CHAPTER 9
DISJOINT SET

9.1 Learning Objectives

1. Students are able to understand the disjoint set data structure.
2. Students are able to solve problems using the disjoint set data structure.

9.2 Material

9.2.1 Disjoint Set

Let's start with the following problem: We have some number of items. We are allowed to merge any two items to consider them equal. At any point, we are allowed to ask whether two items are considered equal or not. The suitable data structure to solve the problem effectively is the disjoint set.

A disjoint–set is a data structure that keeps track of a set of elements partitioned into several disjoint (non-overlapping) subsets. In other words, a disjoint set is a group of sets where no item can be in more than one set. It is also called a **union–find data structure**, as it supports union and find operation on subsets. Let's begin by defining the operations:

- ***Find***: It determines which subset a particular element is in and returns the representative of that particular set. An item from this set typically acts as a "representative" of the set.
- ***Union***: It merges two different subsets into a single subset, and the representative of one set becomes representative of another.

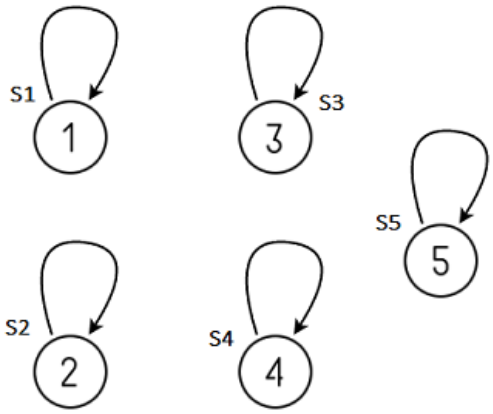
We can determine whether two elements are in the same subset by comparing the result of two Find operations. If the two elements are in the same set, they have the same representation; otherwise, they belong to different sets. If the union is called on two elements, merge the two subsets to which the two elements belong.

9.2.2 Implementation

Disjoint–set forests are data structures where each set is represented by a tree data in which each node holds a reference to its parent and the representative of each set is the root of that set's tree.

- ***Find*** follows parent nodes until it reaches the root.
- ***Union*** combines two trees into one by attaching one tree's root into the root of the other.

For example, consider five disjoint sets S1, S2, S3, S4, and S5 represented by a tree, as shown below diagram. Each set initially contains only one element each, so their parent pointer points to itself.



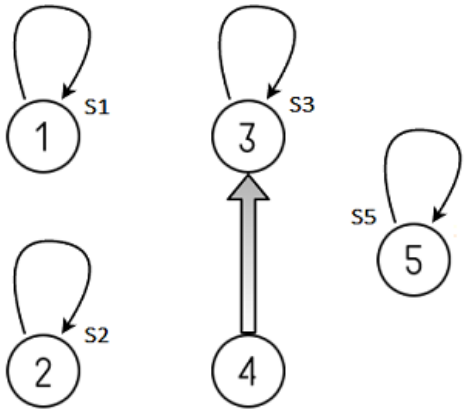
S1 = {1}, S2 = {2}, S3 = {3}, S4 = {4} and S5 = {5}.

The ***Find*** operation on element i will return representative of Si, where 1 <= i <= 5, i.e., Find(i) = i.

If we perform Union(S3, S4), S3 and S4 will be merged into one disjoint set, S3. Now,

S1 = {1}, S2 = {2}, S3 = {3, 4} and S5 = {5}.

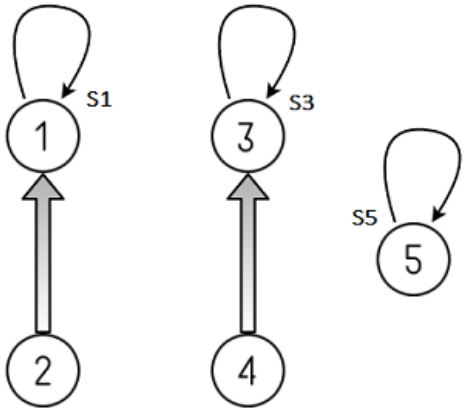
Find(4) will return representative of set S3, i.e., Find(4) = 3.



