Announcements:

- PS4 due Thursday Oct 20, 11:59PM
- Partner up for PS5!

CS 3110 Lecture 12
Imperative Data Structures: Disjoint Sets

Given a set of elements *S*, a partition of *S* is a set of nonempty subsets of *S* such that every element of *S* is in exactly one of the subsets.

In other words the subsets making up the partition are pairwise disjoint, and together contain all the elements of *S* (cover the set).

A *disjoint set* data structure is an efficient way of keeping track of such a partition. There are two operations on a disjoint set:

1. union - merge two sets of the partitition into one, changing the partition structure

2. find - determine which set of the partition contains a given element *e*, returning a canonical element of that set

Sometimes a disjoint set is also referred to as a union-find data structure because it supports these two operations.

In addition, the create operation makes a partition where each element *e* is in its own set (all the subsets in the partition are singletons).

Efficient implementations of the union and find operations make use of the ability to change the values of variables, thus we make use of refs and arrays introduced in recitation.

Disjoint sets are commonly used in graph algorithms.

For instance, consider the problem of finding the connected components in an undirected graph (sets of nodes that are reachable from one another by some path).

If each edge in the graph *E(i,j)* connects two nodes *v(i)* and *v(j)*, then the following simple algorithm will label all the nodes in each component with the same identifier (and nodes in different components with different identifiers):

1. Create a new partition with one element corresponding to each node *v(i)* in the graph.

2. For each edge *E(i,j)* in the graph call the union operation with *v(i)* and *v(j)*

3. For each vertex *v(i)* in the graph call the find operation, which returns the component label for that vertex

This is actually an extremely important algorithm, used in e.g. OCR. You can speed it up by doing some preprocessing, for example a horizontal merge (union).

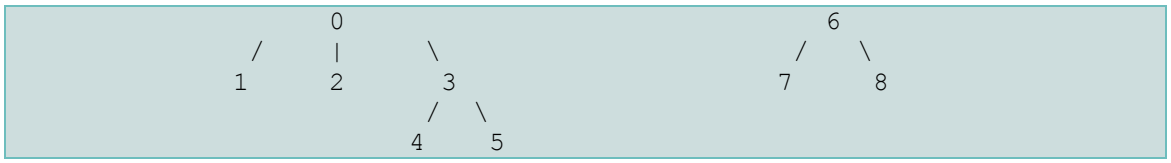
Representing Forests as Arrays

A common way of representing a disjoint set is as a *forest*, or collection of trees, with one tree corresponding to each set of the partition.

When the nodes of the trees are labeled by consecutive natural numbers, it is straightforward to implement a forest using an array.

The array index corresponds to the label of the node, and the corresponding array entry at that index specifies the label of the node's parent (and the root of a tree specifies itself as parent).

For instance the forest:



would be represented by the array

```
+---+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 0 | 3 | 3 | 6 | 6 | 6 |
+---+---+---+---+---+---+---+---+---+
```

With this representation of a disjoint set, a new partition is simply:

```
let createUniverse size =
  Array.init size (fun i -> i)
```

Using this representation, the find operation simply checks the specified index of the array, and if the value is equal to the index returns the index, otherwise recursively does a find with the value in the array.

This searches from a node to the root of its tree in the forest:

```
let rec find s e =
  let p = s.(e) in
    if p = e then e
    else find s p
```

The union operation finds the root of the tree for each of the two elements, and then assigns one of the two roots to have the other as parent:

```
let union s e1 e2 =
  let r1 = find s e1
  and r2 = find s e2 in
    s.(r1) <- r2
```

Union simply does two finds and a pointer update, thus the asymptotic running time of union is clearly the same as that of find.

In the worst case the find operation can take O(n) time for an array of n elements, because the array can represent just one tree which a single path of depth n (in that case it will be necessary to consider every entry of the array before reaching the one that is the root of the tree).

Thus as with the balanced binary tree schemes such as red-black trees that we considered earlier, we need a balancing mechanism to ensure that the trees have depth that is only logarithmic in the number of nodes.

The trees in a forest represented as an array are relatively simple to balance, because they are n-ary trees and thus the branching factor can be adjusted to lower the depth.

The key observation is that when merging two trees in the union operation, if the shallower tree is made a child of the root of the deeper tree then the overall depth of the resulting tree does not change. If the two trees are of the same depth, then the resulting tree is one deeper (and this is the only case that it gets deeper).

Rather than using the actual depth of the trees, we use a quantity referred to as the *rank*, which is an upper bound on the depth of the tree.

One-element trees are defined to have a rank of zero, and whenever two trees of the same rank *r* are united, the rank of the result is *r*+1. This balancing scheme is referred to as *union by rank*.

Our data structure now needs to store a rank for each node, in addition to a parent. For instance:

```
type node = {mutable parent : int; mutable rank : int}
type universe = node array

let createUniverse size =
  Array.init size (fun i -> {parent = i; rank = 0})
```

The union by rank function simply finds the roots of the trees for both elements as before (although now we need to process the node and not only the index).

If the two roots are the same there is nothing to do, if they are different, then the one that is larger rank is made the parent of the one that is smaller rank.

If the ranks are equal one is arbitrarily made the parent and the rank of the resulting tree is increased by one:

```
let union s e1 e2 =
  let r1 = find s e1
  and r2 = find s e2 in
  let n1 = s.(r1)
  and n2 = s.(r2) in
    if r1 != r2 then
      if n1.rank < n2.rank then
      n1.parent <- e2
      else
       (n2.parent <- e1;
       if n1.rank = n2.rank then
         n1.rank <- n1.rank + 1)
```

This process for constructing trees results in ranks that are logarithmic in the number of nodes, and thus the running time of union and find operations is O(log n) for n nodes.

If a node has rank *n* then the subtree rooted at that node has size at least 2^n nodes. Proof by induction. Base case: a node of rank 0 is the root of a subtree that contains at least itself and thus is size at least 2^0=1. Inductive step, show that a rank k+1 tree has at least 2^(k+1) nodes: A node u can have rank k+1 only if, at some point, it had rank k and it was the root of a tree that was joined with another tree whose root also had rank k. Then u became the root of the union of the two trees. Each tree, by the inductive hypothesis, was of size at least 2^k, so u is the root of a tree of size at least 2^(k+1)=2^k+2^k.

The find operation can also be improved, using a technique known as *path compression*. When tracing from a node back to its root, also update the parent pointer to point directly to the root. This in effect makes the tree flatter and bushier (making the tree broad rather than deep).

```
let rec find (s : universe) (e : int) : int =
  let n = s.(e) in
    if n.parent = e then e
    else
      (n.parent <- (find s n.parent);
       n.parent)
```

An analysis more involved than those we consider in CS3110 can establish that with union by rank and path compression, any sequence of *m* union and find operations on a disjoint set with *n* elements take at most O((m+n) log* n) steps, where log* n is the number of times you need to apply the log function to n before you get a number less than or equal to 1 (for all practical purposes log* n is no more than 5 for any number that could be represented in a digital computer). Such more detailed analyses are covered in CS 4820.