An Overview of Disjoint Sets

By Raymond Guo
Spring 2018

This guide will be divided into 3 sections:

1. Motivation behind disjoint sets (the problem)
2. Different attempts at resolving the problem
3. Past midterm questions

Part 1: Motivation

Imagine that you're in a class of 10 people. The teacher assigns a group project, but she says that the groups can be of any size. Each group can be a different size as well. Once a group is formed, nobody is allowed to leave; the group is only allowed to expand as more and more members join it.

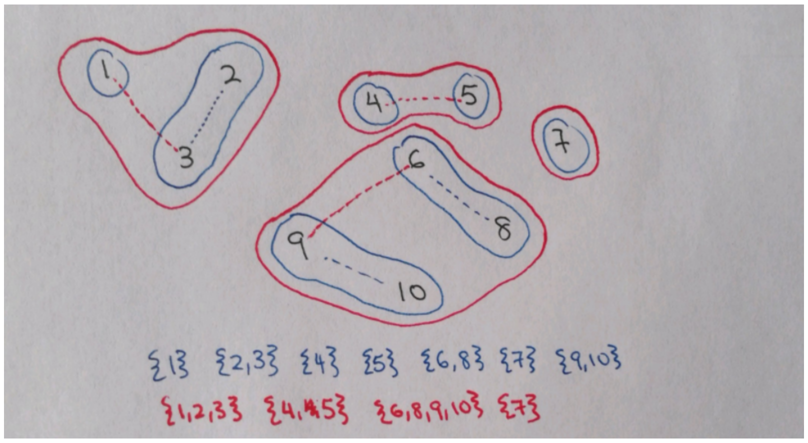
So the class starts splitting into groups.

In the first minute: (see Figure 1 below, in blue)

- Students 1, 4, 5, and 7 all choose to work alone.
- Students 2 and 3 choose to work together and form a group.
- Students 6 and 8 also form a group.
- Students 9 and 10 form another pair.

After a while, the groups start to shift again: (see Figure 1 below, in red)

- Student 1 decides that she wants to be in a group, so she reaches out to student 3 to join his group. Now, students 1, 2, and 3 are all in the same group.
- Students 4 and 5 form a group together. Students 6 and 9 decide to merge groups, so the entire group now consists of students 6, 8, 9, and 10.
- Student 7 decides to remain alone.



The two scenarios described in the previous two paragraphs are all representable with disjoint sets (think of groups like sets). The problem that disjoint sets attempts to solve is to answer the following problem: given any two students, can you tell me whether or not they are in the same group?

Specifically, disjoint sets is a type of data structure that represents the group-forming dynamic as explained previously. The data structure supports two action methods (other than the constructor):

- `void connect (int A, int B)` :: This method connects two students/groups together. In addition to students A and B being in the same group, all of the students originally in student A's group and all of the students originally in student B's group are all now in the same group too.
- `boolean isConnected (int A, int B)` :: This methods returns true if students A and B are in the same group, and false if they are not.

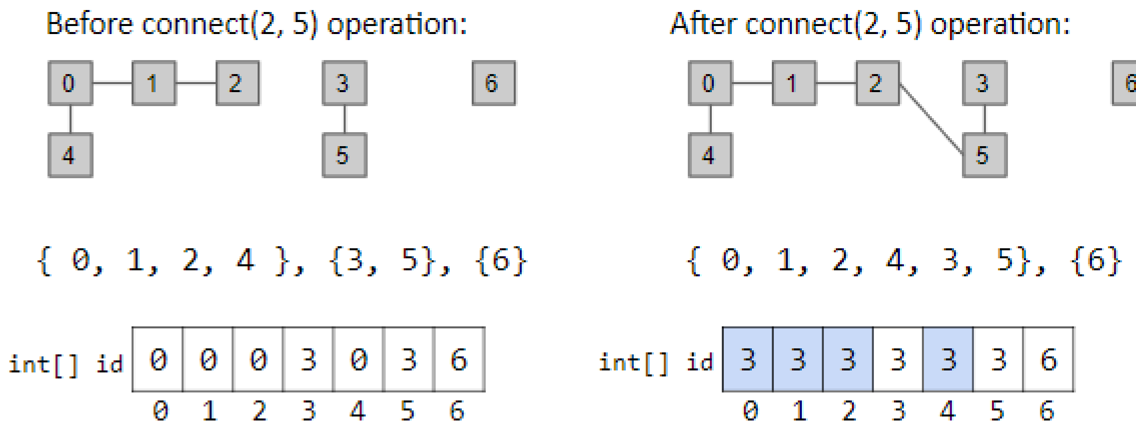
There are limitations to disjoint sets. The two main ones are:

1. The number of students in the class (items supported by the data structure) cannot change. This number is set when the data structure is instantiated.
2. Disconnections/dissociation of groups is not supported. Once connected, two students cannot split up.

