CS61B Summer 2006
Instructor: Erin Korber
Lecture 30: 15 Aug.

Disjoint Sets

Given a set of elements, it is often useful to break them up or *partition* them into a number of separate, nonoverlapping groups.

A *partition* of S is a collection of subsets S_1,S_2... of S such that every element of S belongs to exactly one of the subsets.

A disjoint-set data structure is a data structure that keeps track of such a partitioning. These data structures will have two operations, `union` and `find`. In the `union` operation, we merge two sets into one. In the `find` operation, we ask to which set an item belongs. Data structures supporting these operations are also called (unsuprisingly) "union/find" data structures.

For example, suppose the items in our set S are corporations that still exist today or were acquired by other corporations, and we will form a partition for each corporation that still exists under its own name. For instance, "Microsoft," "Connectix," and "Web TV" are all members of the "Microsoft" subset.

Applications of union/find data structures include maze generation and building minimum spanning trees using Kruskal's algorithm (as we saw yesterday).

1 List-based disjoint sets

The obvious data structure for disjoint sets looks like this.

- Each set references a list of the items in that set.
- Each item references the set that contains it.

With this data structure, find operations take O(1) time (given an item, just look at its reference to what set its in). So we say that list-based disjoint sets use the "quick-find" algorithm. However, union operations are slow, because when two sets are united, we must walk through one set and relabel all the items so that they reference the other set. So let's look at a more efficient structure that we will say uses "quick-union".

2 Tree-based disjoint sets

In tree-based disjoint sets, union operations take O(1) time, but find operations are slower. However, for any sequence of union and find operations, the quick-union algorithm is faster overall than the quick-find algorithm.

To support fast unions, each set is maintained as a general tree. The quick-union data structure comprises a forest (a collection of trees), in which each item is initially the root of its own tree; then trees are merged by union operations.

The quick-union data structure is simpler than the general tree structures you have studied so far, because there are no child or sibling references. Every node knows only its parent, and you can only walk up the tree. We identify a set by its root.

Union is a simple O(1) time operation: we simply make the root of one set become a child of the root of the other set.

However, finding the set to which a given item belongs is not a constant-time operation. The find operation is performed by following the chain of parent references from an item to the root. The cost of this operation is proportional to the item's depth in the tree.

To keep items from getting too deep, we could unite sets intelligently. At each root, we record the size of its tree. When we unite two trees, we make the smaller one a subtree of the larger one (breaking ties arbitrarily). This strategy is called "union-by-size."

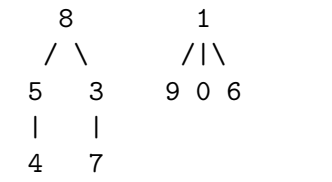
2.1 Implementing Quick-Union with an Array

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 the set it represents. 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. This is somewhat ugly and inelegant, but it's fast (in terms of the constant hidden in the asymptotic notation).

To start:

index	0	1	2	3	4	5	6	7	8	9
value	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

The forest illustrated at below is represented by the array below it:



index	0	1	2	3	4	5	6	7	8	9
value	1	-4	-1	8	5	8	1	3	-5	1

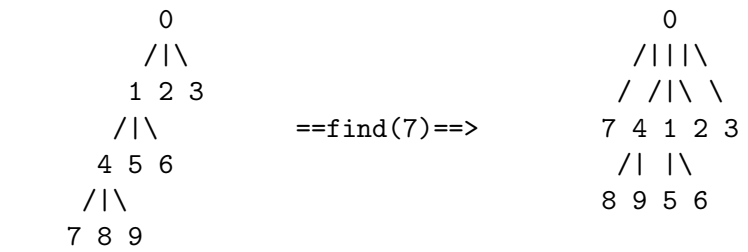
Let root1 and root2 be two items that are roots of their respective trees. Here is code for union.

```
public void union(int root1, int root2) {
  if (array[root2] < array[root1]) {     // root2 has larger tree
    array[root2] += array[root1];        // update # of items in root2's tree
    array[root1] = root2;                // make root2 new root
  } else {                               // root1 has equal or larger tree
    array[root1] += array[root2];        // update # of items in root1's tree
    array[root2] = root1;                // make root1 new root
  }
}
```

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?

For example:



In the example above, find(7) walks up the tree from 7, discovers that 0 is the root, and then makes 0 the parent of 4 and 7, so that future find operations on 4, 7, or their children 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. Recall that items are numbered starting from zero.

```
public int find(int x) {
  if (array[x] < 0) {
    return x;               // x is the root of the tree; return it
  } else {
    // Find out who the root is; compress path
    array[x] = find(array[x]);
    return array[x];                  // Return the root
  }
}
```

What if we want some control over the names of the sets when we perform `union` operations? The solution is to maintain a separate array that maps root items to set names, and perhaps vice versa (depending on the application's needs). For instance, in our corporation example, the name array might map 0 to Microsoft. The `union` method must be modified 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, for example, for Kruskal's algorithm: to decide if there is a path

between `u` and `v`, you only need to run `find(u)` and `find(v)`; if they are equal, there is a path between `u` and `v`.

2.2 Running Time of Quick-Union

Union operations obviously take $\Theta(1)$ time.

A single find operation can take time as great as $\Theta(\log n)$, where n is the number of union operations that took place prior to the find. However, the worst-case *average* running time of union/find operations is better because of path compression.

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 mathematical sense, it's slightly slower.

To be precise: 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." This function is never larger than 4 for any values of f and u you could ever use. (For instance, the first value of n such that $\alpha(n) = 5$ is so large that $\log_2(\log_2(n)) > p^{200}$, where p is the estimated number of particles in the universe.) Hence, for all practical purposes, quick-union has `find` and `union` operations that run, on average, in effectively constant time.

Note that the $|E| \log |V|$ bound for the running time of Kruskal's algorithm is obtainable even without path compression, where a find takes $\log(n)$ time.