom we have connected the other one.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

Now, let’s perform Union by Rank:

```
// Input Given.
int V = 7; // Total 7 vertices.
vector<vector<int>> edges = { {1, 2}, {2, 3}, {4, 5}, {6, 7}, {5, 6}, {3, 7} };
```

Initial Configuration:

Let’s start,

Firstly, the Union function requires two nodes(***let’s say u and v***) as arguments.

In this case, U = 1 and V = 2.

Let's Perform Union:

Union(1, 2):

Now, we have to perform Union of 1, 2 and to perfrom Union we have 3 steps:

1. Find the ultimate parent of node ‘u’ and node ‘v’ and store the ultimate parents of both the node in a separate variable.
2. Find ranks of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent and we will increase the rank by one for the ultimate parent node to whom we have connected the other one.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

UltimateParentOfU = 1 and UltimateParentOfV = 2. (See the parent array).
RankOfUltimateParentOfU = 0 and RankOfUltimateParentOfV = 0 (see the rank array).

Now the ranks of both the ultimate parents is equal, in this case we can connect any ultimate parent to other ultimate parent and we will increase the rank by one for the ultimate parent node to whom we have connected the other one.

Now, I am connecting 2 with 1 and after connecting node 2 with node 1, the graph would look like this,

Now, after making connections, we have to update the parent array and the rank array.

We have to increment the rank of node '1' by one, because we connect node '2' with node '1' and Now the ultimate parent of 2 is 1.

Rank array =	ignore	1	0	0	0	0	0	0
Parent array =	ignore	1	1 ~~2~~	3	4	5	6	7
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Ignore Index 0, because nodes are labelled from 1 to N.

In this case, U = 2 and V = 3.

Let's Perform Union:

Union(2, 3):

Now, we have to perform Union of 2, 3 and to perfrom Union we have 3 steps:

1. Find the ultimate parent of node ‘u’ and node ‘v’ and store the ultimate parents of both the node in a separate variable.
2. Find ranks of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent and we will increase the rank by one for the ultimate parent node to whom we have connected the other one.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

UltimateParentOfU = 1 and UltimateParentOfV = 3. (See the parent array).
RankOfUltimateParentOfU = 1 and RankOfUltimateParentOfV = 0 (see the rank array).

Now the ranks of both the ultimate parents is not equal, in this case we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

Connecting 3 with 1 (because node '1' has larger rank than 3) and after connecting node 3 with node 1, the graph would look like this,

Now, after making connections, we have to update the parent array and there is no need update the rank array because the height of node 1 is still 1.

Now the ultimate parent of node 3 is node 1.

Rank array =	ignore	1	0	0	0	0	0	0
Parent array =	ignore	1	1 ~~2~~	1 ~~3~~	4	5	6	7
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Ignore Index 0, because nodes are labelled from 1 to N.

In this case, U = 4 and V = 5.

Let's Perform Union:

Union(4, 5):

Now, we have to perform Union of ~~2, 3~~ 4, 5 and to perfrom Union we have 3 steps:

1. Find the ultimate parent of node 'u' and node 'v' and store the ultimate parents of both the node in a separate variable.
2. Find ranks of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent and we will increase the rank by one for the ultimate parent node to whom we have connected the other one.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

UltimateParentOfU = 4 and UltimateParentOfV = 5. (See the parent array).
RankOfUltimateParentOfU = 0 and RankOfUltimateParentOfV = 0 (see the rank array).

Now the ranks of both the ultimate parents is equal,in this case you can connect any ultimate parent to other ultimate parent and we will increase the rank by one for the ultimate parent node to whom we have connected the other one.

Now, I am connecting 5 with 4, and after connecting node 5 with node 4, the graph would look like this,

Now, after making connections, we have to update the parent array and the rank array.

We have to update the rank of node '4' by one, because we connect the node '5' with node ~~1~~ 4 and Now the ultimate parent of node 5 is node 4.

Rank array =	ignore	1	0	0	1	0	0	0
Parent array =	ignore	1	1 ~~2~~	1 ~~3~~	4	4 ~~5~~	6	7
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Ignore Index 0, because nodes are labelled from 1 to N.