Part 2: Approaches

Approach 1: Quick Find
Let's first start off with the simplest approach that comes to mind: using an array.

Overview:
Use an array of ints to represent which set (group) each item (student) is in. Specifically, `arr[i]` is equal to the set number that index `i` is in. See Figure 2 below.



Implementation:
`connect (int A, int B)` :: Save the value of `arr[A]`. Iterate through the array, and for every element that is equal to `arr[A]`, change the element value to `arr[B]`.

`isConnected (int A, int B)` :: Return true if `arr[A] == arr[B]`.

Runtime Analysis:

Implementation	connect	isConnected
QuickFindDS	Θ(N)	Θ(1)

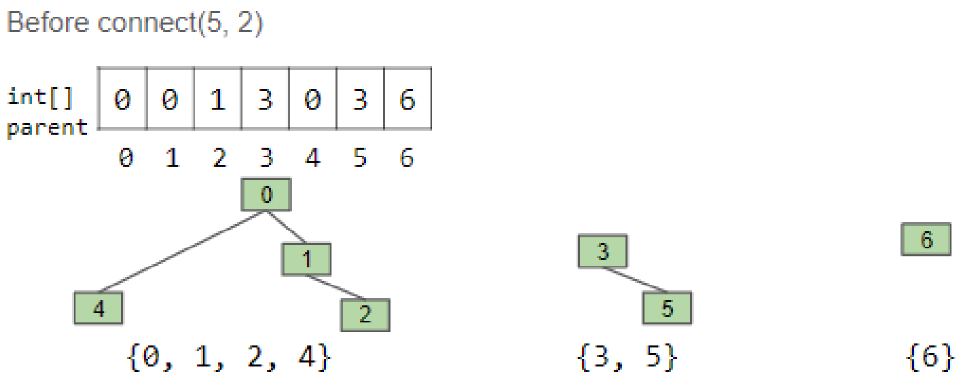
Biggest Problem: must iterate through entire array to connect, so poor runtime performance!

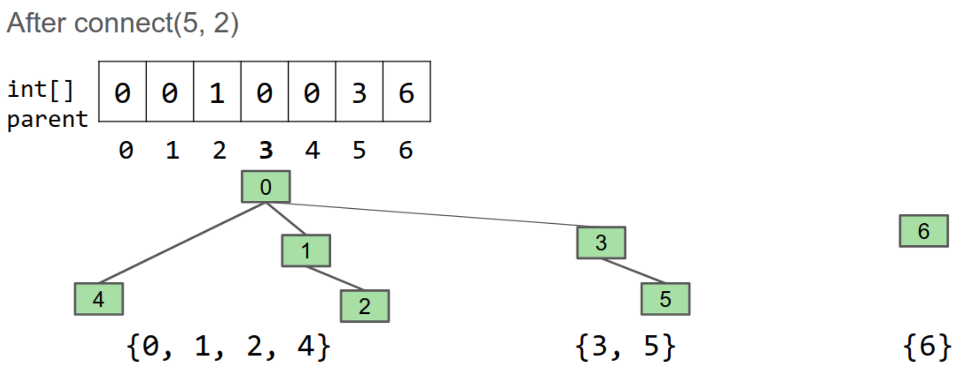
Approach 2: Quick Union

Using Quick Find was pretty good, but because it uses an array, the `connect` operation has a linear runtime. Is it possible to speed this up? Let's trying using a tree instead!

Overview:

Use an integer array, but this time to represent a tree structure. Each set (group) is represented with its own tree. Specifically, `arr` is an array of parents, in which `arr[i]` is equal to the value of the parent item of the index `i`. If `i` has no parent, `arr[i] = i`. See Figure 3 below.





