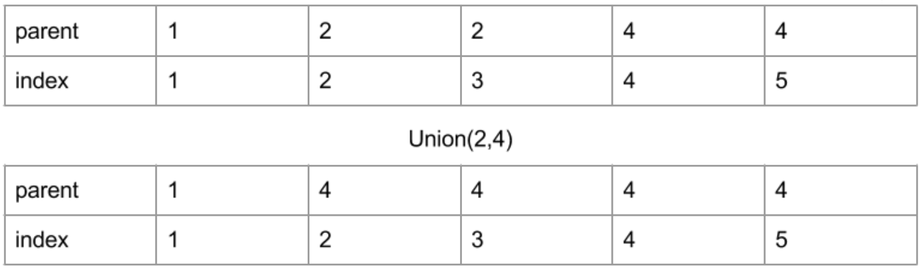
Disjoint Sets

1 The Disjoint Sets:Quick Union and Quick Find

One problem that comes up frequently in Computer Science, and the world in general, is "are two things connected?". This problem is found in social networks (think mutual friends), power grids, and more. We will refer to this problem as the "Disjoint Set Problem". The API for this problem would be

```
void union(int p, int q); \\connects two ints p and q.
int find(int p); \\ the parent/value associated with a certain index(p).
boolean connected(int p, int q); \\are two int's, p and q, connected.
int count(); \\the amount of values.
```

The easiest way to approach this problem would to use some sort of array, we'll use *arr* as our array variable. Where the index of the array is the item we are looking at and the value is the "set" that it is in. In the naive implementation, when checking *connected(a,b)* we just check if arr[a] == arr[b]. To make sure that this is always the case, whenever we call *union(a,b)*, we will check if the two values are the same. If the values are the same, nothing would need to be done; however, if they are different, that means we will need to change all the items in *arr* that have the value of a to the value of b. Let's take the following example, the first image is of the initial array, the second is one where the call *union(2, 4)* has occurred



parent	1	2	2	4	4
index	1	2	3	4	5

Union(2,4)

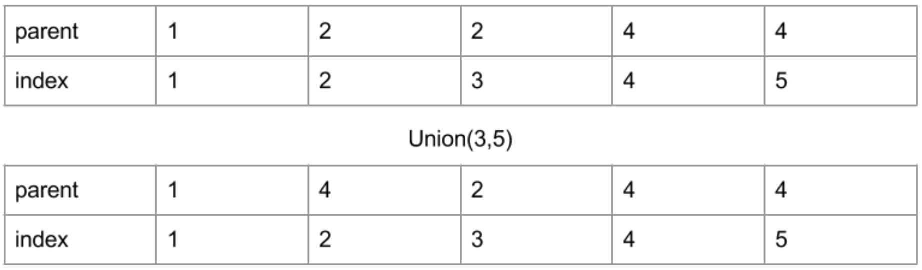
parent	1	4	4	4	4
index	1	2	3	4	5

Analyzing this code, we can recognize that *union* would take N time in worst case because you would have to

go through $O(N)$ elements in the array to connect the two. Find would take $O(1)$ because you would simply return the element associated with it. This data structure is essentially a linkedlist in terms of connectivity. This is decent, but we can do a lot better. Because *find()* takes such little time, we will call this data structure ***Quick Find***

So let's say we wanted to speed up the *union* method- we would use a similar data structure called a ***Quick***

Union. The underlying data structure would also be an array, and the value of each index would be its parent; however, our implementations of *find* and *union* would be quite different. Before, *find* would immediately return the value of the parent; however in the Quick Union Implementation, find would start at one item and go to its parent- this process would keep repeating up until the the parent of the node is itself, in other words, it is the root. To implement *union*, we would use *find* to find the root of the items, then all we would do is change the value of the root to be the value of the other root. Below is the basic java code for *find* and *union*.



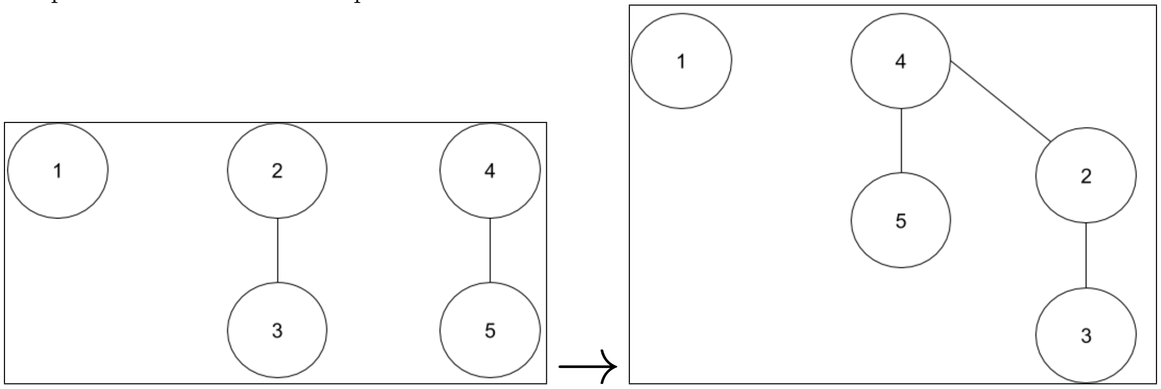
parent	1	2	2	4	4
index	1	2	3	4	5

Union(3,5)

parent	1	4	2	4	4
index	1	2	3	4	5

```
private int find(int p)
  {
        while (p != id[p]){
            p = id[p];
        }
    return p;
}
public void union(int p, int q){
     int pRoot = find(p);
     int qRoot = find(q);
     if (pRoot == qRoot){
          return;}
     id[pRoot] = qRoot;
     count--;
}
```