In this case, U = 6 and V = 7.

Let's Perform Union:

Union(6, 7):

Now, we have to perform Union of 6, 7 and to perfrom Union we have 3 steps:

1. Find the ultimate parent of node ‘u’ and node ‘v’ and store the ultimate parents of both the node in a separate variable.
2. Find ranks of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent and we will increase the rank by one for the ultimate parent node to whom we have connected the other one.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

UltimateParentOfU = 6 and UltimateParentOfV = 7. (See the parent array).
RankOfUltimateParentOfU = 0 and RankOfUltimateParentOfV = 0 (see the rank array).

Now the ranks of both the ultimate parents is equal,in this case you can connect any ultimate parent to other ultimate parent and we will increase the rank by one for the ultimate parent node to whom we have connected the other one.

Now, I am connecting 7 with 6, and after connecting node 7 with node 6, the graph would look like this,

Now, after making connections, we have to update the parent array and the rank array.

We have to update the rank of node '6' by one, because we connect the node '7' with node '6' and Now the ultimate parent of node 7 is node 6.

Rank array =	ignore	1	0	0	1	0	1	0
Parent array =	ignore	1	1 ~~2~~	1 ~~3~~	4	4 ~~5~~	6	6 ~~7~~
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Ignore Index 0, because nodes are labelled from 1 to N.

In this case, U = 5 and V = 6.

Let's Perform Union:

Union(5, 6):

Now, we have to perform Union of 5, 6 and to perfrom Union we have 3 steps:

1. Find the ultimate parent of node 'u' and node 'v' and store the ultimate parents of both the node in a separate variable.
2. Find ranks of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent and we will increase the rank by one for the ultimate parent node to whom we have connected the other one.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

UltimateParentOfU = 4 and UltimateParentOfV = 6. (See the parent array).
RankOfUltimateParentOfU = 1 and RankOfUltimateParentOfV = 1 (see the rank array).

Now the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent and we will increase the rank by one for the ultimate parent node to whom we have connected the other one.

Now, I am connecting 6 with 4, and after connecting node 6 with node 4, the graph would look like this,

Now, after making connections, we have to update the parent array and the rank array.

We have to update the rank of node '4' by one, **basically when you merge 2 components of same rank then the height/rank will increase by one,** and Now the ultimate parent of node 6 is node 4.

Rank array =	ignore	1	0	0	1	0	1	0
Parent array =	ignore	1	1 ~~2~~	1 ~~3~~	4	4 ~~5~~	4 ~~6~~	6 ~~7~~
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Ignore Index 0, because nodes are labelled from 1 to N.

> 💡 Basically when you merge 2 nodes of the same rank then the height/rank will increase by one and when 2 nodes have different ranks in this case we will not do anything with the ranks.

Now, Suppose at this point, someone asks you to tell whether node 1 and node 7 belong to the same component or not. The answer is "No" but how will you find out ? and this is where something like 'findparent()' will help you.

Now, let's see how "findParent()" will help you solve this problem, Look at the parent array, the ultimate parent of node 1 is 1 and the ultimate parent of node 7 is 6, the ultimate parents of both the nodes are different, that's why we can say that node 1 and node 7 does not belongs to the same component.

> 💡 ***If 2 arbitrary nodes belong to the same component, then the ultimate parent of both nodes must be same.*** In a disjoint set data structure, all the nodes in the same component will have the same ultimate parent, which is also known as the representative or leader of the component.

> 💡 ***If 2 arbitrary nodes does not belong to the same component, then the ultimate parent of both the node must be different.***

findParent(int node) function:

This function actually takes a single node as an argument and finds the ultimate parent for each node.

Now, if we try to find the ultimate parent(typically using recursion) of each query separately, it will end up taking O(log(n)) time complexity for each case. But we want the operation to be done in a constant time. This is where the ***path compression technique*** comes in.

Using the ***path compression technique*** we can reduce the time complexity nearly to constant time.

What is path compression?

Basically, connecting each node in a particular path to its ultimate parent refers to path compression. Let's understand it using the following illustration:

How the time complexity reduces:

Before path compression, if we had tried to find the ultimate parent for node 4, we had to traverse all the way back to node 1 which is basically the height of size logN.

