CS 200 fall 2022 Lecture 4 (from OO): more recursion

Another recursive example

Here is an example of another method we wrote, remove first found (abbreviated "rff"), which returns a new SimpleList with the first instance of val removed from the input list (or the input list unchanged if val is not found). We start off again, writing examples:

```
Assert.assertEquals(toSimpleList(15, 111, 17), Lec04.rff(17, toSimpleList(15, 17, 111, 17)));
//                  SimpleList(15, [111, 17])
//                  SimpleList(15, Lec04.rff(17, [17, 111, 17]))
//                  SimpleList(input.first(), Lec04.rff(17, input.rest())
Assert.assertEquals(toSimpleList(    111, 17), Lec04.rff(17, toSimpleList(    17, 111, 17)));
//                               input.rest()

// Example below isn't really informative for seeing the structure of the above examples
// Assert.assertEquals(toSimpleList(__), Lec04.rff(17, toSimpleList(111, 17)));
```

In class, we talked about how the last line isn't really informative for seeing the structure of the above examples, because it doesn't remove the same instance of 17 that the upper two examples do. It seems like, after we find the first instance of 17, we do not need to take the examples further, because we just give back the rest of the list. This translates to the following code:

```
public static SimpleList<Integer> rff(int val, SimpleList<Integer> lst) {
    if (lst == null) {
        return null;
    } else {
        if (lst.first() == val) {
            return lst.rest();
        } else {
            return new SimpleList<Integer>(lst.first(), rff(val, lst.rest()));
        }
    }
}
```

In class, we also talked about the intuition behind each decision we made when writing this code:

- In the **base case** (the case where lst is null), we want to return null. If we step back from the code and think through (in prose) what the method is supposed to do, this makes sense – there's no way any value will exist in an empty list, so we just give back the empty list as a result.
- In the case where the first element of a list is equal to the value we want to remove, we just give back the rest of the list. This also makes sense when we think through it in prose, even without thinking about recursion – if we get a list and the value we're looking to remove is the first element, we just give back the list without that first value, a.k.a. the rest of the list!
- In the remaining case, we need to keep the first value of the input list around (because we're not removing it). We also need to search the rest of the list for the value, in case it exists in the rest of the list. We connect those two

ideas by returning a new SimpleList with the same first element as the input list, and with the result of the recursive call of rff on the rest of the input list as the rest.

Recursion without using SimpleList

We are going to work through this example on paper, first. By hand, merge the two pre-sorted lists into one sorted list. Pay attention to your thought process and the decisions you are making while you do this – what do you notice while you move through the lists?

List 1:

15	33	112

List 2:

17	22	32	111	200

Result:

Copy and paste the following into a new document to see what the contents of the result should be:
[redacted]

When arriving at your result, you may have had a thought process similar to this one:

- Since List 1 and List 2 are sorted, we only have to compare the first element of each list to figure out what the first element of the result list should be. Since 15 is smaller than 17, the first element of the result list should be 15.
- Now we ignore 15 in List 1, since we've already used it, and compare 17 and 33. Since 17 is smaller than 33, the next element of the result should be 17.
- Nowe we ignore 17 in List 2, and compare 22 and 33…

and so on. The takeaway is that we only need to compare two elements at a time, and once we use up an element, we take that element away and continue to merge the *rest* of that list with the other list. This indicates that there's something recursive about this process – we make a decision, and then we continue making similar decisions on problems of slightly smaller size.

Before we code this up, let's think about some possible edge/base cases.

How would you merge:

- An empty list and an empty list?
  Copy/paste for answer: [redacted]

- A sorted list (of any length) and an empty list?
  Copy/paste for answer: [redacted]

We can write these examples out explicitly as JUnit tests (it's also a good idea to include a small non-edge-case example or two, which we've done):

```
Assert.assertEquals(Arrays.asList(15, 17, 22, 32, 33, 111, 112, 200),
       Lec04.merge(Arrays.asList(15, 33, 112), Arrays.asList(17, 22, 32, 111, 200)));
Assert.assertEquals(Arrays.asList(), Lec04.merge(Arrays.asList(), Arrays.asList()));
Assert.assertEquals(Arrays.asList(1, 2, 3),
       Lec04.merge(Arrays.asList(), Arrays.asList(1, 2, 3)));
Assert.assertEquals(Arrays.asList(4, 5, 6),
       Lec04.merge(Arrays.asList(4, 5, 6), Arrays.asList()));
Assert.assertEquals(Arrays.asList(40, 99),
       Lec04.merge(Arrays.asList(99), Arrays.asList(40)));
```

