Lab 06

Disjoint Sets

CS61B Fall 2023

Announcements

Homework 2 is released and will be due Wednesday, 10/4 at 11:59 pm.

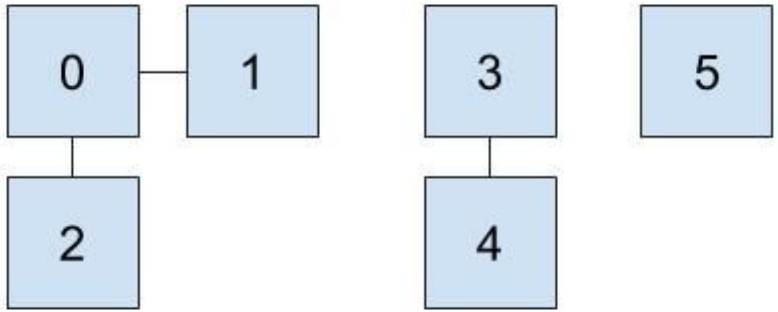
Disjoint Sets

Disjoint Sets

A disjoint set is a type of data structure that represents a *collection* of sets.

Some definitions:

- **Set**: a collection of items where there are no duplicates (each item is unique) and order is not maintained.
- **Disjoint:** Any item in the data structure **can not** be found in more than one set, i.e. an item can only ever exist in one of the sets at a time.



This is what a disjoint set might look like → it is a collection of sets. The sets in this representation consist of {0, 1, 2}, {3, 4}, and {5}. Notice that no specific item exists in more than one set.

Disjoint Sets

The main operations of disjoint sets consist of:

- **`find`:** determines which set an item belongs to
- **`union`:** merges two sets into one

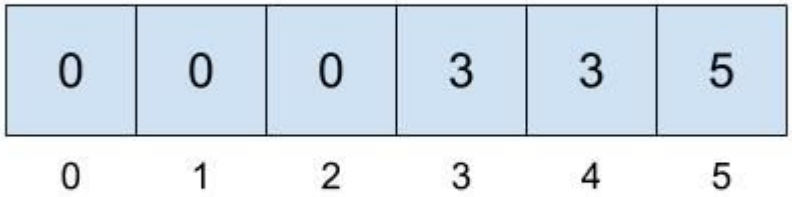
As a byproduct, the disjoint set data structure is also referred to as ***Union-Find.***

Quick Find

Quick Find

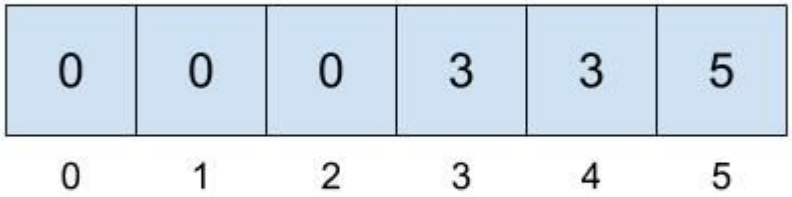
Given the two main operations, let's say we prioritize making the `find` operation faster. To do so, we can use an array to represent our disjoint sets.

Quick Find



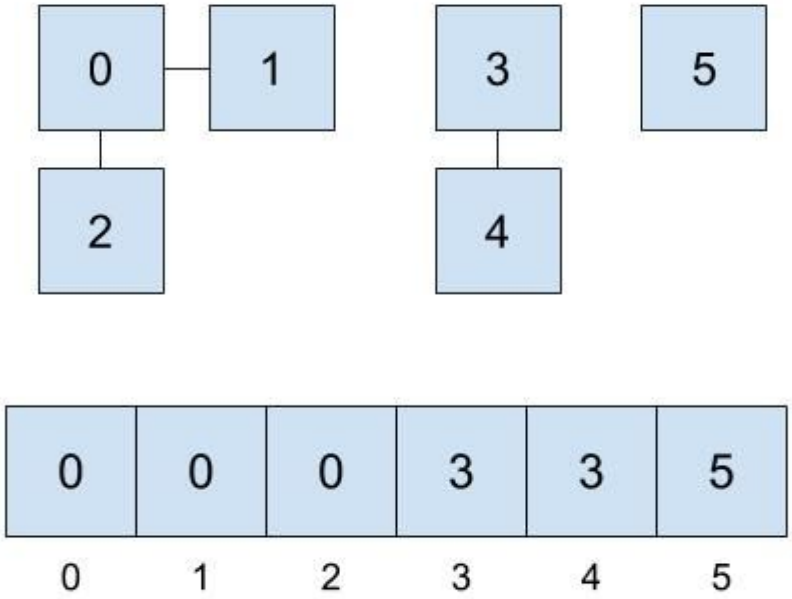
This is the array representation of the disjoint set that we saw earlier with the sets: {0, 1, 2}, {3, 4} and {5}. We treat the indices of the array as the items in the set and the element at each index corresponds to which set the item belongs to.

