Disjoint Sets

• A disjoint set data structure maintains a collection \( \{S_1, S_2, \ldots, S_k\} \) of disjoint dynamic sets
• Each set is identified by a representative which is a member of that set
• Let’s call the members of the sets objects.

Today’s Outline

• Data Structures for Disjoint Sets

Operations

We want to support the following operations:

• Make-Set(\( x \)): creates a new set whose only member (and representative) is \( x \)
• Union(\( x, y \)): unites the sets that contain \( x \) and \( y \) (call them \( S_x \) and \( S_y \)) into a new set that is \( S_x \cup S_y \). The new set is added to the data structure while \( S_x \) and \( S_y \) are deleted. The representative of the new set is any member of the set.
• Find-Set(\( x \)): Returns a pointer to the representative of the (unique) set containing \( x \)
Analysis

- We will analyze this data structure in terms of two parameters:
  1. $n$, the number of Make-Set operations
  2. $m$, the total number of Make-Set, Union, and Find-Set operations
- Since the sets are always disjoint, each Union operation reduces the number of sets by 1
- So after $n - 1$ Union operations, only one set remains
- Thus the number of Union operations is at most $n - 1$

- Note also that since the Make-Set operations are included in the total number of operations, we know that $m \geq n$
- We will in general assume that the Make-Set operations are the first $n$ performed

Application

- Friendster is a web site which keeps track of a social network
- When you are invited to join Friendster, you become part of the social network of the person who invited you to join
- In other words, you can read profiles of people who are friends of your initial friend, or friends of friends of your initial friend, etc., etc.
- If you forge links to new people in Friendster, then your social network grows accordingly

- Consider a simplified version of Friendster
- Every object is a person and every set represents a social network
- Whenever a person in the set $S_1$ forges a link to a person in the set $S_2$, then we want to create a new larger social network $S_1 \cup S_2$ (and delete $S_1$ and $S_2$)
- For obvious reasons, we want these operation of Union, Make-Set and Find-Set to be as fast as possible
Example

- Make-Set(“Bob”), Make-Set(“Sue”), Make-Set(“Jane”), Make-Set(“Joe”)
- Union(“Bob”, “Joe”)  
  there are now three sets \{Bob, Joe\}, \{Jane\}, \{Sue\}
- Union(“Jane”, “Sue”)  
  there are now two sets \{Bob, Joe\}, \{Jane, Sue\}
- Union(“Bob”, “Jane”)  
  there is now one set \{Bob, Joe, Jane, Sue\}

Applications

- We will also see that this data structure is used in Kruskal’s minimum spanning tree algorithm
- Another application is maintaining the connected components of a graph as new vertices and edges are added

Tree Implementation

- One of the easiest ways to store sets is using trees.
- Each object points to another object, called its parent, except for the leader of each set, which points to itself and thus is the root of the tree.

  - Make-Set is trivial (we just create one root node)
  - Find-Set traverses the parent pointers up to the leader (the root node).
  - Union just redirects the parent pointer of one leader to the other.

(Notice that unlike most tree data structures, objects do not have pointers down to their children.)
### Algorithms

```plaintext
Make-Set(x)
    parent(x) = x;
}

Find-Set(x)
    while(x!=parent(x))
        x = parent(x);
    return x;
}

Union(x,y)
    xParent = Find-Set(x);
    yParent = Find-Set(y);
    parent(yParent) = xParent;
```

### Analysis

- Make-Set takes $\Theta(1)$ time
- Union takes $\Theta(1)$ time in addition to the calls to Find-Set
- The running time of Find-Set is proportional to the depth of $x$ in the tree. In the worst case, this could be $\Theta(n)$ time

### Example

Merging two sets stored as trees. Arrows point to parents. The shaded node has a new parent.