> 💡 The "height of size logN" refers to the maximum number of steps needed to traverse from a given node to its root, which is logarithmic in the number of nodes in the disjoint set data structure (N). In other words, without path compression, finding the root of a node could take up to logN steps, which is not efficient for large sets.

But after path compression, we can easily access the ultimate parent with a single step. Thus the traversal reduces and as a result the time complexity also reduces.

> 💡 **Note:** *We cannot change the ranks while applying path compression.*

Does path compression affects the rank of nodes?

We say that, Path compression ***does not affect*** the rank of nodes in the disjoint set data structure but it’s not 100 % correct.

Path compression only affects the parent pointers, not the rank array. The rank array is only used to keep track of the height of each tree or subtree, and is updated only during

the union operation when two trees of the same rank are merged. The rank array is not updated during the find operation, so path compression does not affect it.

Overall, findPar() method helps to reduce the time complexity of the ***union by the rank*** method as it can find the ultimate parent within constant time.

> 💡 But It's not 100% sure whether path compression will affect the rank of the nodes or not.

> 💡 The concept of rank is used to identify whether one component is greater or smaller than the other component. That’s it.

Algorithm for finding Ultimate Parent:

This process is done using the backtracking method.

The algorithm steps are as follows:

1. **Base case:**
    a. If the node and the parent of the node become the same, it will return the node.
    b. Suppose the ultimate parent is node 1 and in the parent array the ultimate parent of node 1 is also 1.
2. We will call the findPar() function for a node until it hits the base case and while backtracking we will update the parent of the current node with the returned value.