Quick Find



This is the array representation of the disjoint set that we saw earlier with the sets: {0, 1, 2}, {3, 4} and {5}. We treat the indices of the array as the items in the set and the element at each index corresponds to which set the item belongs to.

Notice that we take the smallest item to represent its entire set (i.e. 0 represents the set of {0, 1, 2} and 3 represents the set of {3, 4}, etc.). Which item we choose to represent the set is arbitrary and dependent on implementation.



Here is the visual representation seen earlier and its array representation. With this, we can achieve constant time for `find`, but end up with linear time for `union`!

Quick Union

Quick Union

With quick union, in our representation, we can think of our disjoint sets data structure as a collection of trees. Specifically, each set can be thought of as a tree and would have the following qualities:

Quick Union

With quick union, in our representation, we can think of our disjoint sets data structure as a collection of trees. Specifically, each set can be thought of as a tree and would have the following qualities:

- The nodes represent items in our set
- Each node only needs a reference to its parent, instead of a direct reference to the face of the set
- The root of each tree will be the face of the set it represents

Quick Union Modifications

So, to make our `union` operation faster, we'll need to make a couple of changes.

- Our array structure is still the same (each index corresponds to an item)

Quick Union Modifications

So, to make our `union` operation faster, we'll need to make a couple of changes.

- Our array structure is still the same (each index corresponds to an item)

However, instead of storing the item representing the face of a set, we store the parent references of a specific item.

- If an item has no parent, we will refer to this as the root of its set.

Unfortunately, depending on how we connect the disjoint sets, the **worst case runtime** for `union` will still be linear. How can we optimize this?

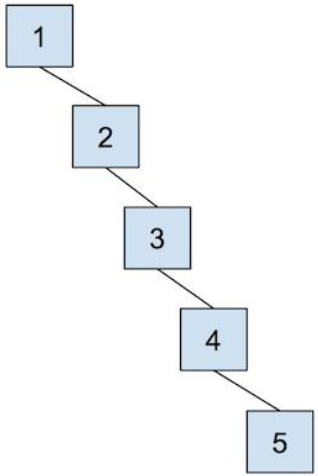
Weighted Quick Union

Union

While our union operation does become faster in some cases, it's worst case is still O(N). `find` is also still O(N). For example, if we end up with the following disjoint sets structure, we'll get the worst case runtime:

Union

While our union operation does become faster in some cases, it's worst case is still O(N). `find` is also still O(N). For example, if we end up with the following disjoint sets structure, we'll get the worst case runtime:



Weighted Quick Union

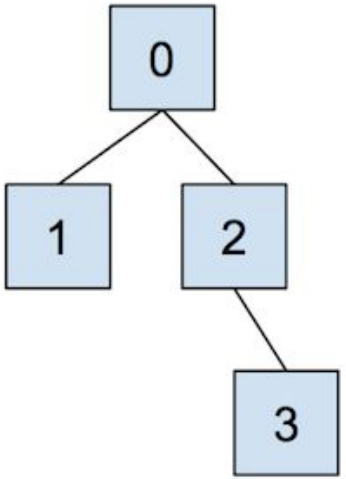
How can we remedy this problem?

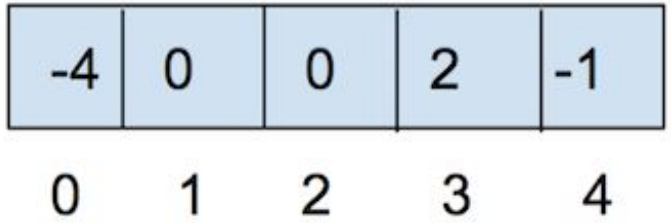
Weighted Quick Union

How can we remedy this problem?

