Lecture 7: Mutable Lists, Conceptually

Contents

1 Built-in Lists in Java **1**

2 Functional vs Java/Python Lists **3**

2.1 Lists in memory 3

2.2 So How Do We Implement LinkedLists for Ourselves? 4

Motivating Question

What's the difference between mutable and immutable lists in memory?

Objectives

By the end of this lecture, you will know:

- How to create a list with Java's LinkedList class
- How to traverse a list with a for loop
- How functional and mutable lists differ from one another in memory

1 Built-in Lists in Java

Last week, we wrote our own implementation of lists, based on the `cons` and `link` constructs in Racket and Pyret (respectively). Like all programming languages, Java also has lists built-in, but they behave somewhat differently. Let's look at a concrete example of how to create a list containing the numbers 5 and 3 (in that order):

```
import java.util.LinkedList;

public class JListExample {

        JListExample(){}

        public static void main(String[] args) {
                LinkedList<Integer> L = new LinkedList<Integer>();
                L.addFirst(3);
                L.addFirst(5);
                System.out.println(L.toString());

                // compute sum of items in L
                Integer total = 0;  // a variable to hold the running sum
```

```
                for (Integer num : L) {
                        total = total + num;
                }
                System.out.println(total);
        }
}
```

Things to notice about the code to set up the list:

- We need to import the `LinkedList` library in order to use lists in Java. (For those with prior Java experience, we are intentionally using `LinkedList` instead of `ArrayList` for now – we will talk about the difference in about a week.)
- When we use the `LinkedList` class, we have to specify the type of the list contents within the angle brackets. Here, we write `LinkedList<Integer>` to indicate a list containing integers. This is a situation when you have to use `Integer` instead of `int`: the type name for list elements has to be the name of a class, abstract class, or interface.
- As in our list implementation, `addFirst` is the method that adds an item to the front of a list.
- Unlike in our list implementation, however, `addFirst` actually modifies the contents of the list.

In Java, we don't process built-in lists recursively: Java lists don't have a "rest of list" operation that we would use in a recursive computation. Instead, Java uses a construct called a `for` loop. The code sample above shows how to use a for-loop to compute the sum of a list of numbers.

Things to notice:

- We start by creating a variable to hold the result of the function (in this case, `total`). We initially set the variable to the result we would get on the empty list (the same value we'd return in the empty case of the corresponding recursive function).
- the for-loop construct walks down the list, calling each number in the list `num` in turn (`num` plays a similar role to the `first` of the list in a recursive version). As Java encounters each num, it adds it to the `total`. Java goes through the lines in the body (inside) of the `for` once for every number in the list `L`. The list items are taken in order from the front to the end of the list.
- After the for-loop is done, Java returns the `total` as the result of the function.
- The computation in the non-empty of a recursive version is the same one that we use to update the `total` inside the for loop.

You'll get more practice with loops in this week's lab.

2 Functional vs Java/Python Lists

As we go through CS18, we'll get more practical experience with when to use mutable lists and when not to. For now, we want to think about what these two styles of lists look like in terms of implementation. We've already written our own functional (non-mutating) list classes. What would a version look like that did mutate the list?

To start thinking about that question, we're going to work through a concrete example of the same core statements written against each of the non-mutating and mutating lists. We'll draw each out in memory this lecture, then use that as a basis for writing the code in the next lecture.

```
public class ListsSideBySide {
    public static void main(String[] args) {
        // the functional-style (non-mutating) lists
        IList LF1 = new EmptyList();
        LF1 = LF1.addFirst(4);
        IList LF2 = LF1;
        LF1.addFirst(7);
        System.out.println("LF1 is " + LF1.toString());
        System.out.println("LF2 is " + LF2.toString());

        // the Java/Python-style (mutating) lists
        LinkedList<Integer> LJ1 = new LinkedList<Integer>();
        LJ1.addFirst(4); // addFirst does not return anything
        LinkedList<Integer> LJ2 = LJ1;
        LJ1.addFirst(7);
        System.out.println("LJ1 is " + LJ1.toString());
        System.out.println("LJ2 is " + LJ2.toString());
    }
}
```

If we run this code, we find that the two functional lists both contain just the number 4 (neither sees the 7 that was added before we printed out the list contents). In contrast, both Java lists contain the 7 followed by the 4.

2.1 Lists in memory

Take a moment and draw out the environment and heap contents for the first segment of code (the functional version, for lists `LF1` and `LF2`. To check your work, look at slide 2 at the following URL:

`https://brown-cs18-master.github.io/content/lectures/07listsimperative/memory-layout-lists.pptx`

Now, we want to turn to understanding what memory might look like for the Java version. We can't get the memory contents exactly right without knowing how the `LinkedList` class is written, but we can infer a lot that will help us implement our own version of Linked List next lecture. Let's take it line by line. The drawings are in the remaining slides in the same slide deck URL.

After the first three lines (through the definition of `LJ2`), we have the contents as shown on slide 3: there is an object of the `LinkedList` class in memory that both `LJ1` and `LJ2` refer to. The `addFirst` suggests that there must also be an actual list somewhere in memory, and somehow, the `LinkedList` object must refer to that list. We don't know what that field might be called, but something along these lines would have to be there.

On slide 4, we start the `addFirst(7)` computation. This will result in creating a new `NodeList` object that stores the 7 and refers to the rest of the list that is under the hood.

On slide 5, we show how the `LinkedList` object could be updated to refer to the new list: the reference/arrow from that object to the actual list changes to refer to the new `NodeList` object (with 7) instead of the previous one (with 4). If we were to image following the arrows to print the content of either `LJ1` or `LJ2`, we would get each of 7 and 4, as we see when we run this code in Java.

Slide 6 puts the functional lists back in the environment, so you can see the contrast side by side. Hopefully, the contrast also helps you see how a change made to the list through either `LJ1` or `LJ2` would be visible in both, unlike changes made between `LF1` and `LF2`.

2.2 So How Do We Implement LinkedLists for Ourselves?

Think about how we might adapt our existing code for lists to support the additional `LinkedList` object – that's where we'll start off on Friday.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.