CS 61B Spring 2025 — Disjoint Sets, ADTs, BSTs

Discussion 06: March 03, 2025

1 Disjoint Sets a.k.a. Union Find

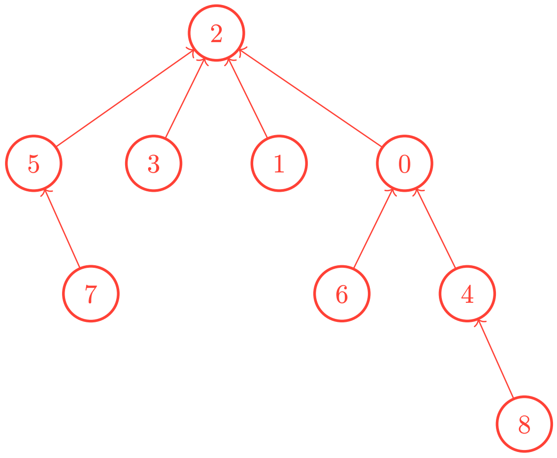
In lecture, we discussed the Disjoint Sets ADT. Some authors call this the Union Find ADT. Today, we will use union find terminology so that you have seen both.

(a) Assume we have nine items, represented by integers 0 through 8. All items are initially unconnected to each other. Draw the union find tree, draw its array representation after the series of `connect()` and `find()` operations, and write down the result of `find()` operations using **WeightedQuickUnion** without path compression. **Break ties by choosing the smaller integer to be the root.**

Note: `find(x)` returns the root of the tree for item `x`.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);
connect(0, 6);
connect(6, 4);
connect(6, 3);
find(8);
find(6);
```

Solution: `find()` returns 2, 2, 2 respectively.
The array is `[2, 2, -9, 2, 0, 2, 0, 5, 4]`.



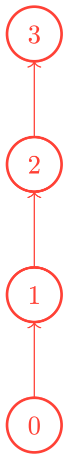
A walkthrough of how we arrive at this result can be found on the website, linked here.

Below is an implementation of the **find** function for a Disjoint Set. Given an integer **val**, **find(val)** returns the root value of the set **val** is in. The helper method **parent(int val)** returns the direct parent of **val** in the Disjoint Set representation. Assume that this implementation only uses **QuickUnion**.

```
public int find(int val) {
    int p = parent(val);
    if (p == -1) {
        return val;
    } else {
        int root = find(p);
        return root;
    }
}
```

(b) If **N** is the number of nodes in the set, what is the runtime of **find** in the worst case? Draw out the structure of the Disjoint Set representation for this worst case.

$\Theta(N)$
The worst case would occur if we have to traverse up $N-1$ nodes to find the root set representative as shown below for **find(0)**. Suppose we started out with elements 0, 1, 2, and 3. Consider the following Disjoint Set:



index	0	1	2	3
parent	1	2	3	-1

The worst case runtime of **find** is $\Theta(N)$, for **find(0)**. Since this implementation does not use WeightQuickUnion, this could potentially arise if we unioned 0 to 1, setting 1 as the root, then unioning 1 to 2, setting 2 as the root, and finally unioning 2 to 3, setting 3 as the root (try drawing this out for yourself!). WQU solves this "spindly set" problem by ensuring that the smaller set is merged into the larger one, so when we try unioning 1 to 2, 1 must be the root and not the 2.

Note that this function also does not implement path compression, making the disjoint set more susceptible to worst cases like this.

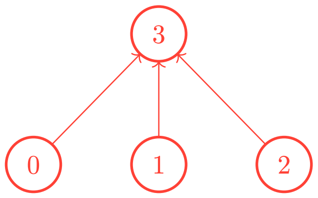
(c) Using a function **setParent(int val, int newParent)**, which updates the value of **val**'s parent to **newParent**, modify **find** to achieve a faster runtime using path compression. You may add at most one line to the provided implementation.

Solution:

```
public int find(int val) {
    int p = parent(val);
    if (p == -1) {
        return val;
    } else {
        int root = find(p);
        setParent(val, root); // sets the val's parent to be the root of the set.
        return root;
    }
}
```

Although our worst case would still be $\Theta(N)$ runtime as in the call to **`find(0)`** above. However, after one call to **`find(0)`**, the structure of the disjoint set would change so subsequent calls to **`find`** would be completed in amortized $O(\log^*(N))$.

Here's the structure of the set after one call to **`find(0)`**:

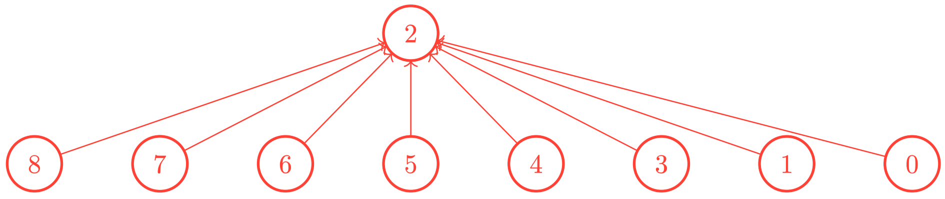


index	0	1	2	3
parent	3	3	3	-1

(d) *Extra Practice:* Draw out the tree and array representation for the following WeightedQuickUnion with path compression that has 9 elements from 0 to 8. Break ties by choosing the smaller integer to be root.