Note: The actual time complexity of union by rank and findPar() is O(4 alpha(n)), where alpha(n) is the inverse Ackermann function. Alpha(n) grows very slowly and is considered to be a constant for all practical purposes. So, we can consider the time complexity of union by rank and findPar() to be very close to O(1). *Now, this 4 term has a long mathematical derivation which is not required for an interview.*

```
#include <bits/stdc++.h>
using namespace std;

/* ----------------------- Disjoint Set Class. ------------------ */
class DisjointSet
```

```
{
private:
    /* Data Members of DisjointSet Class. */
    /* In order to implement Disjoint, we basically need two arrays of size N
      (where N is the number of nodes).
    1. One is the rank array:
        1. The rank array basically stores the rank of each node.
    2. The other one is the parent array:
        1. The parent array stores the ultimate parent for each node. */

    // The rank array basically stores the rank of each node.
    vector<int> rank;

    // The parent array stores the ultimate parent for each node.
    vector<int> parent;

public:
    /* Member Functions / Methods of DisjointSet Class. */
    /* Constructor of DisjointSet Class. */
    DisjointSet(int n /* Initially we have total 'n' nodes. */)
    {
        // Declaring rank array with size 'n+1', because this size will work for
        // both 0 and 1 based indexing graphs.
        // Initially rank of all nodes is 0.
        rank.resize(n + 1, 0);

        // Declaring parent array with size 'n+1', because this size will work for
        // both 0 and 1 based indexing graphs.
        parent.resize(n + 1);

        // Initially each node is a ultimate parent of itself.
        for (int i = 0; i <= n; i++)
        {
            parent[i] = i;
        }
    }

    /* Method to find the Ultimate Parent of a particular node. */
    // This is the code without path compression.
    int findUltimateParent_WithoutPathCompression(int node)
    {
        // Base Case
        if (node == parent[node] /* Because the ultimate parent of a
        ultimate parent is a ultimate parent itself. */)
        {
            // return ultimate parent.
            return node;
        }

        // else, backtrack towards the base case and find the ultimate parent
        // of the node in the path.
        return findUltimateParent_WithoutPathCompression(parent[node]);
    }
```

```
/* Method to find the Ultimate Parent of a particular node. */
// This is the code with path compression.
int findUltimateParent(int node)
{
    // Base Case
    if (node == parent[node] /* Because the ultimate parent of a
    ultimate parent is a ultimate parent itself. */)
    {
        // return ultimate parent.
        return node;
    }

    // else, backtrack towards the base case and find the ultimate
    // parent of the node in the path.
    int ultimateParent = findUltimateParent(parent[node]);

    // Connect all the nodes directly with the ultimate parents.
    parent[node] = ultimateParent;
}

/* Performing Union using Union by rank Method. */
void unionByRank(int u, int v)
{
    // Step 1: Find the ultimate parent of node ‘u’ and node ‘v’
    // and store the ultimate parents of both the node in a separate variable.
    int ultimateParentOfNodeU = findUltimateParent(u);
    int ultimateParentOfNodeV = findUltimateParent(v);

    // Additional Step.
    // If both the nodes already belongs to the same component,
    // you can simply return from here.
    if (ultimateParentOfNodeU == ultimateParentOfNodeV)
    {
        return;
    }

    // Step 2: Find ranks of ultimate parents.
    int rankOfultimateParentOfNodeU = rank[ultimateParentOfNodeU];
    int rankOfultimateParentOfNodeV = rank[ultimateParentOfNodeV];

    /* Step 3: Check the ranks of the ultimate parents,
        3.1. Now, if the ranks of both the ultimate parents is equal,
             in this case you can connect any ultimate parent to other ultimate
             parent and we will increase the rank by one for the ultimate parent
             node to whom we have connected the other one.
        3.2. But, if the ranks of both the ultimate parents are not equal,
             in this case, we always connect the ultimate parent with a smaller
             rank to the other ultimate parent with a larger rank.
            (Smaller rank waale ultimate parent ko larger rank waale ultimate
            parent se connect karo).
    */

    if (rankOfultimateParentOfNodeU < rankOfultimateParentOfNodeV)
    {
```

```
            // If the ranks of both the ultimate parents are not equal, in this case,
            // we always connect the ultimate parent with a smaller rank to the other
            // ultimate parent with a larger rank. (Smaller rank waale ultimate parent
            // ko larger rank waale ultimate parent se connect karo).
            parent[ultimateParentOfNodeU] = ultimateParentOfNodeV;

            // when 2 nodes have different ranks in this case we will not do anything
            // with the ranks.
        }
        else if (rankOfultimateParentOfNodeV < rankOfultimateParentOfNodeU)
        {
            // connect smaller rank with larger rank.
            parent[ultimateParentOfNodeV] = ultimateParentOfNodeU;
            // when 2 nodes have different ranks in this case we will not do anything
            // with the ranks.
        }
        else
        {
            // I am connecting node 'u' with 'v'.
            // Now, the ultimate parent of node 'u' is node 'v'.
            parent[ultimateParentOfNodeU] = ultimateParentOfNodeV;

            // And when you connect 2 nodes of the same rank then the height/rank
            // will increase by one and when 2 nodes have different ranks
            // in this case we will  not do anything with the ranks.

            // increment the rank of 'ultimateParentOfNodeV'.
            rank[ultimateParentOfNodeV]++;
        }
    }
};

int main(void)
{
    DisjointSet ds(7);
    ds.unionByRank(1, 2);
    ds.unionByRank(2, 3);
    ds.unionByRank(4, 5);
    ds.unionByRank(6, 7);
    ds.unionByRank(5, 6);
    ds.unionByRank(3, 7);

    // Let's check if 3, 7 belongs to the same component or not.
    if (ds.findUltimateParent(3) == ds.findUltimateParent(7))
    {
        cout << "Belongs to the same component.\n";
    }
    else{

        cout << "Does not Belongs to the same component.\n";
    }
    return 0;
}
```

Follow-up question:

In the union by rank method, why do we need to connect the smaller rank to the larger rank?

- Let’s understand it using the following example:

In this case, the traversal time to find the ultimate parent for nodes 3, 4, 5, 6, 7, and 8 increases and so the path compression time also increases. But if we do the following

- the traversal time to find the ultimate parent for only nodes 1 and 2 increases. So the path compression time becomes relatively lesser than in the previous case. So, we can conclude that we should always connect a smaller rank to a larger one with the goal of
    - ***shrinking the height of the graph.***
    - ***reducing the time complexity as much as we can.***

Union by Size:

Until now, we have learned union by rank, the findParent() function, and the path compression technique.

