

CSC 378 Lecture 10

July 19, 2001

For another reference, see chapter 22 of the textbook (CLR).

1 The Disjoint Set ADT (also called the Union-Find ADT)

Two sets A and B are *disjoint* if their intersection is empty: $A \cap B = \emptyset$. In other words, if there is no element in both sets, then the sets are disjoint. The following abstract data type, called “Disjoint Set” or “Union-Find,” deals with a group of sets where each set is disjoint from every other set (i.e. they are pairwise disjoint).

Object: A collection of nonempty, pairwise disjoint sets: S_1, \dots, S_k . Each set contains a special element called its *representative*.

Operations:

- **MAKE-SET(x):** Takes an element x that is not in any of the current sets, and adds the set $\{x\}$ to the collection. The representative of this new set is x .
- **FIND-SET(x):** Given an element x , return the representative of the set that contains x (or some NIL if x does not belong to any set).
- **UNION(x,y):** Given two distinct elements x and y , let S_i be the set that contains x and S_j be the set that contains y . This operation adds the set $S_i \cup S_j$ to the collection and it removes S_i and S_j (since all the sets must be disjoint). It also picks a representative for the new set (how it chooses the representative is implementation dependent). Note: if x and y originally belong to the same set, then **Union(x,y)** has no effect.

We’ll see some applications of this ADT when we talk about graphs. For now, just notice that **MAKE-SET** and **UNION** are used for building sets and **FIND-SET** is good for testing whether two elements are in the same set: elements x and y are in the same set iff **FIND-SET(x) = FIND-SET(y)**.

2 Data Structures for Union-Find

1. **Linked lists:** Represent each set by a linked list, where each node is an element. The representative element is the head of the list. Each node contains a pointer back to the head. The head also contains a pointer to the tail. We can implement the operations as follows ($list_x$ is the list containing x and $list_y$ is the list containing y):

- **MAKE-SET(x):** Just create a list of one node containing x . Time: $O(1)$.

- **FIND-SET(x)**: Just follow x 's pointer back to the head and return the head. Time: $O(1)$.
- **UNION(x,y)**: Append $list_y$ to the end of $list_x$. Since we can find the head of $list_y$ and the tail of $list_x$ in constant time, this takes $O(1)$ time. The representative of this combined list is the head of $list_x$, but the nodes of $list_y$ still point to the head of $list_y$. To update them to point to the head of $list_x$, it takes time $\Theta(\text{length of } list_y)$.

The worst-case sequence complexity for m of these operations is certainly $O(m^2)$: no list will contain more than m elements since we can't call **MAKE-SET** more than m times. The most expensive operation is **UNION**; if we call this m times on lists of length m , it will take time $O(m^2)$. Obviously this an overestimate of the time since we can't call both **MAKE-SET** and **UNION** m times.

We can show, however, that the worst-case sequence complexity of m operations is $\Omega(m^2)$. To do this, we have to give a sequence that will take time $\Omega(m^2)$: start by calling **MAKE-SET** $m/2 + 1$ times on elements $x_1, x_2, \dots, x_{m/2+1}$. Now do the loop:

```
for i = 2 to m/2 do
  UNION (x_i, x_1)
```

This will create a longer and longer list that keeps getting appended to a single element. The execution of the loop takes time $\Theta(m^2)$.