Implementation:

`connect (int A, int B)` :: Find the root node of the tree containing A (start with setting `curr = arr[A]` and continuing [looping] to get the `curr`'s parent until `arr[curr] = curr`). Find the root node of the tree containing B. Set the parent of A's root node to B's root node (by setting `arr[rootA] = rootB`).

`isConnected (int A, int B)` :: Find the root node of the tree containing A. Find the root node of tree containing B. Return true if `rootA == rootB`.

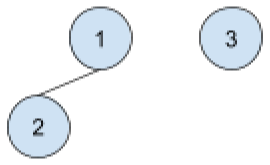
Runtime Analysis:

Implementation	connect	isConnected
QuickUnionDS	O(N)	O(N)

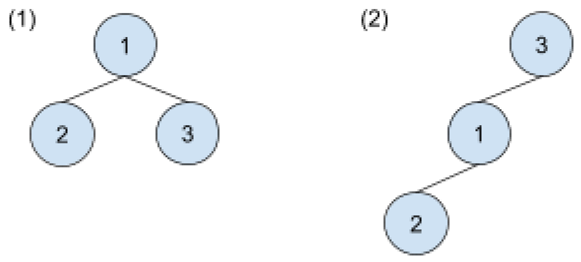
Biggest Problem: Trees can get to splindly, actually leading to worst runtime performance!

Approach 3: Weighted Quick Union (KNOW THIS)

Let's first quickly remember the main issue that we ran into using Quick Union: sometimes, the trees get very spindly, thus leading to poor runtimes. But when exactly do trees get spindly? Consider the following example:



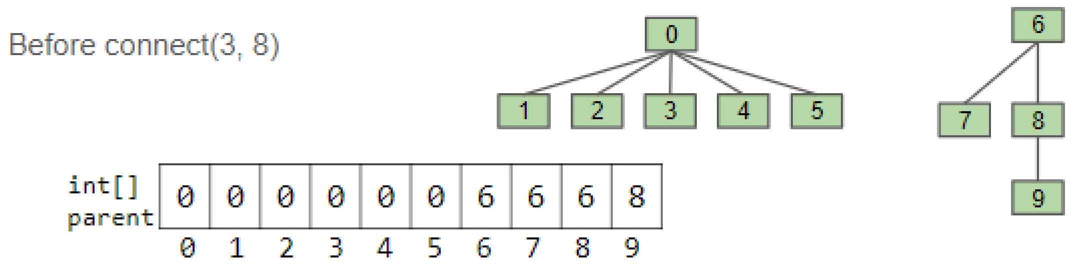
We want to connect these two trees. There are two ways: (1) connect the smaller tree to the larger tree, or (2) connect the larger tree to the smaller tree. Let's see what happens in both scenarios:

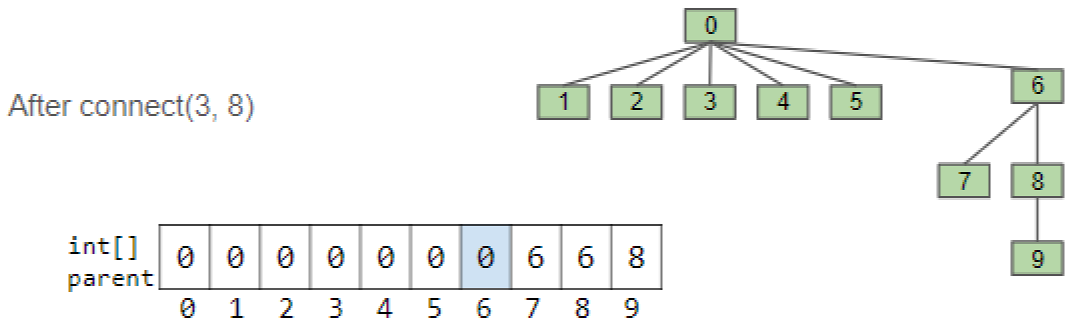


Strategy (1) led to a more bushier tree! Let's incorporate this new strategy into Quick Union.

Overview:

Implement Quick Union, but also keep track of the size of each tree (number of elements). When you connect, connect smaller tree to larger tree. See Figure 4 below.