Now, if we again carefully observe, after applying path compression it may be possible that the rank of the node gets changed.

Although, the The concept of rank is just used to identify whether one component is greater or smaller than the other component. Rank is just used for this purpose, that's it.

Now, can we use something like 'size', basically 'size of the component', which tells us the size of the component and we can compare two components based of their sizes, using this we can easily identify which component is greater or which component is smaller.

> 💡 In the context of the Disjoint Set data structure, the ***size of a component*** usually means the number of nodes in the component.

So, here comes the concept of ***Union by size***.

Union by size:

This is as same as the Union by rank method except this method uses the size to compare the components while connecting. That is why we need a ***'size'*** array of size N(no. of nodes) instead of a ***rank*** array. The size array will be storing the size for each particular node i.e. size[i] will be the size of the component starting from node i.

Typically, the size of a node refers to the number of nodes that are connected to it.

Let's perform Union by Size:

```
// Input Given.
int V = 7; // Total 7 vertices.
vector<vector<int>> edges = {
                                {1, 2},
                                {2, 3},
                                {4, 5},
                                {6, 7},
                                {5, 6},
```

```
                                {3, 7},
};
```

Initial Configuration:

We have seen the Initial configurations, Now let's quickly see the pseudocode for the Union operation and then we move toward performing the union**.**

Pseudocode for union(u, v):

Firstly, the Union function requires two nodes(***let's say u and v***) as arguments.

1. Find the ultimate parent of node 'u' and node 'v' and store the ultimate parents of both the node in a separate variable.
2. Find size of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

> 💡 Now, in case of Union by rank, we only increment the rank, when we connect 2 components of same sizes, but in case of Union by Size While connecting in both cases we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

Let’s start,

Firstly, the Union function requires two nodes(***let’s say u and v***) as arguments.

In this case, U = 1 and V = 2.

<u>Let's Perform Union:</u>

Union(1, 2):

Now, we have to perform Union of 1, 2 and to perfrom Union we have 3 steps:

1. Find the ultimate parent of node ‘u’ and node ‘v’ and store the ultimate parents of both the node in a separate variable.
2. Find size of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

Note: Now, in case of Union by rank, we only increment the rank, when we connect 2 components of same sizes, but in case of Union by Size While connecting in both cases we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

UltimateParentOfU = 1 and UltimateParentOfV = 2. (See the parent array).
sizeOfUltimateParentOfU = 0 and sizeOfUltimateParentOfV = 0 (see the size array).

Now the size of both the ultimate parents is equal, in this case we can connect any ultimate parent to other ultimate parent and we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

Now, I am connecting 2 with 1 and after connecting node 2 with node 1, the graph would look like this,

Now, after making connections, we have to update the parent array and the size array.

We have to increment the size of node '1' with the size of node '2', because we connect node '2' with node '1' and Now the ultimate parent of 2 is 1.

Size array =	ignore	2	1	1	1	1	1	1
Parent array =	ignore	1	1 ~~2~~	3	4	5	6	7
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Ignore Index 0, because nodes are labelled from 1 to N.

In this case, U = 2 and V = 3.

Let's Perform Union:

Union(2, 3):

Now, we have to perform Union of 2, 3 and to perfrom Union we have 3 steps:

1. Find the ultimate parent of node ‘u’ and node ‘v’ and store the ultimate parents of both the node in a separate variable.
2. Find size of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

Note: Now, in case of Union by rank, we only increment the rank, when we connect 2 components of same sizes, but in case of Union by Size While connecting in both cases we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

UltimateParentOfU = 1 and UltimateParentOfV = 3. (See the parent array).
sizeOfUltimateParentOfU = 2 and sizeOfUltimateParentOfV = 1 (see the size array).

Now the size of both the ultimate parents is not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

Connecting 3 with 1 and after connecting node 3 with node 1, the graph would look like this,

Now, after making connections, we have to update the parent array and the size array.

We have to increment the size of node '1' with the size of node '3', because we connect node '3' with node '1' and Now the ultimate parent of 3 is 1.

