Prepared by Linan Qiu <lq2137@columbia.edu>

Disjoint Sets

One of my favorite childhood pasttimes is to sort my toy figurines. I'd group them by their universe (Pokemon vs Digimon vs others), species (humans vs others) and so on. Now let's say another kid comes along after I've grouped the figures into a set (say by universe), picks up Pikachu and Bulbasaur, and asks me, "Hey Linan, are these two figures from the same group?" I would probably say yes. However, I probably would answer wrongly if I had a thousand figurines (which was not true due to my limited budget).

Wouldn't it be nice if there was a data structure to help me answer that? Enter disjoint sets.

Disjoint Set Operations

Disjoint sets answer the question: Are `x` and `y` part of the same group? Additionally, it does organizes `x`, `y` and other elements by grouping items together. We can boil these down into 3 operations:

- `makeSet`: Creates a new subset with a single element
- `find`: Determines which subset a particular element is in
- `union`: joins two subsets into a single subset

Let's say I had 6 elements to begin with:

`pikachu, bulbasaur, charmander, squirtle, galadriel, legolas`

For the uninitiated, the first 4 are Pokemons and the other 2 are elves from Lord of the Rings.

A disjoint set would first give each element its own set (i.e. a set that contains only the element itself). We do so by calling `makeSet` on each of the elements. i.e. `makeSet(pikachu)`, `makeSet(squirtle)` etc...

`[pikachu] [bulbasaur] [charmander] [squirtle] [galadriel] [legolas]`

Then, if I call `union(pikachu, bulbasaur)`, `bulbasaur` gets merged into `pikachu`'s set.

`[pikachu, bulbasaur] [charmander] [squirtle] [galadriel] [legolas]`

Similarly, if I call `union(charmander, squirtle)`, this happens:

`[pikachu, bulbasaur] [charmander, squirtle] [galadriel] [legolas]`

Now if I call `union(pikachu, squirtle)`, I would combine the set that `pikachu` belongs to with the set that `squirtle` belongs to, resulting in this:

`[pikachu, bulbasaur, charmander, squirtle] [galadriel] [legolas]`

Finally, I finish by grouping the two elves together `union(galadriel, legolas)`

`[pikachu, bulbasaur, charmander, squirtle] [galadriel, legolas]`

Now what `find(data)` does is to retrieve the representative element of the set that `data` belongs to. What does that mean? Let's say that I make the first element in each list the **representative element**. That means the set is defined by that particular element. Hence the first set is `pikachu`'s set, and the second is `galadriel`'s set.

Then, `find(bulbasaur)` should return `pikachu` because `bulbasaur` belongs in `pikachu`'s set. Similarly, `find(charmander)` should return `pikachu` as well since `charmander` belongs in `pikachu`'s set. `find(pikachu)` would return `pikachu` since `pikachu` belongs in its own set. `find(legolas)` would return `galadriel`.

Now I can check if two elements belong to same set by comparing the result of `find`. If I want to check that `charmander` and `bulbasaur` belong in the same set, all I have to do is check `find(bulbasaur).equals(find(charmander))`.

With this, we can define an interface for `DisjointSet`.

```java
// DisjointSet.java

public interface DisjointSet<T> {
  public void makeSet(T data);

  public void union(T data1, T data2);

  public T find(T data);
}
```

Disjoint Set Linked List (`DisjointSetLinkedList.java`)

We can implement a Disjoint Set using linked lists. We are unable to make use of `java.util.LinkedList` since we need finer control over the `prev` pointers of each node in the linked list. Hence, we create our own linked list. This is accomplished by creating a `Node` class. This `Node` inner class wraps a data element.

```java
// in DisjointSetLinkedList.java

public class Node<R> {
  public Node<R> prev;
  public R data;

  public Node(Node<R> prev, R data) {
    this.prev = prev;
```

```
    this.data = data;
  }

  public String toString() {
    return data.toString();
  }
}
```

You have seen this before during the early weeks of the course (oh how time flies). This should be no stranger to you.

We also use a `HashMap<T, Node<T>>` to keep track of the data inside the `Node` and the `Node` itself. Useful if we want to modify a `Node` given only its `data`. That is, we can use `T` to look up its corresponding `Node<T>`. This is a slight memory trade-off but one that we have to make.

The key idea in this implementation is that each set is represented by a linked list. The root of each linked list (the `Node` with `.prev == null`) is the **representative element** of the set.

