Lecture 7: Lists, Version 2 (with Mutation)

Contents

1 The Impact of addFirst on Lists

2 The Challenge of Sharing Data Structures

2.1 Functional Lists in Memory

3 Enabling Sharing

4 Introducing Java’s LinkedList class

5 Traversing LinkedLists

6 Lots of Loose Ends to Tie Up ...

Objectives

By the end of this lecture, you will know:

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

1 The Impact of addFirst on Lists

Consider the following sequence of let-like expressions that use our previous definition of lists. What do you expect this code fragment to print?

```java
LinkedList L1 =
(new EmptyList()).addFirst(9).addFirst(3);
LinkedList L2 = L1.addFirst(7);
LinkedList L3 = L1.addFirst(5);
System.out.println(L1.first);
```

There are two plausible answers: 3 and 5. If `addFirst` returns a new list without modifying the existing one (the behavior in Racket, OCaml and Pyret), you’d get 3. If `addFirst` modifies the list object when adding an element, you’d get 5. Each behavior is desirable in certain situations. So let’s look at a second version of lists, this time one that would return 5 in the above scenario.
Specifically, we are looking to support the following ListTest class, with 5 being printed out when the main method is run:

```java
public class ListTest {
    static LinkList L1 = new LinkList();

    public static void main(String[] args) {
        L1.addFirst(9);
        L1.addFirst(6);
        L1.addFirst(7);
        System.out.println(L1.getFirst()); // want to see 7
    }
}
```

LinkTest here is the name of the class we will implement (named this way to avoid the build in LinkedList name in Java).

2 The Challenge of Sharing Data Structures

2.1 Functional Lists in Memory

Let’s look back at our implementation of lists from last week (the functional version). What would memory look like if we ran the following code?

```java
LinkedList L1 = (new EmptyList()).addFirst(9);
LinkedList L2 = L1.addFirst(6);
```

The following slidedeck on memory contents shows what this looks like in memory (we’ll use the same deck for all memory diagrams in these notes).

[http://cs.brown.edu/courses/csci0180/content/lectures/07/notional-machine-updated-list.pptx](http://cs.brown.edu/courses/csci0180/content/lectures/07/notional-machine-updated-list.pptx)

The slidedeck also shows what happens if we try to update L2 and hang onto the changes. While we can mimic that by saving the return value from addFirst, that idea breaks down if we are trying to maintain access to the list from two different names (which we might do if two people shared a todo list, or a calendar, or a gradebook, etc), as in the following code.

```java
LinkedList L1 = (new EmptyList()).addFirst(9);
LinkedList L2 = L1.addFirst(6);
L2forKathi = L2
L2 = L2.addFirst(7);
```

So if we want to share lists and have changes visible in two places (either through two names, two parts of the code, etc), we need a different approach.

3 Enabling Sharing

If we want to enable sharing, we need one object that represents a list. The contents within that object should change as we add or remove items. This way, two parts of the code could reference
the same list object, and both would see the changes.

Let’s sketch out the idea here using two classes: one for lists and one for Nodes within lists. We’re leaving off constructors and methods for now, so we can focus on objects and their connections.

```java
public class Node {
    int item;
    Node next;
}

public class LinkList implements IList {
    Node start;
    int eltCount;
}
```

With these, we want to run the following code:

```java
LinkList L = new LinkList();
LforKathi = L;
L.addFirst(6);
L.addFirst(7);
```

The goal is to have both L and LforKathi able to see the updates made through the name L. Roughly, addFirst will create Node objects for new list elements, and update the references (connections) within memory to maintain the order of the items.

Go back to the slide deck to see how we want this to behave in memory.

4 Introducing Java’s LinkedList class

Java has a built-in class for what we have called LinkList. The Java version is called LinkedList. Here’s how to set up our example using the LinkedList class:

```java
import java.util.LinkedList;

public class JListExample {
    public static void main(String[] args) {
        LinkedList<Integer> L = new LinkedList<Integer>();
        L.addFirst(6);
        L.addFirst(7);
    }
}
```

Things to note here:

- At the top of the file, we have to import the LinkedList data structure (similar to what you do for tester).
• When you create a LinkedList in Java, you have to tell Java the type of the items within the list (our LinkList version assumed that the contents would be integers). You do that using the <Integer> annotation on the LinkedList. The angle brackets mean “I’m providing a type”.

We use Integer rather than int because the type provided must be the name of a class. In Java, Integer is the class version of int (similarly, there are Double and double).

5 Traversing LinkedLists

In the interest of preparing everyone for lab this week, we digress here to show you how to write a program that traverses a list. Note that our LinkList class, like the built-in LinkedList class, has no notion of the “rest” of the list. That means you can’t write simple recursive functions like you are used to from the fall.

Instead, Java (and many other languages) provide a construct known as a for loop. This construct traverses (walks over) the list for you, giving you access to the item in each node one at a time so you can compute a running value with it. Here’s an example showing how to sum up a list of numbers:

```java
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);

        // 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);
    }
}
```

In the for loop, you indicate the type of the list elements (Integer), the name to use for each element in turn (num), and the lists to take the elements from (L).

Inside the loop, you perform some computation with the individual elements. Here, we create a variable named total to hold the running sum. Initially, the total is 0 (the result we would get from the empty list). As we get each num in turn, we add it to the total.

How does this compare to how you would have written this recursively?

• We’ve created a variable to hold the result as we build it up
• The value that you would have returned in the base case of the recursive function becomes the initial value of the variable

• The computation you would have done on the rest of the list becomes the computation you do with the running variable inside the loop

• The final value is in the variable when the loop is finished

6 Lots of Loose Ends to Tie Up ...

This lecture has tried to build your intuition for how lists have to be laid out in order to have a single object for the list whose contents update on operations like addFirst. It has also shown you how such lists get processed (via for loops) for functions that you would have written recursively in the fall. To bring this full circle, we still have several questions to answer:

• What do empty lists look like in the new version?

• What goes in the next field of the last item in the list?

• How is the addFirst method written within the LinkList class?

• How would we implement the for loop ourselves in our LinkList class?

With intuition in hand, we’ll come back to these next lecture.

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: http://cs.brown.edu/courses/cs018/feedback