Size array =	ignore	3	1	1	1	1	1	1
Parent array =	ignore	1	1 ~~2~~	1 ~~3~~	4	5	6	7
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Ignore Index 0, because nodes are labelled from 1 to N.

In this case, U = 4 and V = 5.

Let's Perform Union:

Union(4, 5):

Now, we have to perform Union of 4, 5 and to perfrom Union we have 3 steps:

1. Find the ultimate parent of node ‘u’ and node ‘v’ and store the ultimate parents of both the node in a separate variable.
2. Find size of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

Note: Now, in case of Union by rank, we only increment the rank, when we connect 2 components of same sizes, but in case of Union by Size While connecting in both cases we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

UltimateParentOfU = 4 and UltimateParentOfV = 5. (See the parent array).
sizeOfUltimateParentOfU = 1 and sizeOfUltimateParentOfV = 1 (see the size array).

Now the size of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent.

I am Connecting 5 with 4 and after connecting node 5 with node 4, the graph would look like this,

Now, after making connections, we have to update the parent array and the size array.

We have to increment the size of node '4' with the size of node '5', because we connect node '5' with node '4' and Now the ultimate parent of 5 is 4.

Size array =	ignore	3	1	1	2	1	1	1
Parent array =	ignore	1	1 ~~2~~	1 ~~3~~	4	4 ~~5~~	6	7
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Ignore Index 0, because nodes are labelled from 1 to N.

In this case, U = 6 and V = 7.

Let's Perform Union:

Union(6, 7):

Now, we have to perform Union of 6, 7 and to perfrom Union we have 3 steps:

1. Find the ultimate parent of node ‘u’ and node ‘v’ and store the ultimate parents of both the node in a separate variable.
2. Find size of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

Note: Now, in case of Union by rank, we only increment the rank, when we connect 2 components of same sizes, but in case of Union by Size While connecting in both cases we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

UltimateParentOfU = 6 and UltimateParentOfV = 7. (See the parent array).
sizeOfUltimateParentOfU = 1 and sizeOfUltimateParentOfV = 1 (see the size array).

Now the size of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent.

I am Connecting 7 with 6 and after connecting node 7 with node 6, the graph would look like this,

Now, after making connections, we have to update the parent array and the size array.

We have to increment the size of node '6' with the size of node '7', because we connect node '7' with node '6' and Now the ultimate parent of 7 is 6.

Size array =	ignore	3	1	1	2	1	2	1
Parent array =	ignore	1	1 ~~2~~	1 ~~3~~	4	4 ~~5~~	6	6 ~~7~~
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Ignore Index 0, because nodes are labelled from 1 to N.

In this case, U = 5 and V = 6.

Let's Perform Union:

Union(5, 6):

Now, we have to perform Union of 5, 6 and to perfrom Union we have 3 steps:

1. Find the ultimate parent of node ‘u’ and node ‘v’ and store the ultimate parents of both the node in a separate variable.
2. Find size of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

Note: Now, in case of Union by rank, we only increment the rank, when we connect 2 components of same sizes, but in case of Union by Size While connecting in both cases we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

UltimateParentOfU = 4 and UltimateParentOfV = 6. (See the parent array).
sizeOfUltimateParentOfU = 2 and sizeOfUltimateParentOfV = 2 (see the size array).

Now the size of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent.

I am Connecting 6 with 4 and after connecting node 6 with node 4, the graph would look like this,

Now, after making connections, we have to update the parent array and the size array.

We have to increment the size of node '4' with the size of node '6', because we connect node '6' with node '4' and Now the ultimate parent of 6 is 4.

Size array =	ignore	3	1	1	4	1	2	1
Parent array =	ignore	1	1 ~~2~~	1 ~~3~~	4	4 ~~5~~	4 ~~6~~	6 ~~7~~
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Ignore Index 0, because nodes are labelled from 1 to N.

In this case, U = 3 and V = 7.