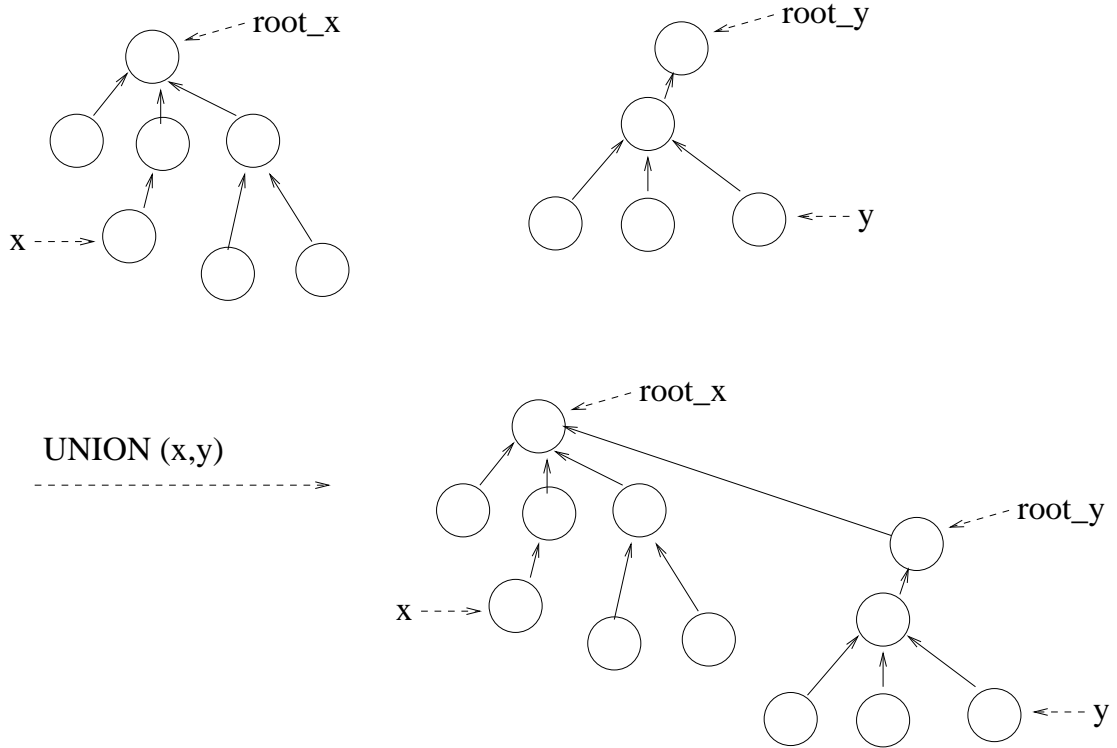
2. **Linked lists with union-by-weight**: Everything remains the same except we will store the length of each linked list at the head. Whenever we do a **UNION**, we will take the shorter list and append it to the longer list. So, **UNION(x,y)** will no longer take $O(\text{length of } list_y)$, but rather $O(\min\{\text{length}(list_x), \text{length}(list_y)\})$. This type of union is called "union-by-weight" (where "weight" just refers to the length of the list).

It might seem like union-by-weight doesn't make much of a difference, but it greatly affects the worst-case sequence complexity. Consider a sequence of m operations and let n be the number of **MAKE-SET** operations in the sequence (so there are never more than n elements in total). **UNION** is the only expensive operation and it's expensive because of the number of times we might have to update pointers to the head of the list. For some arbitrary element x , we want to prove an upper bound on the number of times that x 's head pointer can be updated during the sequence of m operations. Note that this happens only when $list_x$ is unioned with a list that is no shorter (because we update pointers only for the shorter list). This means that each time x 's back pointer is updated, x 's new list is at least twice the size of its old list. But the length of $list_x$ can double only $\log n$ times before it has length greater than n (which it can't have because there are only n elements). So we update x 's head pointer at most $\log n$ times. Since x could be any of n possible elements, we do total of at most $n \log n$ pointer updates. So the cost for all the **UNION**'s in the sequence is $O(n \log n)$. The other operations can cost at most $O(m)$ so the total worst-case sequence complexity is $O(m + n \log n)$.

3. **Trees**: Represent each set by a tree, where each element points to its parent and the root points back to itself. The representative of a set is the root. Note that the trees are not necessarily binary trees: the number of children of a node can be arbitrarily large (or small).

- **MAKE-SET(x)**: Just create a tree with a single node x . Time: $O(1)$.

- **FIND-SET(x)**: Follow the parent pointers from x until you reach the root. Return root. Time: $\Theta(\text{height of tree})$.
- **UNION(x,y)**: Let $root_x$ be the root of the tree containing x , $tree_x$, and let $root_y$ be the root of the tree containing y , $tree_y$. We can find $root_x$ and $root_y$ using **FIND-SET(x)** and **FIND-SET(y)**. Then make $root_y$ a child of root x . Since we have to do both **FIND-SETs**, the running time is $\Theta(\max\{\text{height}(tree_x), \text{height}(tree_y)\})$.