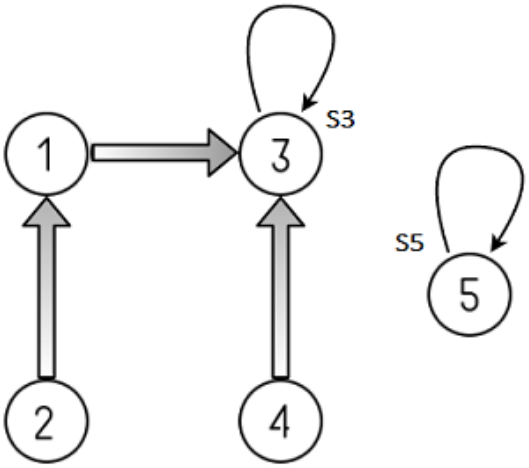
If we perform Union(S1, S2), S1 and S2 will be merged into one disjoint set, S1. Now,

S1 = {1, 2}, S3 = {3, 4} and S5 = {5}.

Find(2) or Find(1) will return the representative of set S1, i.e., Find(2) = Find(1) = 1.



If we perform Union(S3, S1), S3 and S1 will be merged into one disjoint set, S3. Now, S3 = {1, 2, 3, 4} and S5 = {5}.



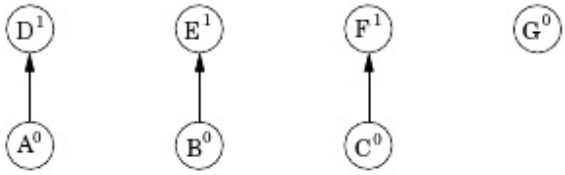
The naive implementation may result in highly unbalanced tree; however, we can enhance it in two steps:

1. **Union by rank**

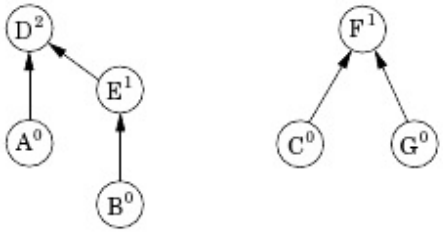
   This is to always attach the smaller tree to the root of the larger tree. Since it is the depth of the tree that affects the running time, the tree with a smaller depth gets added under the root of the deeper tree, which only increases the depth if the depths were equal. Single element trees are defined to have a rank of zero, and whenever two trees of the same rank r are united, the result has the rank of r+1. The worst-case running-time improves to O(log(n)) for the Union or Find operation. The following is the illustration of union-by-rank technique.

After $\texttt{makeset}(A), \texttt{makeset}(B), \ldots, \texttt{makeset}(G)$:

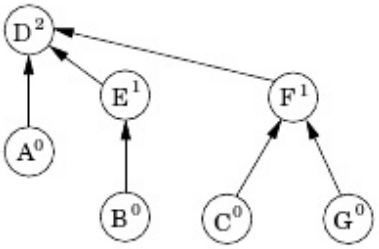
After $\texttt{union}(A, D), \texttt{union}(B, E), \texttt{union}(C, F)$:



After $\texttt{union}(C, G), \texttt{union}(E, A)$:

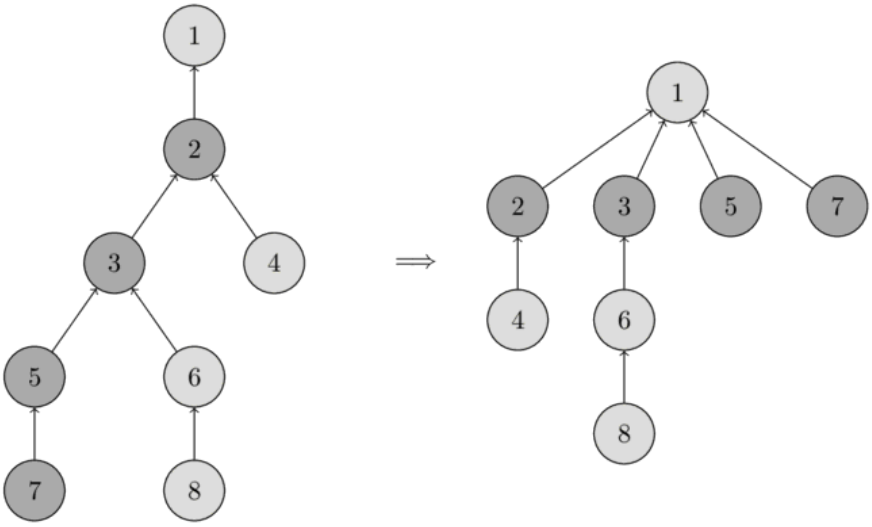


After $\texttt{union}(B, G)$:



2. Path compression

This is a way of flattening the tree's structure whenever Find is used on it. The idea is that each node visited heading to a root node may as well be attached directly to the root node; they all share the same representative. To affect this, as Find recursively traverses up the tree, it changes each node's parent reference to point to the root that is found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them, directly or indirectly. You can see the operation in the following image. On the left there is a tree, and on the right side there is the compressed tree after calling find(7), which shortens the paths for the visited nodes 7, 5, 3 and 2.



