5.3. INTRODUCTION TO DISJOINT SET (UNION-FIND DATA STRUCTURE)

A **Disjoint Set Union (DSU)**, also called **Union-Find**, is a data structure commonly used to find **connected components** in an undirected graph efficiently.

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 representative of a disjoint set using **Find** operation.
- Check if two elements belong to same set or not. We mainly find representative of both and check if 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 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}

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

Partitioning the individuals into different sets according to the groups in which they fall. This method is known as a **Disjoint set Union** which maintains a collection of **Disjoint sets** and each set is represented by one of its members.

To answer the above question two key points to be considered are:

- **How to Resolve sets?** Initially, all elements belong to different sets. After working on the given relations, we select a member as a **representative**.
- **Check if 2 persons are in the same group?** If representatives of two individuals are the same, then they are friends.

Operations on Disjoint Set Data Structures:

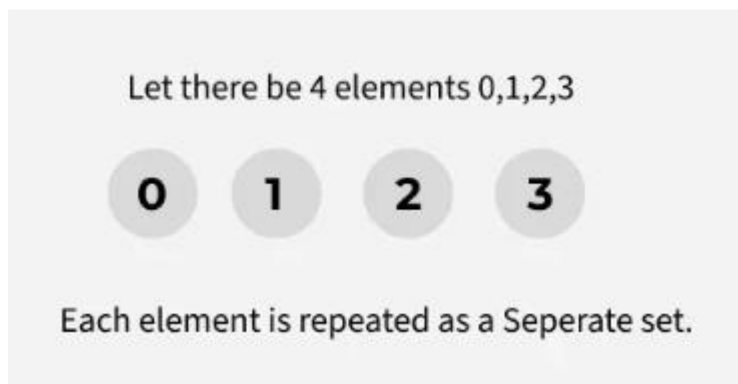
1. Find:

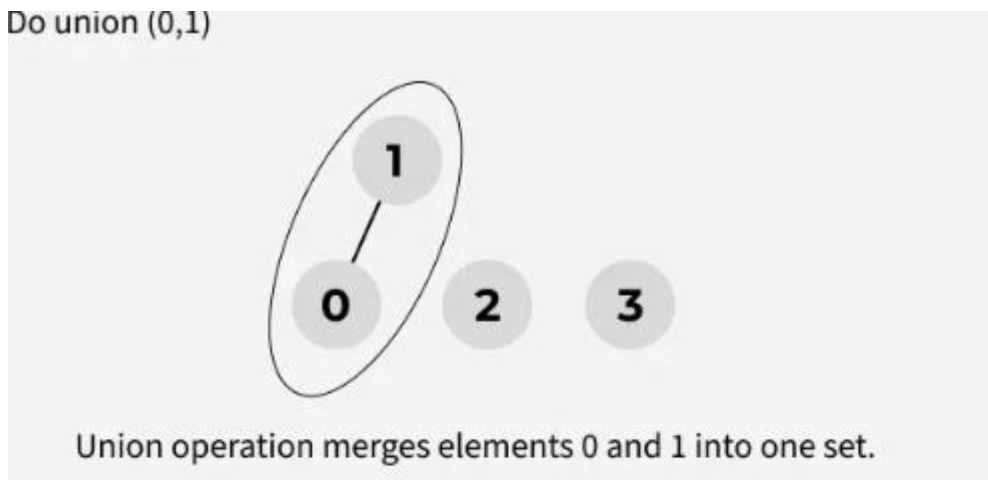
The task is to find representative of the set of a given element. The representative is always root of the tree. So we implement find() by recursively traversing the parent array until we hit a node that is root (parent of itself).

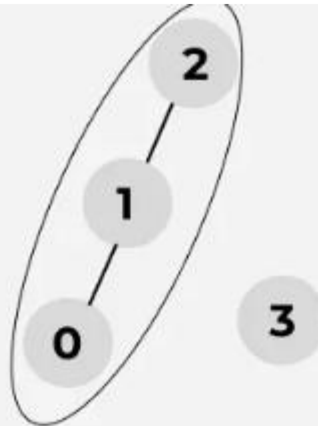
2. Union:

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.