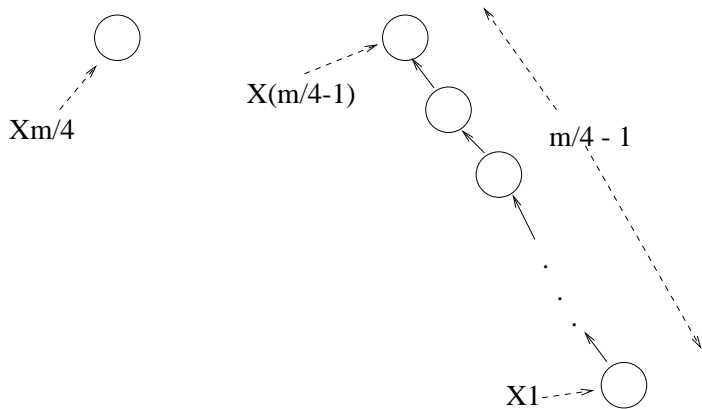


The worst-case sequence complexity for m operations is just like the linked list case, since we can create a tree which is just a list:

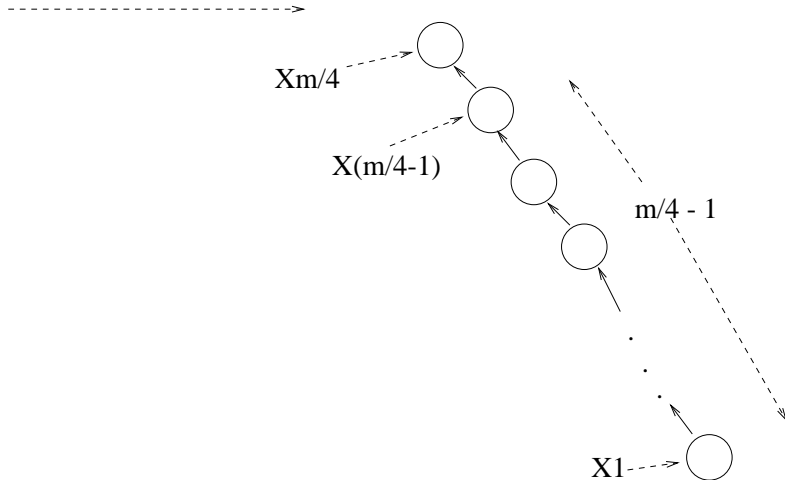
```

for i = 1 to m/4 do
  MAKE-SET(x_i)
for i = 1 to m/4 - 1 do
  UNION(x_(i+1), x_i)

```



UNION($X_{m/4}$, $X_{(m/4-1)}$)



Creating this tree takes $m/4$ MAKE-SET operations and $m/4 - 1$ UNION operations. The running time for $m/2 + 1$ FIND-SET operations on x_1 now is $m/4(m/2 + 1) = \Theta(m^2)$. How do we know there is not a sequence of operations that takes longer than $\Theta(m^2)$?

4. **Trees with union-by-rank:** We improved the performance of the linked-list implementation by using “weight” or “size” information during UNION. We will do the same thing for trees, using “rank” information. The rank of a tree is an integer that will be stored at the root:

- MAKE-SET(x): Same as before. Set $rank = 0$.
- UNION(x, y): If $rank(tree_x) \geq rank(tree_y)$ then make $root_y$ a child of $root_x$. Otherwise, make $root_x$ a child of $root_y$. The rank of the combined tree is $rank(tree_x) + 1$ if $rank(tree_x) = rank(tree_y)$, and $\max\{rank(tree_x), rank(tree_y)\}$ otherwise. The running time is still $\Theta(\max\{height(tree_x), height(tree_y)\})$.
- FIND-SET(x): Same as before.

We can prove two things about union-by-rank:

- (a) The rank of any tree created by a sequence of these operations is equal to its height.

- (b) The rank of any tree created by a sequence of these operations is $O(\log n)$, where n is the number of MAKE-SETs in the sequence.

These two facts imply that the running times of FIND-SET and UNION are $O(\log n)$, so the worst-case sequence complexity of m operations is $O(m \log n)$.

5. **Trees with union-by-rank and path compression:** In addition to doing union-by-rank, there is another way to improve the tree implementation of Union-Find: When performing FIND-SET(x), keep track of the nodes visited on the path from x to $root_x$ (in a stack or queue), and once the root is found, update the parent pointers of each of these nodes to point directly to the root. This at most doubles the running time of the current FIND-SET operation, but it can speed up future FIND-SETs. This technique is called “path compression.”

This is the state-of-the-art data structure for Union-Find. Its worst case sequence complexity is $O(m \log^* n)$ (see section 22.4 of the text for a proof). The function $\log^* n$ is a *very* slowly growing function; it is equal to the number of times you need to apply \log to n before the answer is less than 1. For example, if $n = 15$, then $3 < \log n < 4$, so $1 < \log \log n < 2$ and $\log \log \log n < 1$. So $\log^* n = 3$.