`makeSet`

`makeSet` in $O(1)$ creates a new list consisting of only a single element.

```
// in DisjointSetLinkedList.java

public void makeSet(T data) {
  if (nodeReference.containsKey(data)) {
    System.err.println("Duplicate element added. Ignoring.");
  }

  Node<T> root = new Node<>(null, data);
  nodeReference.put(data, root);
}
```

We also check for duplicates. Since we are using a `HashMap<T, Node<T>>` to store references to nodes, we cannot allow duplicates. This list (or rather a single node) consists only of a single root node. Hence its `.prev` is `null` as we have done in the constructor. We also add this entry to the `nodeReference` map.

Note that the `DisjointSetLinkedList` class itself does not need to keep references to the nodes other than inside the map. That's the function of our map! Think of it as a quick "phonebook" index to access each node.

This process is $O(1)$ because we are simply creating a node and putting it inside a map.

`find`

`find` in $O(N)$ traverses a list backwards from a given element.

```java
// in DisjointSetLinkedList.java

public T find(T data) {
  Node<T> node = nodeReference.get(data);
  if (node.prev == null) {
    return data;
  } else {
    return find(node.prev.data);
  }
}
```

Since we defined the root node of a linked list to be the representative element, we must traverse backwards from a given node to find the representative element. For example, in our earlier example:

`["pikachu", "bulbasaur", "charmander", "squirtle"] ["galadriel", "legolas"]`

If I want to `find("charmander")`, first I will have to get the `Node<String>` that contains `"charmander"`. Then, I will traverse backwards (to the `Node<String>` holding on to `"bulbasaur"`, then finally to the `Node<String>` holding on to `"pikachu"` whose `.prev` is `null`). I arrive at the root node containing `"pikachu"`, and return `"pikachu"` as the result of `find("charmander")`, indicating that "`"charmander"` belongs to `"pikachu"`'s set".

This process is $O(N)$ since we have to traverse, in the worst case, an entire linked list.

`union`

`union` in $O(N)$, append a list to another.

```java
// in DisjointSetLinkedList.java

public void union(T data1, T data2) {
  Node<T> data1Node = nodeReference.get(data1);

  T root2 = find(data2);
  Node<T> root2Node = nodeReference.get(root2);

  root2Node.prev = data1Node;
}
```

`["pikachu", "bulbasaur", "charmander", "squirtle"] ["galadriel", "legolas"]`

Let's say that I want to merge the set containing `"squirtle"` with the set containing `"galadriel"`. What I can do is to get the Node<String> of the representative element of `"galadriel"`, which happens to be the node holding on to `"galadriel"` itself (since `"galadriel"` is the representative element). Then, I can set `"galadriel"`'s Node<String>'s `.prev` to be the node for `"squirtle"`, in so doing appending `"galadriel"`'s linked list to the node holding `"squirtle"`.

```
["pikachu", "bulbasaur", "charmander", "squirtle", "galadriel", "legolas"]
```

This makes them all one set, which is exactly what I wanted.

Now let's reset our disjoint set to this state:

```
["pikachu", "bulbasaur", "charmander", "squirtle"] ["galadriel", "legolas"]
```

What if I merged the set containing `"bulbasaur"` with the set containing `"legolas"` instead? We'd get a slightly different result that yields us pretty much the same answer. We'd traverse up the linked list `"legolas"` is in due to the `find(data2)` line in the code. We'd arrive at the node containing `"galadriel"` which is the root node (the representative element's node). We'd append this node to `"bulbasaur"`'s node, causing a branch to happen in the linked list.

```
["pikachu", "bulbasaur", "charmander", "squirtle"]
                         "galadriel", "legolas"]
```

That means `"charmander"`'s node's `.prev` would be `"bulbasaur`'s node. At the same time, `"galadriel"`'s node's `.prev` would be `"bulbasaur"`'s node. This would be a huge problem in a linked list implementation, but isn't a problem at all for ours. In fact, calling `find` on any element in this new structure would result in the same result (`"pikachu"`) as the previous example of `union("squirtle", "galadriel")` when the set is a single linked list. Again, this works because all we have to do is to look up the root element from a linked list.

This is $O(N)$ because of the `find` operation.