We can solve this by unioning based on size/weight. This is done to keep trees as shallow as possible and to avoid the possibility of spindly trees, like before, from forming.

- By convention, when we `union`, we connect the **smaller tree (less nodes) as a subtree of the larger one.**
- In the case of ties, we can break it **arbitrarily** (some convention dictates to make the smaller-valued root the root of the combined sets, but ultimately implementation dependent)
- In our array, we also track the size of a set at the index corresponding to the root as **-size**.





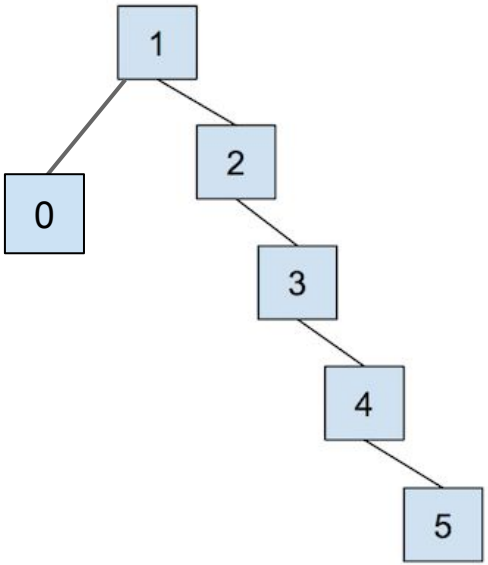
Refer to lab for an example for unioning by weight, but at this point, this is what a possible array representation would look like for our disjoint sets.

Path Compression

Path Compression

How can we optimize our Union-Find data structure even more?

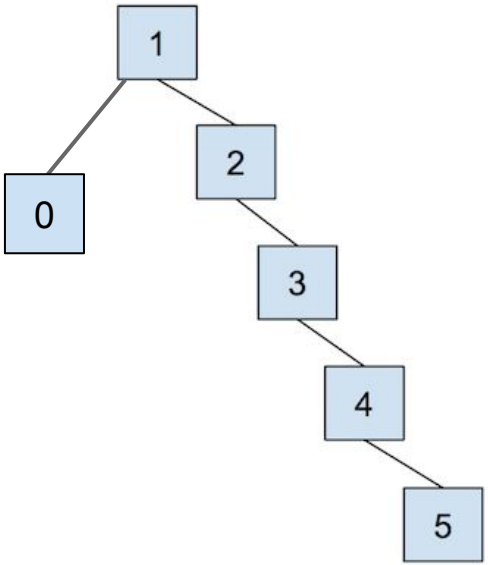
Consider the following Disjoint Set instance:



Path Compression

How can we optimize our Union-Find data structure even more?

Consider the following Disjoint Set instance:



If you repeatedly call `find` on the deepest leaf in this tree, i.e. `find(5)`, you would have to traverse through each parent from the leaf to the root every time with our current implementation.

Path Compression

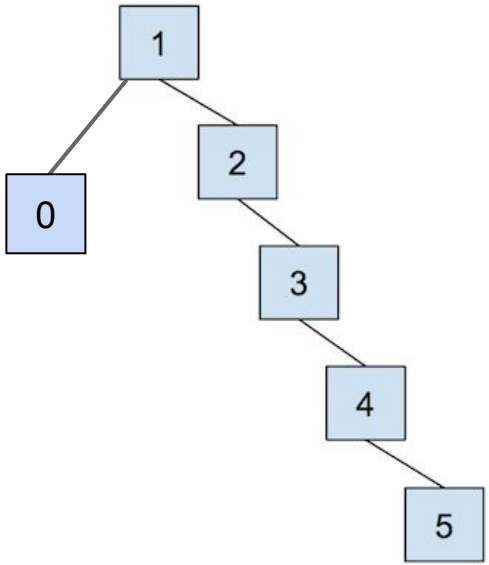
We can implement path compression to `find` an item and all the nodes on the path to the root in constant time, after the first call to `find`!

Path compression involves setting the parent of an item to be the root of its tree after we find the root. This also applies to every node on the path from the item to the root!