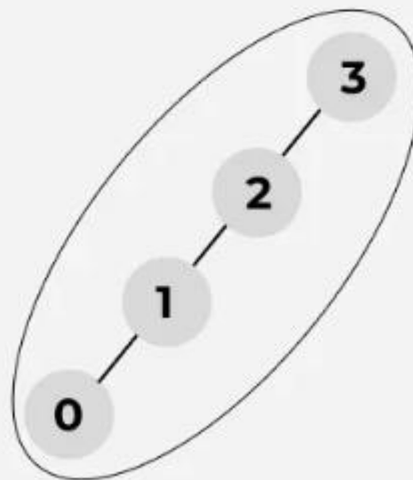
The above *union()* and *find()* are naive and the worst case time complexity is linear. The trees created to represent subsets can be skewed and can become like a linked list. Following is an example of worst case scenario.







Union (1,2) merges the set containing 0 and with element 2.



Union (2,3) merges the set containing 0 and 2 with element 3.

Disjoint Set Union

This article discusses the data structure **Disjoint Set Union** or **DSU**. Often it is also called **Union Find** because of its two main operations.

This data structure provides the following capabilities. We are given several elements, each of which is a separate set. A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is. The classical version also introduces a third operation, it can create a set from a new element.

Thus the basic interface of this data structure consists of only three operations:

- `make_set(v)` - creates a new set consisting of the new element `v`
- `union_sets(a, b)` - merges the two specified sets (the set in which the element `a` is located, and the set in which the element `b` is located)
- `find_set(v)` - returns the representative (also called leader) of the set that contains the element `v`. This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely

after `union_sets` calls). This representative can be used to check if two elements are part of the same set or not. `a` and `b` are exactly in the same set, if `find_set(a) == find_set(b)`. Otherwise they are in different sets.

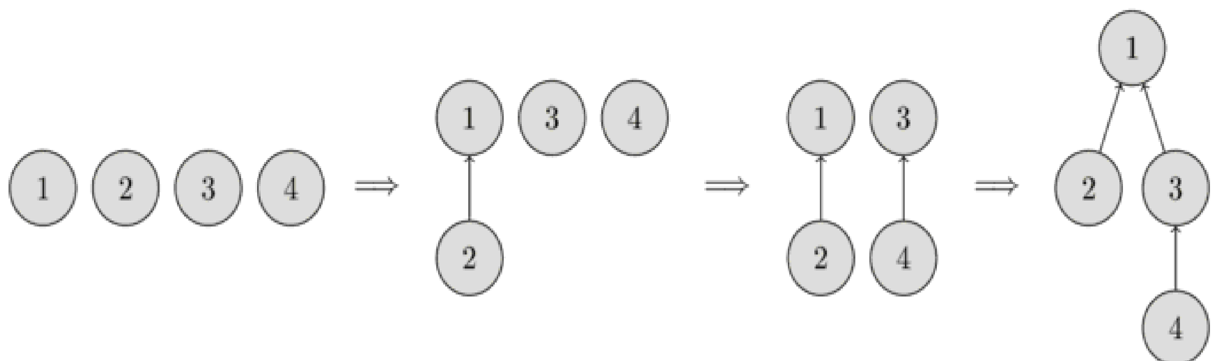
As described in more detail later, the data structure allows you to do each of these operations in almost time on average.

Also in one of the subsections an alternative structure of a DSU is explained, which achieves a slower average complexity of , but can be more powerful than the regular DSU structure.

Build an efficient data structure

We will store the sets in the form of **trees**: each tree will correspond to one set. And the root of the tree will be the representative/leader of the set.

In the following image you can see the representation of such trees.



In the beginning, every element starts as a single set, therefore each vertex is its own tree. Then we combine the set containing the element 1 and the set containing the element 2. Then we combine the set containing the element 3 and the set containing the element 4. And in the last step, we combine the set containing the element 1 and the set containing the element 3.

