csce350 — Data Structures and Algorithms
Fall 2019 — Extra Notes on Disjoint Set Forests

This document is a brief review of the disjoint set forest data structure discussed in class on December 3 and December 5, used as part of Kruskal's algorithm.

The **disjoint set forest** data structure is useful for grouping a collection of elements into sets, so that it is easy to tell whether any two elements are in the same set or not. If the number of elements is known ahead of time, the abstract data type looks like this:[1]

> ADT: UNIONFIND
> find(x): return a representative of the set containing the given element.
> union(x, y): union the set containing x with the set containing y.

There are two operations:

- **Find** returns a *representative* of the set containing the given element. The representative is another element from the same set. Think of the representative as the "team leader" for the set. The important thing is that *every element in the set should have the same representative*. This allows us to determine whether two elements are in the same set by comparing their representatives.
- **Union** joins two sets together. Informally, the "team leader" of one set takes over as the team leader of the other.

To implement these operations, we use a **parent array**. This is an array of integers, with one cell for each element. Each cell stores the index of the *parent* of its element. In this way, parent array describes a collection of rooted trees, each representing a single set. The root of such a tree (which is its own parent) is the representative of the set.

- To implement **find**(x), we start with cell x, and move up the tree until reaching the root, that is, until reaching an element that is its own parent. This root element is the representative of the set containing x.
- To implement **union**(x, y), we perform a **find** on both x and y, then make one of the tree roots we find a child of the other.

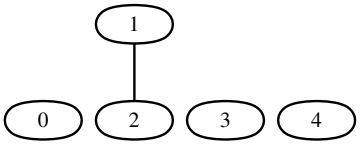
Here's an example, using 5 elements numbered 0 through 4. To start, all of the elements are in separate sets, so each is its own parent.

$P[0]$	$P[1]$	$P[2]$	$P[3]$	$P[4]$
0	1	2	3	4

[1] The textbook shows a slightly different data structure that has a separate **MakeSet** operation, for the case in which you don't know ahead of time how many elements there will be. That case works out the same as what's described here, except that you must be prepared to allocate memory dynamically. For Kruskal's algorithm, the elements are nodes in a known graph, so that extra complexity isn't needed.

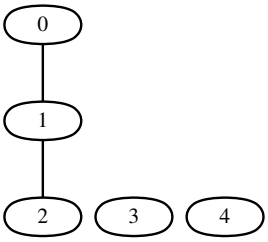
Now suppose were execute **union**$(1, 2)$. We end up with:

$P[0]$	$P[1]$	$P[2]$	$P[3]$	$P[4]$
0	1	1	3	4



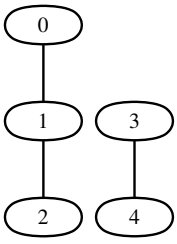
Next, let's try a **union**$(0, 2)$. Remember, for the union operation, we need to find the root of the two trees first, then make one a child of the other. In this case, the roots are 0 and 1:

$P[0]$	$P[1]$	$P[2]$	$P[3]$	$P[4]$
0	0	1	3	4



Notice that each union only changes one cell in the parent array, even if the tree structures being merged are complicated. Now **union**$(3, 4)$:

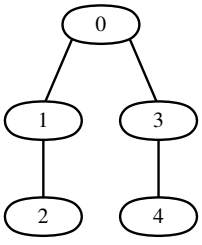
$P[0]$	$P[1]$	$P[2]$	$P[3]$	$P[4]$
0	0	1	3	3



This is a good time to emphasize the way this data structure lets us determine that 2 and 4 are in different sets: **find**(2) returns 0, and **find**(4) returns 3. Since the roots are different, we know that 2 and 4 are not in the same set.

Let's finish things up by doing **union**$(1, 4)$:

$P[0]$	$P[1]$	$P[2]$	$P[3]$	$P[4]$
0	0	1	0	3



Now any **find** operation will return 0, which indicates that all of the elements are in the same set.