### Problem

- Problem: The running time of Find-Set is very slow
- Q: Is there some way to speed this up?
- A: Yes we can ensure that the depths of our trees remain small
- We can do this by using the following strategy when merging two trees: we make the root of the tree with fewer nodes a child of the tree with more nodes
- This means that we need to always store the number of nodes in each tree, but this is easy
The Code

Make-Set(x){
    parent(x) = x;
    size(x) = 1;
}

Union(x,y){
    xRep = Find-Set(x);
    yRep = Find-Set(y);
    if (size(xRep)) > size(yRep)){
        parent(yRep) = xRep;
        size(xRep) = size(xRep) + size(yRep);
    }else{
        parent(xRep) = yRep;
        size(yRep) = size(yRep) + size(xRep);
    }
}

Analysis

- It turns out that for these algorithms, all the functions run in $O(\log n)$ time
- We will be showing this is the case in the In-Class exercise
- We will show this by showing that the heights of all the trees are always logarithmic in the number of nodes in the tree

In-Class Exercise

- We will show that the depth of our trees are no more than $O(\log x)$ where $x$ is the number of nodes in the tree
- We will show this using proof by induction on, $x$, the number of nodes in the tree
- We will consider a tree with $x$ nodes and, using the inductive hypothesis (and facts about our algs), show that it has a height of of $O(\log x)$

The Facts

- Let $T$ be a tree with $x$ nodes that was created by a call to the Union Algorithm
- Note that $T$ must have been created by merging two trees $T_1$ and $T_2$
- Let $T_2$ be the tree with the smaller number of nodes
- Then the root of $T$ is the root of $T_1$ and a child of this root is the root of the tree $T_2$
- Key fact: the number of nodes in $T_2$ is no more than $x/2$
In-Class Exercise

To prove: Any tree \( T \) with \( x \) nodes, created by our algorithms, has depth no more than \( \log x \)

- Q1: Show the base case \( (x = 1) \)
- Q2: What is the inductive hypothesis?
- Q3: Complete the proof by giving the inductive step. (hint: note that depth\( (T) = \max(\text{depth}(T1), \text{depth}(T2) + 1) \)

Problem

- Q: \( O(\log n) \) per operation is not bad but can we do better?
- A: Yes we can actually do much better but it’s going to take some cleverness (and amortized analysis)

Shallow Threaded Trees

- One good idea is to just have every object keep a pointer to the leader of it’s set
- In other words, each set is represented by a tree of depth 1
- Then Make-Set and Find-Set are completely trivial, and they both take \( O(1) \) time
- Q: What about the Union operation?

Union

- To do a union, we need to set all the leader pointers of one set to point to the leader of the other set
- To do this, we need a way to visit all the nodes in one of the sets
- We can do this easily by “threading” a linked list through each set starting with the sets leaders
- The threads of two sets can be merged by the Union algorithm in constant time
The Code

```c
Make-Set(x){
    leader(x) = x;
    next(x) = NULL;
}
Find-Set(x){
    return leader(x);
}
```

```c
Union(x,y){
    xRep = Find-Set(x);
    yRep = Find-Set(y);
    leader(y) = xRep;
    while(next(y)!=NULL){
        y = next(y);
        leader(y) = xRep;
    }
    next(y) = next(xRep);
    next(xRep) = yRep;
}
```

Analysis

- Worst case time of Union is a constant times the size of the larger set
- So if we merge a one-element set with a \( n \) element set, the run time can be \( \Theta(n) \)
- In the worst case, it's easy to see that \( n \) operations can take \( \Theta(n^2) \) time for this alg

Example

Merging two sets stored as threaded trees. Bold arrows point to leaders; lighter arrows form the threads. Shaded nodes have a new leader.
Problem

• The main problem here is that in the worst case, we always get unlucky and choose to update the leader pointers of the larger set
• Instead let’s purposefully choose to update the leader pointers of the smaller set
• This will require us to keep track of the sizes of all the sets, but this is not difficult

The Code

Make-Weighted-Set(x)
{
    leader(x) = x;
    next(x) = NULL;
    size(x) = 1;
}

Weighted-Union(x,y)
{
    xRep = Find-Set(x);
    yRep = Find-Set(y)
    if(size(xRep)>size(yRep){
        Union(xRep,yRep);
        size(xRep) = size(xRep) + size(yRep);
    }else{
        Union(yRep,xRep);
        size(yRep) = size(xRep) + size(yRep);
    }
}

Analysis

• The Weighted-Union algorithm still takes $\Theta(n)$ time to merge two $n$ element sets
• However in an amortized sense, it is more efficient:
• A sequence of $m$ Make-Weighted-Set operations and $n$ Weighted-Union operations takes $O(m + n \log n)$ time in the worst case.