Disjoint Set Data Structures
Data Structures used are:
Array:
An array of integers is called Parent[]. If we are dealing with N items, i’ th element of the array represents the i’th item. More precisely, the i’th element of the Parent[] array is the parent of the i’th item. These relationships create one or more virtual trees.

Tree:
It is a Disjoint set. If two elements are in the same tree, then they are in the same Disjoint set. The root node (or the topmost node) of each tree is called the representative of the set. There is always a single unique representative of each set. A simple rule to identify a representative is if ‘i’ is the representative of a set, then Parent[i] = i. If i is not the representative of his set, then it can be found by traveling up the tree until we find the representative.

Operations:
Find:
Can be implemented by recursively traversing the parent array until we hit a node that is the parent of itself.

```
// Finds the representative of the set
// that i is an element of

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

int find(int i)

{

    // If i is the parent of itself
    if (parent[i] == i) {

        // Then i is the representative of
        // this set
        return i;
    }
    else {

        // Else if i is not the parent of
        // itself, then i is not the
        // representative of his set. So we
```

```
            // recursively call Find on its parent
            return find(parent[i]);
        }
}
```

It takes two elements as input and finds the representatives of their sets using the Find operation, and finally puts either one of the trees (representing the set) under the root node of the other tree, effectively merging the trees and the sets

```
// Unites the set that includes i
// and the set that includes j

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

void union(int i, int j) {

    // Find the representatives
    // (or the root nodes) for the set
    // that includes i
    int irep = this.Find(i),

    // And do the same for the set
    // that includes j
    int jrep = this.Find(j);

    // Make the parent of i’ s representative
    // be j’ s representative effectively
    // moving all of i’ s set into j’ s set)
    this.Parent[irep] = jrep;
}
```

Improvements (Union by Rank and Path Compression):

The efficiency depends heavily on the height of the tree. We need to minimize the height of tree in order to improve efficiency. We can use Path Compression and Union by rank method to do so.

Path Compression (Modifications to Find()):

It speeds up the data structure by compressing the height of the trees. It can be achieved by inserting a small caching mechanism into the Find operation. Take a look at the code for more details:

```
// Finds the representative of the set that i
// is an element of.

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

```
int find(int i)
{

    // If i is the parent of itself
    if (Parent[i] == i) {

        // Then i is the representative
        return i;
    }
    else {

        // Recursively find the representative.
        int result = find(Parent[i]);

        // We cache the result by moving i's node
        // directly under the representative of this
        // set
        Parent[i] = result;

        // And then we return the result
        return result;
    }
}
```

Union by Rank:

First of all, we need a new array of integers called rank[]. The size of this array is the same as the parent array Parent[]. If i is a representative of a set, rank[i] is the height of the tree representing the set.

Now recall that in the Union operation, it doesn't matter which of the two trees is moved under the other (see last two image examples above). Now what we want to do is minimize the height of the resulting tree. If we are uniting two trees (or sets), let's call them left and right, then it all depends on the rank of left and the rank of right.

If the rank of left is less than the rank of right, then it's best to move left under right, because that won't change the rank of right (while moving right under left would increase the height). In the same way, if the rank of right is less than the rank of left, then we should move right under left.

If the ranks are equal, it doesn't matter which tree goes under the other, but the rank of the result will always be one greater than the rank of the trees.

```
// Unites the set that includes i and the set
// that includes j
```

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

void union(int i, int j) {

    // Find the representatives (or the root nodes)
    // for the set that includes i
    int irep = this.find(i);

    // And do the same for the set that includes j
    int jrep = this.Find(j);

    // Elements are in same set, no need to
    // unite anything.
    if (irep == jrep)
        return;

    // Get the rank of i’s tree
    irank = Rank[irep],

    // Get the rank of j’s tree
    jrank = Rank[jrep];

    // If i’s rank is less than j’s rank
    if (irank < jrank) {

        // Then move i under j
        this.parent[irep] = jrep;
    }

    // Else if j’s rank is less than i’s rank
    else if (jrank < irank) {

        // Then move j under i
        this.Parent[jrep] = irep;
    }

    // Else if their ranks are the same
    else {

        // Then move i under j (doesn’t matter
```

```
            // which one goes where)
            this.Parent[irep] = jrep;

            // And increment the result tree's
            // rank by 1
            Rank[jrep]++;
        }
}
// C++ implementation of disjoint set

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

class DisjSet {
        int *rank, *parent, n;

public:

        // Constructor to create and
        // initialize sets of n items
        DisjSet(int n)
        {
                rank = new int[n];
                parent = new int[n];
                this->n = n;
                makeSet();
        }

        // Creates n single item sets
        void makeSet()
        {
                for (int i = 0; i < n; i++) {
                        parent[i] = i;
                }
        }

        // Finds set of given item x
        int find(int x)
        {
                // Finds the representative of the set
                // that x is an element of
```

```
	if (parent[x] != x) {

		// if x is not the parent of itself
		// Then x is not the representative of
		// his set,
		parent[x] = find(parent[x]);

		// so we recursively call Find on its parent
		// and move i's node directly under the
		// representative of this set
	}

	return parent[x];
}

// Do union of two sets represented
// by x and y.
void Union(int x, int y)
{
	// Find current sets of x and y
	int xset = find(x);
	int yset = find(y);

	// If they are already in same set
	if (xset == yset)
		return;

	// Put smaller ranked item under
	// bigger ranked item if ranks are
	// different
	if (rank[xset] < rank[yset]) {
		parent[xset] = yset;
	}
	else if (rank[xset] > rank[yset]) {
		parent[yset] = xset;
	}

	// If ranks are same, then increment
	// rank.
	else {
		parent[yset] = xset;
```

```
                rank[xset] = rank[xset] + 1;
            }
        }
};

// Driver Code
int main()
{

    // Function Call
    DisjSet obj(5);
    obj.Union(0, 2);
    obj.Union(4, 2);
    obj.Union(3, 1);

    if (obj.find(4) == obj.find(0))
        cout << "Yes\n";
    else
        cout << "No\n";
    if (obj.find(1) == obj.find(0))
        cout << "Yes\n";
    else
        cout << "No\n";

    return 0;
}
```

The time complexity of creating n single item sets is O(n). The time complexity of find() and Union() operations is O(log n). Hence, the overall time complexity of the Disjoint Set Data Structure is O(n + log n).

The space complexity is O(n) because we need to store n elements in the Disjoint Set Data Structure.