Implementation:

`connect (int A, int B)` :: Find the root node of the tree containing A. Find the root node of the tree containing B. If size of A's tree is smaller, set the parent of A's root node to B's root node (by setting `arr[rootA] = rootB`). If size of A's tree is greater, set the parent of B's root node to A's root node (by setting `arr[rootB] = rootA`). If size of both trees equal, tree with smaller parent node becomes the parent.

`isConnected (int A, int B)` :: (same as QuickUnion) Find the root node of the tree containing A. Find the root node of tree containing B. Return true if `rootA == rootB`.

Runtime Analysis:

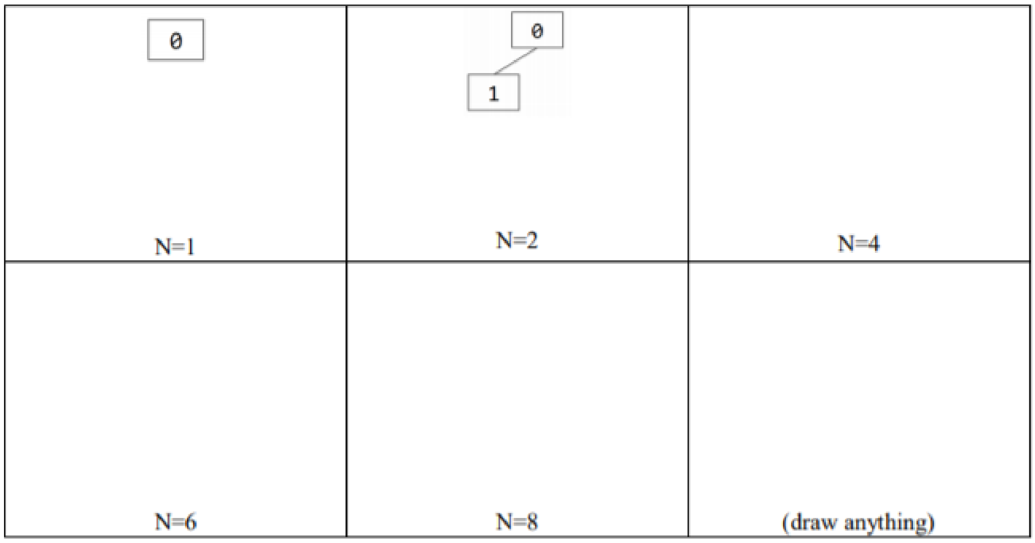
Implementation	connect	isConnected
WeightedQuickUnionDS	O(log N)	O(log N)

Part 3: Practice

CS61B Midterm 2 Spring 2016

2. WeightedQuickUnionUF (4 Points).

a) Draw a valid `WeightedQuickUnionUF` tree with worst case height, given sizes of N=1, N=2, N=4, N= 6, and N=8 in the boxes below, where N is the number of items in the `WeightedQuickUnionUF`. The first two are done for you. Recall that the height of a tree is the length of the longest path from the root to any leaf, so the height of the tree for N=2 is 1.



b) Give the best case height and worst case height of a `WeightedQuickUnionUF` tree in Θ notation in terms of N, the number of items in the `WeightedQuickUnionUF`.