> Technically this implementation violates a linked list, since it will create linked lists that branch into two. This kind of makes this a tree. That's a great point! In fact, that's exactly what we will do in an improved version later on.
>
> We can actually make this a proper linked list by using a doubly linked list, and traversing to the tail of `data1` each time we want to append another list. However, that'd actually **increase** in the runtime. Instead, this little *hack* is reducing our runtime (and simplifying our implementation by allowing us to use a singly linked list instead).

toString

And finally a kickass `toString()` that maps `data` in each `Node` to their root's `data`.

```java
// in DisjointSetLinkedList.java

public String toString() {
  HashMap<T, LinkedList<T>> rootNode = new HashMap<>();

  for (T data : nodeReference.keySet()) {
    T root = find(data);

    if (!rootNode.containsKey(root)) {
      rootNode.put(root, new LinkedList<T>());
    }

    rootNode.get(root).add(data);
  }

  return rootNode.toString();
}
```

This prints out the entire set in a nice format (see the `main` method below).

main

Now we test it using a `main` method!

```java
// in the main method of DisjointSetLinkedList.java

String poke1 = "pikachu";
String poke2 = "bulbasaur";
String poke3 = "charmander";
String poke4 = "squirtle";
String elf1 = "galadriel";
String elf2 = "legolas";
DisjointSetLinkedList<String> set = new DisjointSetLinkedList<>();
set.makeSet(poke1);
set.makeSet(poke2);
set.makeSet(poke3);
set.makeSet(poke4);
set.makeSet(elf1);
set.makeSet(elf2);
System.out.println(set);
// {squirtle=[squirtle], bulbasaur=[bulbasaur], legolas=[legolas],
```

```java
// pikachu=[pikachu], charmander=[charmander], galadriel=[galadriel]}

set.union(poke1, poke2);
System.out.println(set.find(poke1) + " " + set.find(poke2));
// pikachu pikachu
System.out.println(set);
// {squirtle=[squirtle], legolas=[legolas], pikachu=[bulbasaur, pikachu],
// charmander=[charmander], galadriel=[galadriel]}

set.union(poke3, poke4);
System.out.println(set);
// {legolas=[legolas], pikachu=[bulbasaur, pikachu], charmander=[squirtle,
// charmander], galadriel=[galadriel]}

set.union(poke1, poke3);
set.union(elf1, elf2);
System.out.println(set);
// {pikachu=[squirtle, bulbasaur, pikachu, charmander], galadriel=[legolas,
// galadriel]}
```

Disjoint Set Forest

Instead of using linked lists, we can use trees to achieve higher performance gains on `find`. Realize that in `find`, the depth of the tree (or, equivalently, the length of the linked list if your mode of thought is still stuck in the previous chapter) dictates the runtime. If we can use a tree to represent nodes, we can traverse back up to the root in $\log N$ time. This is exactly what we will do.

We keep the same `Node` class. We just change the `prev` field to `parent` because well tree. Furthermore, we add a `int rank` field. This will be used later. You can safely ignore it for the next page or so.

```java
// DisjointSetForest.java

public class DisjointSetForest<T> implements DisjointSet<T> {

  private HashMap<T, Node<T>> nodeReference;

  public DisjointSetForest() {
    nodeReference = new HashMap<>();
  }

  public class Node<R> {
    public Node<R> parent;
    public R data;
    public int rank;
```

```java
    public Node(Node<R> parent, R data, int rank) {
      this.parent = parent;
      this.data = data;
      this.rank = rank;
    }

    public String toString() {
      return data.toString();
    }
  }
}
```

makeSet

`makeSet` in $O(1)$ creates an empty tree.

```java
// in DisjointSetForest.java

public void makeSet(T data) {
  if (nodeReference.containsKey(data)) {
    System.err.println("Duplicate element added. Ignoring.");
  }

  Node<T> root = new Node<>(null, data, 0);
  nodeReference.put(data, root);
}
```

This is exactly the same as the linked list implementation.

find

```java
// in DisjointSetForest.java

public T find(T data) {
  Node<T> node = nodeReference.get(data);

  if (node.parent == null) {
    return data;

  } else {
    return find(node.parent.data);
  }
}
```

This code is basically equivalent to the linked list code. However, we can improve this now that we can think about this in terms of trees. Let's say we're traversing up this linked list to find the root:

```
["pikachu", "bulbasaur", "charmander", "squirtle"]
```

Let's say we're starting at the node containing `"squirtle"`. Won't it be nice if at the end of the traversal, we get this instead:

```
["pikachu", "bulbasaur"]
            "charmander"]
            "squirtle"]
```

That means the node for `"bulbasaur"`'s `.prev` will be the node for `"pikachu"`. However, the node for `"charmander"`'s `.prev` will also be modified to `"pikachu"`, as will the node for `"squirtle"`. This structure will yield similar results on `find` as the single flat list earlier. However, it is much faster, since we have to skip through fewer `.prev`s in the traversal.