Let's code up this example, starting with the base/edge cases:

```
public static List<Integer> merge(List<Integer> list1, List<Integer> list2) {
   if (list1.size() == 0) {
       // returns a copy of list2
       return new LinkedList<Integer>(list2);
   } else if (list2.size() == 0) {
       return new LinkedList<Integer>(list1);
   }
   ...
```

Notice that, since we're working with Java Lists instead of our SimpleLists, we check if they are empty by checking their size.

Also notice that, instead of returning list2 or list1 directly, we return copies of them by creating a new LinkedList. We'll get to the subtleties of why we do this in future lectures, but for now, when using recursion, it's a good idea to make copies of data instead of returning existing data to avoid possibly strange bugs.

Do we need to check for the case where both list1 and list2 are empty? [redacted]

Now let's consider the rest of the code. Above, we talked about comparing the first element of each list to decide what to put in the result. Let's consider the case where the first element of list1 is smaller than the first element of list2. Then, we need to:

1. Put the first element of list1 into the result
2. Get the "rest" of list1 (i.e. everything in list1 except for the first element) somehow
3. Merge the rest of list1 and all of list 2
4. Append the result of step 3 onto the result

We can translate this to code directly:

```
public static List<Integer> merge(List<Integer> list1, List<Integer> list2) {
   if (list1.size() == 0) {
       // returns a copy of list2
       return new LinkedList<Integer>(list2);
   } else if (list2.size() == 0) {
       return new LinkedList<Integer>(list1);
   } else {
       if (list1.get(0) < list2.get(0)) { // compare the first elements of list1
           List<Integer> result = new LinkedList<Integer>();
           result.add(list1.get(0)); // add the first element of list1 to the result
           List<Integer> list1Rest = list1.subList(1, list1.size());
           // get everything in list1 that is not the first element
           List<Integer> restMerged =  merge(list1Rest, list2);
           // merge the rest of list1 and all of list2 together using a recursive call
           result.addAll(restMerged); // append the above answer to the result
           return result;
       }
       ...
```

In the case that the first element of list1 is not less than the first element of list2, we can do the same thing, but swapped:

```
public static List<Integer> merge(List<Integer> list1, List<Integer> list2) {
   if (list1.size() == 0) {
       // returns a copy of list2
       return new LinkedList<Integer>(list2);
   } else if (list2.size() == 0) {
       return new LinkedList<Integer>(list1);
   } else {
       if (list1.get(0) < list2.get(0)) { // compare the first elements of list1
           List<Integer> result = new LinkedList<Integer>();
           result.add(list1.get(0)); // add the first element of list1 to the result
           List<Integer> list1Rest = list1.subList(1, list1.size());
           // get everything in list1 that is not the first element
           List<Integer> restMerged =  merge(list1Rest, list2);
           // merge the rest of list1 and all of list2 together using a recursive call
           result.addAll(restMerged); // append the above answer to the result
           return result;
       } else {
           // the symmetrical case of the first case
           List<Integer> result = new LinkedList<Integer>();
           result.add(list2.get(0)); // add the first element of list2 to the result
           List<Integer> list2Rest = list2.subList(1, list2.size());
           // get everything in list2 that is not the first element
           List<Integer> restMerged =  merge(list1, list2Rest);
           // merge the rest of list2 and all of list1 together using a recursive call
           result.addAll(restMerged); // append the above answer to the result
           return result;
       }
   }
}
```

Bonus: try to explain to yourself why the code below is equivalent to the code above:

```
public static List<Integer> merge(List<Integer> list1, List<Integer> list2) {
   if (list1.size() == 0) {
       return new LinkedList<Integer>(list2);
   } else if (list2.size() == 0) {
       return new LinkedList<Integer>(list1);
   } else {
       if (list1.get(0) <= list2.get(0)) {
           List<Integer> result = new LinkedList<Integer>();
           result.add(list1.get(0));
           List<Integer> list1Rest = list1.subList(1, list1.size());
           List<Integer> restMerged =  merge(list1Rest, list2);
           result.addAll(restMerged);
           return result;
       } else {
           return merge(list2, list1);
       }
   }
}
```