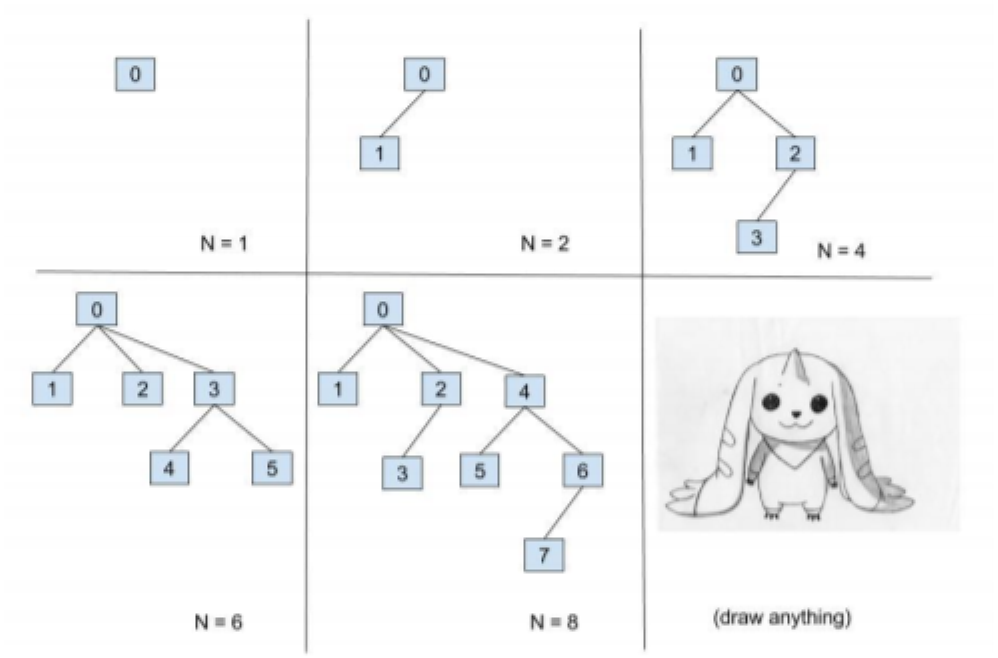
Best: _____________

Worst: _____________

Solution:

2. WeightedQuickUnionUF (4 Points).

a) Draw a valid `WeightedQuickUnionUF` tree with worst case height, given sizes of N=1, N=2, N=4, N= 6, and N=8 in the boxes below, where N is the number of items in the `WeightedQuickUnionUF`. The first two are done for you. Recall that the height of a tree is the length of the longest path from the root to any leaf, so the height of the tree for N=2 is 1.



There are many valid examples for worst case heights. One approach is to do a union of two sets of size $\frac{N}{2}$ that have worst case height.

b) Give the best case height and worst case height of a `WeightedQuickUnionUF` tree in Θ notation in terms of N, the number of items in the `WeightedQuickUnionUF`.

Best: $\Theta(1)$
If all unions are made such that the second set is size 1, the height will be 1.

Worst: $\Theta(\log N)$
In the worst case, we can combine 2 trees of size $\frac{N}{2}$ that already have worst case height. Doing so will increase the height of the tree by 1 every time you double N, which translates to $\log_2(N)$.

3. Weighted Quick Union.

a) **(10 points).** Define a "fully connected" `DisjointSets` object as one in which `connected` returns true for any arguments, due to prior calls to `union`. Suppose we have a fully connected `DisjointSets` object with **6 items**. Give the best and worst case height for the two implementations below. The height is the number of links from the root to the deepest leaf, i.e. a tree with 1 element has a height of 0. **Give your answer as an exact value**. Assume Heighted Quick Union is like Weighted Quick Union, except uses height instead of weight to determine which subtree is the new root.

	Best Case Height	Worst Case Height
Weighted Quick Union		
Heighted Quick Union		

b) **(8 points).** Suppose we have a Weighted Quick Union object of height H. Give a general formula for the minimum number of objects in a tree of height H as a function of H. Your answer must be exact (e.g. not big theta).

c) **(6 points).** Draw a Quick Union tree of 6 objects or fewer that would be **possible** for Heighted Quick Union, but **impossible** for Weighted Quick Union. If no such tree exists, simply write "none exists."

Solution:

3. Weighted Quick Union.

a) **(10 points).** Define a "fully connected" `DisjointSets` object as one in which `connected` returns true for any arguments, due to prior calls to `union`. Suppose we have a fully connected `DisjointSets` object with **6 items**. Give the best and worst case height for the two implementations below. The height is the number of links from the root to the deepest leaf, i.e. a tree with 1 element has a height of 0. **Give your answer as an exact value**. Assume Heighted Quick Union is like Weighted Quick Union, except uses height instead of weight to determine which subtree is the new root.

	Best Case Height	Worst Case Height
Weighted Quick Union	1	2
Heighted Quick Union	1	2

b) **(8 points).** Suppose we have a Weighted Quick Union object of height H. Give a general formula for the minimum number of objects in a tree of height H as a function of H. Your answer must be exact (e.g. not big theta).
2^H

c) **(6 points).** Draw a Quick Union tree that would be **possible for Heighted Quick Union**, but **impossible** for Weighted Quick Union. If no such tree exists, simply write "none exists."
Example tree shown below. In general, the heights of the two input trees should be the same, but the heavier tree should be attached to the lighter one.