Path Compression

Consider the following Disjoint Set instance before Path Compression:

Tree Representation



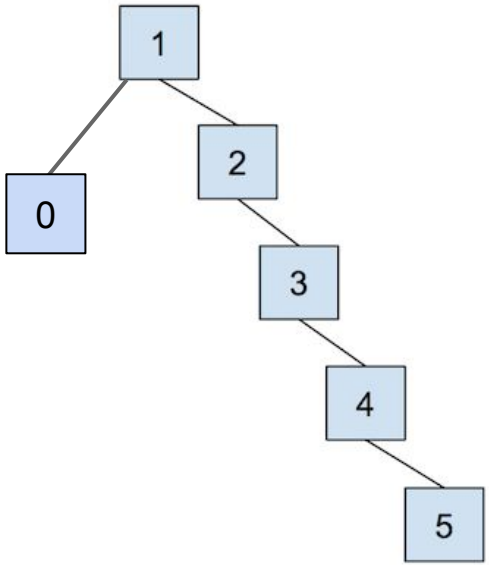
Array Representation

1	-6	1	2	3	4
0	1	2	3	4	5

Path Compression

Consider the following Disjoint Set instance before Path Compression:

Tree Representation



Array Representation

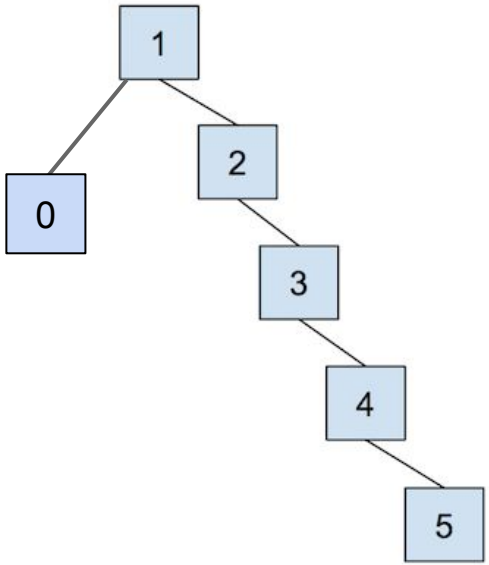
1	-6	1	2	3	4
0	1	2	3	4	5

Let's call `find(5)` -> `1`
Nodes in path to root: 5, 4, 3, 2, 1

Path Compression

Consider the following Disjoint Set instance before Path Compression:

Tree Representation



Array Representation

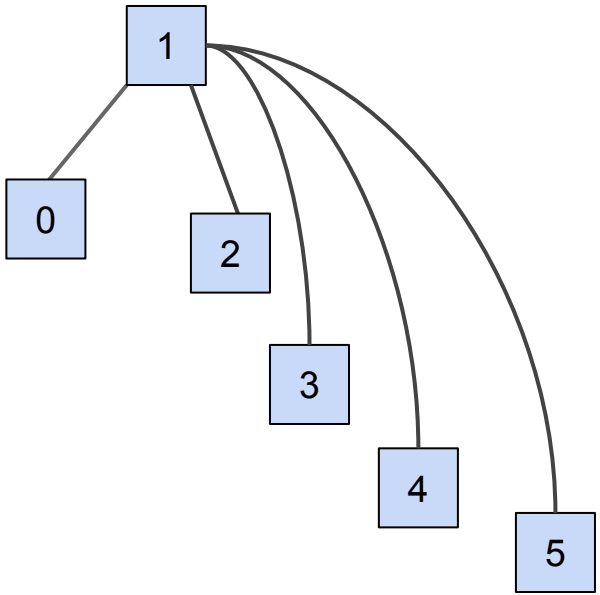
1	-6	1	2	3	4
0	1	2	3	4	5

Let's call `find(5) -> 1`
Nodes in path to root: 5, 4, 3, 2, 1
Path Compression: update parent of all nodes in path to become the root !

Path Compression

Consider the following Disjoint Set instance after Path Compression:

Tree Representation



Array Representation

1	-6	1	1	1	1
0	1	2	3	4	5

Let's call `find(5)` -> `1`
Nodes in path to root: 5, 4, 3, 2, 1
Path Compression: update parent of all nodes in path to become the root !

Lab Overview

An Overview

Lab 6 is due Friday, 9/29 at 11:59 pm.

- As a reminder, to get the lab assignment, run `git pull skeleton main` in your personal repository.

Deliverables:

- Complete `UnionFind.java` by implementing the Disjoint Sets data structure

For help, use the Lab queue: [INSERT]