Lecture 7: Implementing Lists, Version 2

Contents

1 The Impact of addFirst on Lists
2 Mutating List Contents
   2.1 The Basic List Classes
   2.2 List Invariants
   2.3 Implementing addFirst
   2.4 Getting the First Element
3 Aside 1: Node as an Inner Class
4 Aside 2: Allowing Any Type of Elements
   4.0.1 Integer vs int

Objectives

By the end of this lecture, you will know:

- How functional and mutating implementations of LinkedLists differ from one another

By the end of this lecture, you will be able to:

- Implement LinkedLists whose contents change on add and remove operations
- Create inner classes
- Define a class that is parameterized by a field type
- Write a while loop to traverse a data structure in Java

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.head());
```
There are two plausible answers: 3 and 5. If `addFirst` returns a new list without modifying the existing one (the behavior in Racket and OCaml), 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 implementation 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 LinkedList L1 = new LinkedList();

    public static void main(String[] args) {
        L1.addFirst(3);
        L1.addFirst(5);
        L1.addLast(8);
        System.out.println(L1.getFirst()); // want to see 5
    }
}
```

2 Mutating List Contents

2.1 The Basic List Classes

As before, the heart of our list implementation is a `Node` class that stores an item and a link to the next item in the list. In keeping with typical idioms of programming with mutation (called imperative programming, in contrast to functional programming), we’ll name the `rest` field `next`.

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

    Node(int item, Node next) {
        this.item = item;
        this.next = next;
    }

    /**
     * return the item in a node
     */
    public int getItem() {
        return this.item;
    }
}
```

In our first implementation, we represented a list by the node that holds the first element in the list. Since the contents of the list never changed, that representation worked fine. But here, as we add elements, the first item will keep changing. We want a programmer to have a fixed object that represents “the list”, even as the elements change. So we will need a separate class for the list, which has a field that refers to the current first item on the list. It will also be handy for this class
to have a field that refers to the end of the list (so we can optionally add items to the end, rather than the beginning, of a list).

```java
public class LinkedList implements IList {
    Node start;
    Node end;

    public LinkedList() {
        this.start = ???;
        this.end = ???;
    }
}
```

When we create a new list, what should the constructor do with the `start` and `end` fields? They should get a value that signals that there are no contents yet, akin to our `EmptyList` objects from the first implementation. A pure OO implementation would set these to an object from a separate class such as `NoNode`.

In practice, this isn’t what Java programmers do. Instead, they use a Java construct called `null`, which, like `void`, is a way of saying “no information” (`null` can be used in place of an object in field values without triggering compiler errors). The use of `null` here is a bit off, because we don’t have “no information” – we actually know that there is no list contents here. However, using `null` is idiomatic Java practice so we show it to you here. But don’t abuse it. You should really only use `null` when you really have no information about the object in question.

```java
public class LinkedList implements IList {
    Node start;
    Node end;

    public LinkedList() {
        this.start = null;
        this.end = null;
    }
}
```

### 2.2 List Invariants

Before we start to implement the methods, let’s step back and think a bit about the `start` and `end` fields. For our lists to behave properly, our methods have to maintain certain properties about these fields. The technical term for such properties is *invariants*. An invariant is a boolean predicate that should be true about our data before and after each method call.

In our implementation of linked lists, we will maintain the following invariants:

1. When the linked list is empty,
   - `start` is null
   - `end` is null

2. When the linked list is nonempty,
• start refers to the first node of the list
• end refers to the last node of the list
• if the list contains exactly one node, then both start’s and end’s next fields are null
• if the list contains more than one node, then only end.next is null; no other node’s next field is null.

Are the invariants true about LinkedList objects after the constructor has finished? Yes – the list is empty, and both start and end are null.

With this, let’s go on to write addFirst.

2.3 Implementing addFirst

Here’s a first attempt at writing addFirst:

```java
public void addFirst(int newItem) {
    Node newNode = new Node(newItem, this.start);
    this.start = newNode;
}
```

Are you satisfied with this definition? A good strategy here is to check it against the invariants. Assume the invariants were true before the call to addFirst – are they still true afterwards?

Not quite – we have start referring to the first item in the list, but if the list had been empty before calling addFirst, then end still refers to null, rather than the new Node. This suggests the following modification:

```java
public void addFirst(int newItem) {
    Node newNode = new Node(newItem, this.start);
    this.start = newNode;
    if (this.end == null) {
        this.end = newNode;
    }
}
```

You should work through all the invariants and convince yourself that this method preserves them.

The process of writing addLast is similar, though it raises a couple of interesting tidbits:

• A helper method that determines whether the list is empty makes the code a bit cleaner

• When adding a element to the end of a list, the next field of the Node for the new element will remain null. It’s useful to have a second constructor in the Node class that only takes the element, and internally sets the next field to null (the source code posted with this lecture shows both constructors in the Node class).

```java
boolean isEmpty() {
    return ((this.start == null) && (this.end == null));
}
```
public void addLast(int newItem) {
    if (this.isEmpty()) {
        this.addFirst(newItem);
    } else {
        Node newNode = new Node(newItem);
        this.end.next = newNode;
        this.end = newNode;
    }
}

2.4 Getting the First Element

public int getFirst() {
    return this.start.getItem();
}

With the addition of getFirst, our desired ListTest class runs as desired.

3 Aside 1: Node as an Inner Class

Now that we have addFirst on linked lists, we never create nodes directly. Only the LinkedList class creates nodes. This suggests that Node is really more of a helper class, than a class in its own right. How can we capture the idea that only the LinkedList class should be allowed to create nodes? (Note that this is a very different situation from making classes abstract – here we need to be able to create nodes, we just want to limit which class can create them).

The solution is to define the Node class inside the LinkedList class. This is sort of like defining one function inside of another (like a local lambda) in Racket. Such inner classes aren’t visible outside their enclosing class, so we get the restriction we want:

public class LinkedList implements IList {
    // the LinkedList fields
    Node start;
    Node end;

    // the inner class
    class Node {
        int item;
        Node next;
        ...
    }

    // the LinkedList methods
    public void addFirst(int newItem) {
        ...
    }
}

The posted source code shows this version of the class layout.
4 Aside 2: Allowing Any Type of Elements

As of now, we can only create lists of ints, which is rather limiting. We would like to be able to create lists of strings, students, animals, and so on. Nearly all of the code would be the same as what we have now, aside from the type of item in the Node class. Surely there is a way to be flexible on the types without having to copy all of this code for every new type?!

Indeed there is. Java actually allows classes to take separate “type parameters”. The notation <T> after a class name introduces a type parameter named T (you can use any name you want—it doesn’t have to be T). You can then use T in place of any type that should come from the parameter. Here’s the skeleton of the LinkedList class with the type parameter.

```java
public class LinkedList<T> implements IList<T> {
    // the LinkedList fields
    Node start;
    Node end;

    // the inner class
class Node {
        T item;
        Node next;
        ...
    }

    // the LinkedList methods
    public void addFirst(T newItem) {
        ...
    }
}
```

Notice that we use T for both the type of the item in a Node, as well as for the input type of addFirst. (You can have multiple type parameters, separated by commas, in the class annotation, should you need that – we’ll see uses of that in a couple of weeks).

How does this change how we create LinkedList objects? Here’s ListTest rewritten to use the type parameter:

```java
public class ListTest {
    static LinkedList<Integer> L1 = new LinkedList<Integer>();

    public static void main(String[] args) {
        L1.addFirst(3);
        L1.addFirst(5);
        L1.addLast(8);
        System.out.println(L1.getFirst());
    }
}
```

When we write down the type of a list variable, or when we use the constructor, we provide a concrete type to use for the item.
4.0.1 Integer vs int

Here we are using Integer, rather than the int we have been using. Basically, Integer is a class that contains an int, whereas int is just a numeric constant. Anytime you are in a context that needs integers as objects, use the type Integer instead. The same distinction holds for Boolean vs boolean and String vs string.

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](http://cs.brown.edu/courses/cs018/feedback)