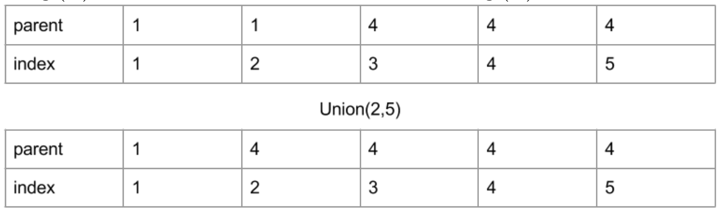
Note how only one of the values changed. Intuitively, this makes us feel that *Union* would be faster, but exactly how much faster? Well let's look at a diagram of how exactly QuickUnions are formed. Let's use the previous table as an example.



We can see that this forms a tree-like structure. This means that in worst case, the runtime for *union* and *find* is the $\Theta(N)$; however, the average runtime for both these functions is $\Theta(log(n)$, because that tends to be the height of a tree. Because the connections are arbitrary, the height can, at times, make the structure essentially a linked list. We'll tackle how to solve this problem in the next section.

2 Disjoint Set Improvement: Weighted Quick Union

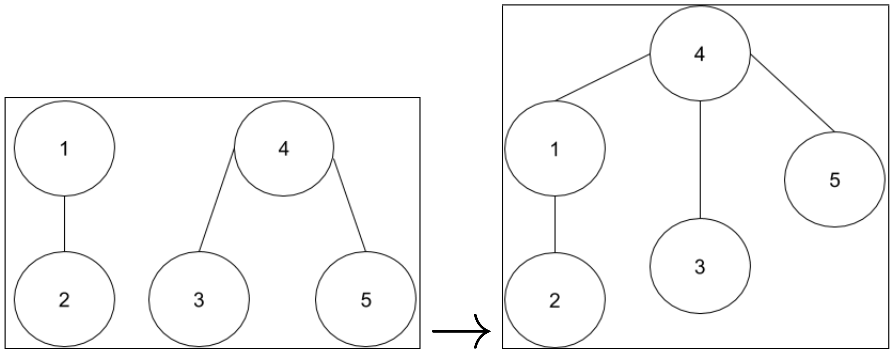
To solve the prior problem of having ridiculously large tree heights, we will implement a data structure called the ***Weighted Quick Union***. A Weighted Quick Union follows the same pattern as the Quick Union; however, we keep track of the size of the two trees being connected- we can do this with a separate "size" array or store it inside the node. The root of the smaller tree is then connected to the root of the larger tree- becoming its child- this ensures that the height of the tree will be no bigger than $log_2(n)$ or $lg(n)$. This makes it so *union* takes $lg(n)$ time in the worst case, a substantial improvement over N.



parent	1	1	4	4	4
index	1	2	3	4	5

Union(2,5)

parent	1	4	4	4	4
index	1	2	3	4	5

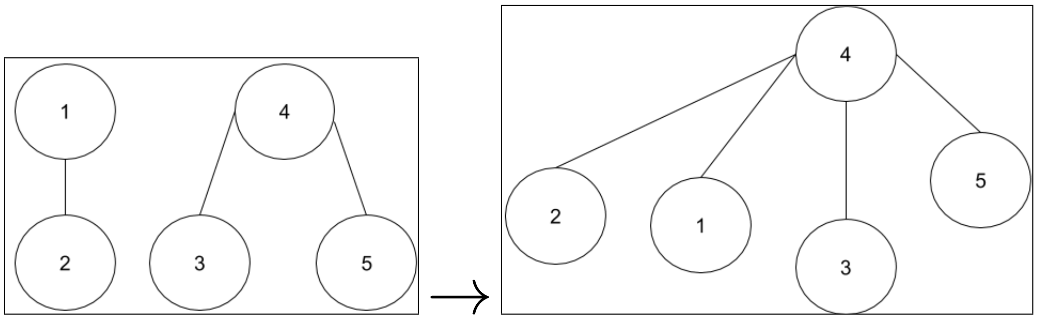


Though the Weighted Quick Union is good, we can make it even better by using a strategy called path compression. This strategy will provide us with a very fast *find* and *union* take nearly constant time. To implement this strategy, inside our *find* method, every node that you find on the way will connected directly to the root. Unlike Quick Find, where we connect everything in *union*, we connect everything inside the *find* method- this means that we will not go out of our way in order to connect things to the root as follows

parent	1	1	4	4	4
index	1	2	3	4	5

Union(2,5)

parent	4	4	4	4	4
index	1	2	3	4	5



In the above instance, right after 2 told us it's root is 1, we changed it's root to be 4. This makes it so that, in the long run, the tree is usually 1-2 levels, allowing for near constant time access. Let's compare the runtimes of the Union Data Structures:

Algorithm	Union Runtime	Find Runtime
Quick Find	N	N
Quick Union	Tree Height	Tree Height
Weighted Quick Union	lgN	lgN
Weighted Quick Union with Path Compression	Almost constant time Amortized	Almost constant time Amortized