These two techniques complement each other, and running time per operation is effectively a small constant.

9.2.3 Java Implementation

The first is to implement the **Set** class, representing a set in the disjoint-set data structure.

```
1   public class Set{
2       private int parent;
3       private int rank;
4
5       public Set(int data){
6           this.parent = data;
7           this.rank = 0;
8       }
9
10      public int getParent(){
11          return this.parent;
12      }
13
14      public void setParent(int parent){
15          this.parent = parent;
16      }
17
18      public int getRank(){
19          return this.rank;
20      }
21
22      public void setRank(int rank){
23          this.rank = rank;
24      }
25  }
```

The Set class has two attributes, parent and rank. The parent attribute is the representative of a set if the set has only one node, and parent's node if the set has more than one node, and the rank attribute is to facilitate the union-by-rank process.

The next is to implement the **DisjointSet** class. It has find and union methods. The find method is using the path compression technique and the union method is using union-by-rank technique.

```java
public class DisjointSet {
    private Set[] sets;
    private int sz;

    public DisjointSet(int numItem){
        this.sz = numItem;
        this.sets = new Set[sz + 1];
        for (int i=1 ; i<=this.sz ; i++){
            this.sets[i] = new Set(i);
        }
    }

    public int find(int item){
        int parent = this.sets[item].getParent();
        if (item == parent){
            return item;
        }
        else{
            parent = find(parent);
            this.sets[item].setParent(parent); // path compression
            return parent;
        }
    }
```

```java
    public boolean isSameSet(int firstItem, int secondItem){
        return find(firstItem) == find(secondItem);
    }

    public void union(int firstItem, int secondItem){ // union by rank
        int firstItemParent = find(firstItem);
        int secondItemParent = find(secondItem);

        if (firstItemParent != secondItemParent){
            int firstRank = this.sets[firstItemParent].getRank();
            int secondRank = this.sets[secondItemParent].getRank();

            if (firstRank < secondRank){
                this.sets[firstItemParent].setParent(secondItemParent);
            }
            else if (firstRank > secondRank){
                this.sets[secondItemParent].setParent(firstItemParent);
            }
            else{
                this.sets[secondItemParent].setParent(firstItemParent);
                this.sets[firstItemParent].setRank(firstRank + 1);
            }
        }
    }
```

```
50      public void print(){
51          for (int i=1 ; i<=this.sz ; i++){
52              System.out.println("Parent of " + i + " = " + find(i));
53          }
54      }
55
56      public void printRank(){
57          for (int i=1 ; i<=this.sz ; i++){
58              System.out.println("Rank of " + i + " = " + this.sets[i].getRank());
59          }
60      }
61  }
```

After implementing the Set and DisjointSet class, it is the time to implement the DisjointSetMain class, which is our main class.

```
1   public class DisjointSetMain {
2       public static void main(String[] args) {
3           DisjointSet disjointSet = new DisjointSet(5);
4           disjointSet.print();
5
6           disjointSet.union(3, 4);
7           System.out.println("After union 3 and 4:");
8           disjointSet.print();
9           disjointSet.printRank();
10
11          disjointSet.union(1, 2);
12          disjointSet.union(1, 3);
13
14          System.out.println("Final result:");
15          disjointSet.print();
16          disjointSet.printRank();
17      }
18  }
```

9.3 Assignments

1. Implement a method to compute the number of sets! For example, for a disjoint-set data structure with 5 data, then S1 = {1}, S2 = {2}, S3 = {3}, S4 = {4}, and S5 = {5}, hence the method will return **5**. After performing Union(S3, S4), then S1 = {1}, S2 = {2}, S3 = {3, 4} and S5 = {5}, and hence the method will return **4**.
2. Implement a method to compute the number of elements for each set! In the example in Problem 1, the method will write "S1 has 1 element, S2 has 1 element, S3 has 2 elements, S5 has 1 element".