

Implementing Quick-Union with an Array

Suppose the items are non-negative integers, numbered from zero. We'll use an array to record the parent of each item. If an item has no parent, we'll record the size of its tree. To distinguish it from a parent reference, we'll record the size s as the negative number $-s$. Initially, every item is the root of its own tree, so we set every array element to -1 .

```
-----
|-1|-1|-1|-1|-1|-1|-1|-1|-1|-1|
-----
 0  1  2  3  4  5  6  7  8  9
```

The forest illustrated at left below is represented by the array at right.

```

      8      1      2
    / \    / \
   5   3  9 0 6
   |   |
   4   7

-----
| 1|-4|-1| 8| 5| 8| 1| 3|-5| 1|
-----
 0  1  2  3  4  5  6  7  8  9
```

This is a slightly kludgy way to implement tree-based disjoint sets, but it's fast (in terms of the constant hidden in the asymptotic notation).

Let `root1` and `root2` be two items that are roots of their respective trees. Here is code for the union operation with the union-by-size strategy.

```
public void union(int root1, int root2) {
    if (array[root2] < array[root1]) {
        // root2 has larger tree
        array[root2] += array[root1];
        array[root1] = root2;
    } else {
        // root1 has equal or larger tree
        array[root1] += array[root2];
        array[root2] = root1;
    }
}
```

Path Compression

The `find()` method is equally simple, but we need one more trick to obtain the best possible speed. Suppose a sequence of union operations creates a tall tree, and we perform `find()` repeatedly on its deepest leaf. Each time we perform `find()`, we walk up the tree from leaf to root, perhaps at considerable expense. When we perform `find()` the first time, why not move the leaf up the tree so that it becomes a child of the root? That way, next time we perform `find()` on the same leaf, it will run much more quickly. Furthermore, why not do the same for every node we encounter as we walk up to the root?

```

      0
    / \
   1 2 3
  / \
 4 5 6
 / \
7 8 9

==find(7)==>

      0
    / \
   7 4 1 2 3
  / \
 8 9 5 6
```

In the example above, `find(7)` walks up the tree from 7, discovers that 0 is the root, and then makes 0 be the parent of 4 and 7, so that future find operations on 4, 7, or their descendants will be faster. This technique is called `_path_compression_`.

Let x be an item whose set we wish to identify. Here is code for `find`, which returns the identity of the item at the root of the tree.

```
public int find(int x) {
    if (array[x] < 0) {
        return x;
    } else {
        // Find out who the root is; compress path by making the root x's parent
        array[x] = find(array[x]);
        return array[x];
    }
}
```

Naming Sets

Union-by-size means that if Microsoft acquires US Air, US Air will be the root of the tree, even though the new conglomerate might still be called Microsoft. What if we want some control over the names of the sets when we perform `union()` operations?

The solution is to maintain an additional array that maps root items to set names (and perhaps vice versa, depending on the application's needs). For instance, the array "name" might map 0 to Microsoft. We must modify the `union()` method so that when it unites two sets, it assigns the union an appropriate name.

For many applications, however, we don't care about the name of a set at all; we only want to know if two items x and y are in the same set. This is true in both Homework 9 and Project 3. You only need to run `find(x)`, run `find(y)`, and check if the two roots are the same.

Running Time of Quick-Union

Union operations obviously take $\Theta(1)$ time. (Look at the code--no loops or recursion.)

If we use union-by-size, a single `find` operation can take $\Theta(\log u)$ worst-case time, where u is the number of union operations that took place prior to the find. Path compression does not improve this worst-case time, but it improves the average running time substantially--although a find operation can take $\Theta(\log u)$ time, path compression will make that operation fast if you do it again. The average running time of find and union operations in the quick-union data structure is so close to a constant that it's hardly worth mentioning that, in a rigorous asymptotic sense, it's slightly slower.

The bottom line: a sequence of f find and u union operations (in any order and possibly interleaved) takes $\Theta(u + f \alpha(f + u, u))$ time in the worst case. α is an extremely slowly-growing function known as the `_inverse_Ackermann_function_`.

When I say "extremely slowly-growing function", I mean "comically slowly-growing function." The inverse Ackermann function is never larger than 4 for any values of f and u you could ever use (though it does grow arbitrarily large--for unimaginably gigantic values of f and u). Hence, for all practical purposes (but not on the Final Exam), you should think of quick-union as having find operations that run, on average, in constant time.