Introduction to Disjoint Set (Union-Find Algorithm)

Two sets are called disjoint sets if they don't have any element in common. The disjoint set data structure is used to store such sets. It supports following operations:

- Merging two disjoint sets to a single set using Union operation.
- Finding a representative of a disjoint set using Find operation.
- Check if two elements belong to the same set or not. We mainly find representatives of both and check if they are the same.

Consider a situation with a number of persons and the following tasks to be performed on them:

- Add a new friendship relation, i.e. a person x becomes the friend of another person y i.e adding a new element to a set.
- Find whether individual x is a friend of individual y (direct or indirect friend)

Examples:

We are given 10 individuals say, a, b, c, d, e, f, g, h, i, j

Following are relationships to be added:
a <-> b
b <-> d
c <-> f
c <-> i
j <-> e
g <-> j

Given queries like whether a is a friend of d or not. We basically need to create following 4 groups and maintain a quickly accessible connection among group items:
G1 = {a, b, d}
G2 = {c, f, i}
G3 = {e, g, j}
G4 = {h}

QUESTION: Find whether x and y belong to the same group or not, i.e. to find if x and y are direct/indirect friends.

Operations on Disjoint Set Data Structures:

1. Find:
The task is to find a representative of the set of a given element. The representative is always the root of the tree. So we implement find() by recursively traversing the parent array until we hit a node that is root (parent of itself).

2. Union:
The task is to combine two sets and make one. 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.

```
Let there be 4 elements 0, 1, 2, 3

Initially, all elements are single element
subsets.
0 1 2 3

Do Union(0, 1)
   1   2   3
  /
 0

Do Union(1, 2)
     2   3
    /
   1
 /
0

Do Union(2, 3)
         3
        /
      2
     /
   1
 /
0
```

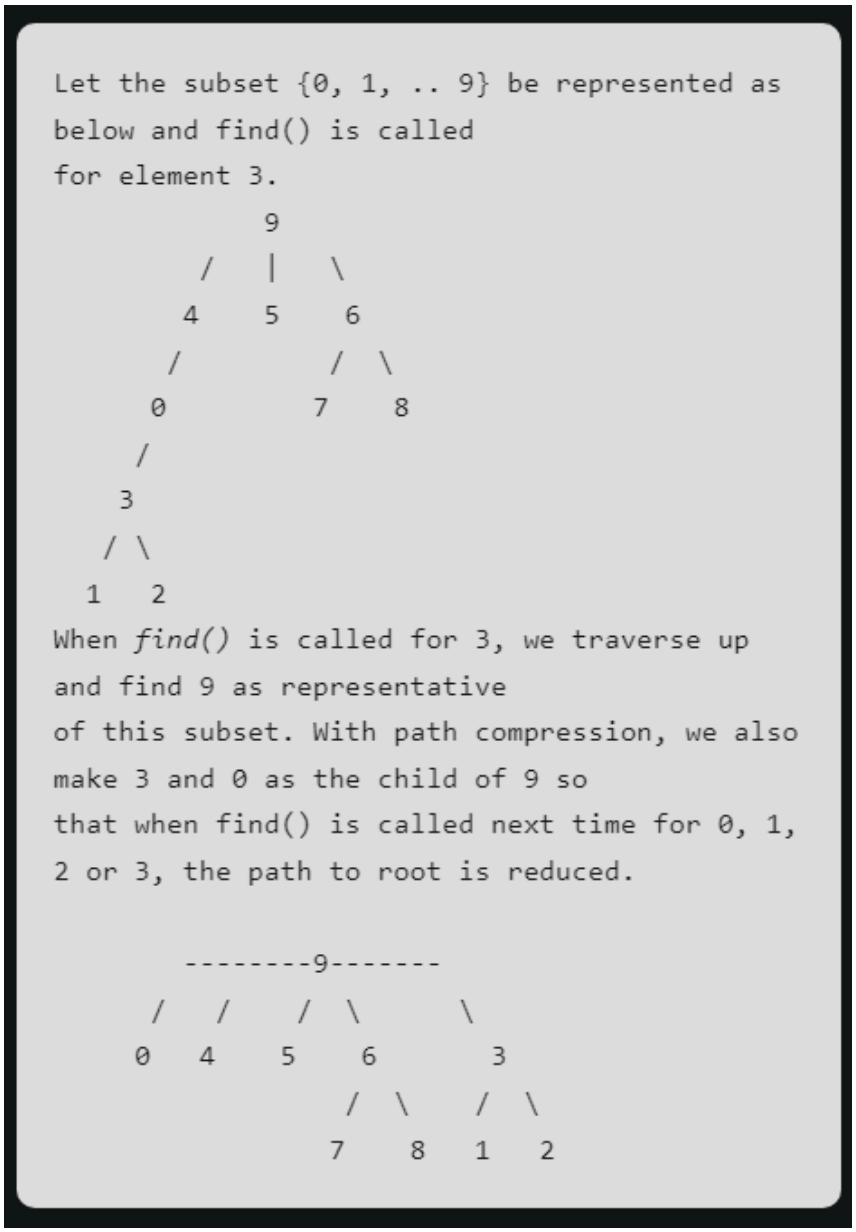
Optimizations (Path Compression and Union by Rank/Size):

The main idea is to reduce heights of trees representing different sets. We achieve this with two most common methods:

1) Path Compression
2) Union By Rank

1. **Path Compression (Used to improve find()):**

The idea is to flatten the tree when find() is called. When find() is called for an element x, the root of the tree is returned. The find() operation traverses up from x to find root. The idea of path compression is to make the found root as parent of x so that we don't have to traverse all intermediate nodes again. If x is the root of a subtree, then path (to root) from all nodes under x also compresses.



2. Union by Rank (Modifications to union())

Rank is like the height of the trees representing different sets. We use an extra 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 rank of the element i. Rank is the same as height if path compression is not used. With path compression, rank can be more than the actual height.

Now recall that in the Union operation, it doesn't matter which of the two trees is moved under the other. 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.