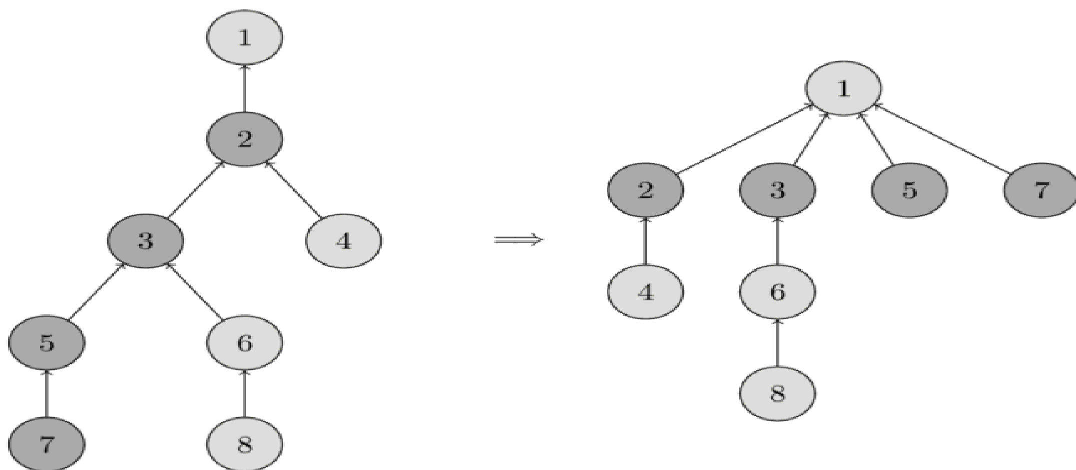
For the implementation this means that we will have to maintain an array `parent` that stores a reference to its immediate ancestor in the tree.

Path compression optimization

This optimization is designed for speeding up `find_set`.

If we call `find_set(v)` for some vertex `v`, we actually find the representative `p` for all vertices that we visit on the path between `v` and the actual representative `p`. The trick is to make the paths for all those nodes shorter, by setting the parent of each visited vertex directly to `p`.

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_set(7)`, which shortens the paths for the visited nodes 7, 5, 3 and 2.



The new implementation of `find_set` is as follows:

```
int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}
```

The simple implementation does what was intended: first find the representative of the set (root vertex), and then in the process of stack unwinding the visited nodes are attached directly to the representative.

This simple modification of the operation already achieves the time complexity per call on average (here without proof). There is a second modification, that will make it even faster.

Union by size / rank

In this optimization we will change the `union_set` operation. To be precise, we will change which tree gets attached to the other one. In the naive implementation the second tree always got attached to the first one. In practice that can lead to trees containing chains of length . With this optimization we will avoid this by choosing very carefully which tree gets attached.

There are many possible heuristics that can be used. Most popular are the following two approaches: In the first approach we use the size of the trees as rank, and in the second one we use the depth of the tree (more precisely, the upper bound on the tree depth, because the depth will get smaller when applying path compression).

In both approaches the essence of the optimization is the same: we attach the tree with the lower rank to the one with the bigger rank.

Here is the implementation of union by size:

```
void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}
```

And here is the implementation of union by rank based on the depth of the trees:

```
void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
```

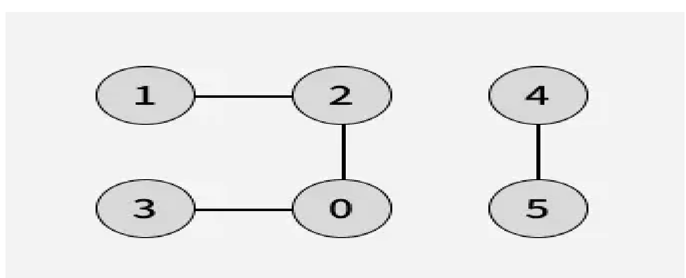
Both optimizations are equivalent in terms of time and space complexity. So in practice you can use any of them.

CONNECTED COMPONENTS IN AN UNDIRECTED GRAPH

Given an **undirected** graph(the graph may contain one or more components) represented by an adjacency list **adj[][]**, return all the connected components in any order.

Examples:

Input: *adj[][] = [[2, 3], [2], [0, 1], [0], [5], [4]]*



Output: *[[0, 1, 2, 3], [4, 5]]*
Explanation: *There are 2 different connected components. They are {0, 1, 2, 3} and {4, 5}.*
Input: *adj[][] = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]*