```
connect(2, 3);
connect(1, 2);
connect(5, 7);
connect(8, 4);
connect(7, 2);
find(3);
connect(0, 6);
connect(7, 4);
connect(6, 3);
find(8);
find(6);
```

$[2, 2, -9, 2, 2, 2, 2, 2, 2]$



2 ADT Matchmaking

Match each task to the best Abstract Data Type for the job and justify your answer (ie. explain why other options would be less ideal). The options are **`List, Map, Queue, Set, and Stack`**. Each ADT will be used once.

1. You want to keep track of all the unique users who have logged on to your system.

   You should use a set because we only want to keep track of unique users (i.e. if a user logs on twice, they shouldn't show up in our data structure twice). Additionally, our task doesn't seem to require that the structure is ordered.

2. You are creating a version control system and want to associate each file name with a Blob.

   You should use a map. Maps naturally let you pair a key and value, and here we could have the file name be the key, and the blob be the value.

3. We are grading a pile of exams and want to grade starting from the top of the pile (*Hint:* what order do we pile papers in?).

   We should use a Stack. When papers are added to a pile, the top of the pile is the last paper added. Since we want to grade the top of the pile first, it makes sense for us to use a last-in-first-out (LIFO) approach in which we continually pop papers off the top of our Stack as we grade them.

4. We are running a server and want to service clients in the order they arrive.

   We should use a Queue. We can push clients to the front of the Queue as they arrive, and pop them off the Queue as we service them.

5. We have a lot of books at our library and we want our website to display them in some sorted order. We have multiple copies of some books and we want each listing to be separate.

   We should use a List because a List is an ordered collection of items. Additionally, we need to allow for duplicate items because we have multiple copies of some books.

3 BST Asymptotics

Below we define the **find** method of a BST (Binary Search Tree) as in lecture, which returns the BST rooted at the node with key **sk** in our overall BST. In this setup, assume a **BST** has a **key** (the value of the tree root) and then pointers to two other child BSTs, **left** and **right**.

```
public static BST find(BST tree, Key sk) {
    if (tree == null) {
        return null;
    }
    if (sk.compareTo(tree.key) == 0) {
        return tree;
    } else if (sk.compareTo(tree.key) < 0) {
        return find(tree.left, sk);
    } else {
        return find(tree.right, sk);
    }
}
```

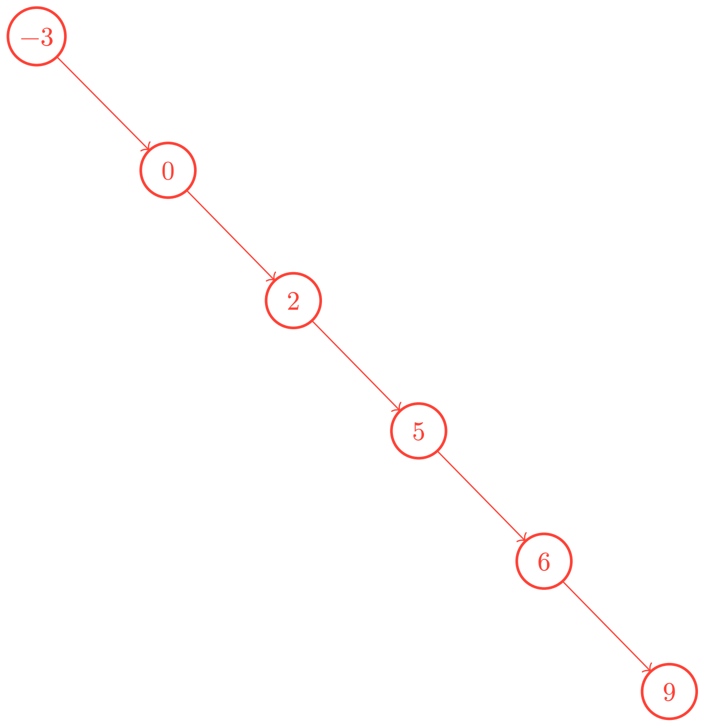
(a) Assume our BST is perfectly bushy. What is the runtime of a single **find** operation in terms of N, the number of nodes in the tree? Can we generalize the runtime of **find** to a theta bound?

Find operations on a perfectly bushy BST take $O(\log(N))$ time, as the height of a perfectly bushy BST is $\log(N)$. In the worst case scenario, the key we're looking at is all the way at a leaf, so we have to traverse a path from root to leaf of length $\log(N)$.

We cannot generalize the runtime of **find** to a theta bound because the lower and upper bounds are different. It is lower-bounded by $\Omega(1)$ (the case where the key we are looking for is at the root of the BST provided) and upper-bounded by $O(\log(N))$ (mentioned above as the path from root to leaf). Therefore, there is no theta bound for **find**.

(b) Say we have an empty BST and want to insert the keys `[6, 2, 5, 9, 0, -3]` (in some order). In what order should we insert the keys into the BST such that the runtime of a single **find** operation after all keys are inserted is $O(N)$? Draw out the resulting BST.

We should insert the keys in ascending sorted order: `[-3, 0, 2, 5, 6, 9]`. This results in a perfectly linear BST, which means that the longest path for `find` (from root to leaf) traverses every single node in the BST. The resulting BST looks like a direct chain of nodes:



Alternatively, we could insert the keys in descending sorted order, which also results in a perfectly linear BST (but the keys would chain left from $9 \to 6 \to 5 \to 2 \to 0 \to -3$).