This is known as **path compression**. We can accomplish this by setting the nodes for `"bulbasaur"`, `"charmander"`, and `"squirtle"`'s `.prevs` to the node for `"pikachu"`. We can do this recursively:

```java
// in DisjointSetForest.java

public T find(T data) {
  Node<T> node = nodeReference.get(data);

  if (node.parent == null) {
    return data;

  } else {
    // path compression
    node.parent = nodeReference.get(find(node.parent.data));
    return node.parent.data;
  }
}
```

The base condition of the recursion will return `"pikachu"` in our case. That return gets propagated to the prior layers of recursion. Then, `"pikachu"`'s node gets set as the `.prev` for all the nodes traversed.

This means that every time we call `find` on the disjoint set, we are actually optimizing it / cleaning it up.

`find` now runs in $O(\log N)$ in depth of tree, and we know for sure that the base of the log will be really low due to path compression.

union

Since we know that we are dealing with trees, then to merge two sets, we can simply set one tree's root as the `.prev` of the other tree's root.

```
// in DisjointSetForest.java

public void union(T data1, T data2) {
  T root1 = find(data1);
  T root2 = find(data2);

  if (root1.equals(root2)) {
    // same set
    return;
  }

  Node<T> root1Node = nodeReference.get(root1);
  Node<T> root2Node = nodeReference.get(root2);

  // 1 and 2 are not in the same set. merge them via union by rank
  root2Node.parent = root1Node;
}
```

This is exactly what we do here.

However, there's an additional trick we can do. Quote Wikipedia:

> Since it is the depth of the tree that affects the running time, the tree with smaller depth gets added under the root of the deeper tree, which only increases the depth if the depths were equal. In the context of this algorithm, the term rank is used instead of depth since it stops being equal to the depth if path compression (described above) is also used. 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 $r+1$. Just applying this technique alone yields a worst-case running-time of $O(logn)$ for the `union` or `find` operation.

In other words, we attach smaller trees to bigger trees to minimize the depth of trees. Let's say we have a tree `A` of depth 3 and tree `B` of depth 5. Let's say `rootA` is the root node (node of the representative element) of `A` and `rootB` is the root node of `B`. If we set `rootA.prev = rootB`, we are appending tree `A` to `B`, making the new tree still depth 5 (since the A subtree will have a depth of $3+1=4$). If we set `rootB.prev = rootA`, we are appending tree `B` to `A`, making the new tree depth 6 (since the B subtree will have a depth of $5+1=6$). Hence it is always optimal to attach a smaller tree to a bigger tree.

```
// in DisjointSetForest.java
```

```java
public void union(T data1, T data2) {
  T root1 = find(data1);
  T root2 = find(data2);

  if (root1.equals(root2)) {
    // same set
    return;
  }

  Node<T> root1Node = nodeReference.get(root1);
  Node<T> root2Node = nodeReference.get(root2);

  // 1 and 2 are not in the same set. merge them via union by rank
  if (root1Node.rank < root2Node.rank) {
    root1Node.parent = root2Node;

  } else if (root1Node.rank > root2Node.rank) {
    root2Node.parent = root1Node;

  } else {
    root2Node.parent = root1Node;
    root1Node.rank++;
  }
}
```

We implement this using a little `if` condition that is rather trivial.

`union` is $O \log N$ due to the `find` operation.

`toString`

We use the same `toString()` code.

`main`

We use the same main code, and it should run the same.

The source code for `DisjointSet.java` (the interface), `DisjointSetLinkedList.java`, and `DisjointSetForest.java` is available in this same folder.

Applications

If you're curious about the application of disjoint set beyond checking Pokemons and elves, here's some:

- http://cs.stackexchange.com/questions/6308/practical-applications-of-disjoint-set-datastructure
- https://www.wikiwand.com/en/Disjoint-set_data_structure#/Applications

We should totally have set maze generation as an assignment. `:p`