Let's Perform Union:

Union(3, 7):

Now, we have to perform Union of 3, 7 and to perfrom Union we have 3 steps:

1. Find the ultimate parent of node ‘u’ and node ‘v’ and store the ultimate parents of both the node in a separate variable.
2. Find size of ultimate parents.
3. Now, if the ranks of both the ultimate parents is equal, in this case you can connect any ultimate parent to other ultimate parent.
4. But, if the ranks of both the ultimate parents are not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

Note: Now, in case of Union by rank, we only increment the rank, when we connect 2 components of same sizes, but in case of Union by Size While connecting in both cases we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

UltimateParentOfU = 1 and UltimateParentOfV = 4 (after path compression). (See the parent array).

sizeOfUltimateParentOfU = 3 and sizeOfUltimateParentOfV = 4 (see the size array).

Now the size of both the ultimate parents is not equal, in this case, we always connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. (Smaller rank waale ultimate parent ko larger rank waale ultimate parent se connect karo).

Connecting 1 with 4 and after connecting node 1 with node 4, the graph would look like this,

Now, after making connections, we have to update the parent array and the size array.

We have to increment the size of node '4' with the size of node '1', because we connect node '1' with node '4' and Now the ultimate parent of 1 is 4.

Size array =	ignore	3	1	1	7	1	2	1
Parent array =	ignore	~~4 1~~	~~1 2~~	~~1 3~~	4	~~4 5~~	~~4 6~~	~~4 7~~
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Ignore Index 0, because nodes are labelled from 1 to N.

Note: *It seems much more intuitive than union by rank as the rank gets distorted after path compression.*

Note: *The findPar() function remains the exact same as we have discussed earlier.*

```
// Union by size code.
// Rest of the code remains the same.
/* Performing Union using Union by size Method. */
    void unionBySize(int u, int v)
    {
        // Step 1: Find the ultimate parent of node ‘u’ and node ‘v’
        // and store the ultimate parents of both the node in a separate variable.
        int ultimateParentOfNodeU = findUltimateParent(u);
```

```
        int ultimateParentOfNodeV = findUltimateParent(v);

        // Additional Step.
        // If both the nodes already belongs to the same component,
        // you can simply return from here.
        if (ultimateParentOfNodeU == ultimateParentOfNodeV)
        {
            return;
        }

        // Step 2: Find size of ultimate parents.
        int sizeOfultimateParentOfNodeU = size[ultimateParentOfNodeU];
        int sizeOfultimateParentOfNodeV = size[ultimateParentOfNodeV];

        /* Step 3: Check the sizes of the ultimate parents,
            3.1. Now, if the ranks of both the ultimate parents is equal,
            in this case you can connect any ultimate parent to other ultimate parent.

            3.2 But, if the ranks of both the ultimate parents are not equal,
            in this case, we always connect the ultimate parent with a smaller rank
            to the other ultimate parent with a larger rank.
            (Smaller rank waale ultimate parent ko larger rank waale ultimate parent
             se connect karo).

            Note: Now, in case of Union by rank, we only increment the rank,
            when we connect 2 components of same sizes, but in case of Union by Size
            While connecting in both cases we will increase the size of the parent
            node to whom we have connected by the size of the other parent node
            which is actually connected.
        */

        if (sizeOfultimateParentOfNodeU < sizeOfultimateParentOfNodeV)
        {
            // If the ranks of both the ultimate parents are not equal,
            // in this case, we always connect the ultimate parent with a smaller rank
            // to the other ultimate parent with a larger rank.
            // (Smaller rank waale ultimate parent ko larger rank waale ultimate parent
            // se connect karo).
            parent[ultimateParentOfNodeU] = ultimateParentOfNodeV;

            // Update the size of "ultimateParentOfNodeV" with the size of
            // "ultimateParentOfNodeU".
            size[ultimateParentOfNodeV] += size[ultimateParentOfNodeU];
        }
        else
        {
            // There is no need for writing the separate case for equal,
            // because in we have to increment the size in all the cases.
            // connect smaller rank with larger rank.
            parent[ultimateParentOfNodeV] = ultimateParentOfNodeU;

            // Update the size of "ultimateParentOfNodeU" with the size of
            // "ultimateParentOfNodeV".
            size[ultimateParentOfNodeU] += size[ultimateParentOfNodeV];
```

```
        }
    }
```

Note: Time complexity of both union by size and union by rank is same i.e., O(4 alpha), which is basically a considered as Constant i.e., O(1) time complexity.