Lecture 8: Implementing LinkLists

Contents

Motivating Question

How can we build Java-style LinkedLists whose contents can be mutated?

Objectives

By the end of this lecture, you will know:

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

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

- add elements to the front and end of a mutable list

1 Implementing Mutable Lists

Last class, we worked out the memory diagrams showing the objects that would be needed to implement a mutable linked list. This class, we turned that memory layout into actual code.

1.1 Adding a LinkList class

We started by adding a class `LinkList` (as opposed to `LinkedList` as shown in the memory diagram. We changed the name to avoid clashes with Java's built-in classes. Here's the initial class, with a field `start` to refer to the start of the list.

```
public class LinkList implements IList {
    ______ start;  // the actual list underneath
    LinkList(){}
}
```

What type might we insert into the blank space as the type for `start`? Here are several possibilities:

- `int`
- `Object`
- `IList`
- `AbsList`
- `EmptyList`
- `NodeList`

The `start` of the list should be an actual list, not the contents, which rules out `int`. This means it has to be a type that allows both `EmptyList` and `NodeList`. Each of `Object`, `AbsList`, and `IList` would do that. That said, `IList` is the best choice. `Object` is too general, and we should generally use interfaces as types when they are available (we'll see why that's a particularly good decision in this case in the next lecture).

Our class now looks like:

```
public class LinkList {
    IList start;  // the actual list underneath
    LinkList(){}
}
```

Is there any work for the constructor to do? Well, when we make a new list, it should start out as the empty list. There are two ways we could set that up: as part of the field or as part of the constructor:

```java
public class LinkList {
    IList start;  // the actual list underneath
    LinkList(){
      this.start = new EmptyList();
    }
}
```

```java
public class LinkList {
    IList start = new EmptyList();  // the actual list underneath
    LinkList(){}
}
```

Both approaches work, though the latter is more conventional since it doesn't depend on any input when the list is created.

1.2 Implementing AddFirst

Our goal of implementing `LinkList` is to eventually implement the `IList` interface. That interface name is already in use, however, as the common type name for the `EmptyList` and `NodeList` classes. Conflating all of these under `IList` is tempting (and indeed, we took that route in lecture), but it will create problems for compiling our code, since we are implementing the `LinkList` methods gradually. It also isn't conceptually correct: we don't want the `rest` field of a `NodeList` to refer to a `LinkList`. We therefore want two different interface names.

For consistency with what's coming in the next few lectures, we will reserve `IList` for the lists that we expect programmers to use. We will rename the current `IList` interface to `IListInternal`, and have `EmptyList` and `NodeList` implement that instead. Here are the current interfaces:

```java
public interface IListInternal {
    public boolean isEmpty();
    public IListInternal addFirst(int elt);
    public IListInternal addLast(int elt);
    public IListInternal remEltOnce(int elt);
    public int length();
    public int head();
}

public abstract class AbsList implements IListInternal { ... }

public interface IList {
    public IList addFirst(int elt);
}

public class LinkList implements IList {
    IListInternal start = new EmptyList();  // the actual list underneath
    LinkList(){}
}
```

Stop and Think: Is it a problem that we have two different methods called `addFirst` with different output types in our different classes?

Now, let's write the `addFirst` method in `LinkList`. Here are some options:

```
public IList addFirst(int newelt) {
   return this.start.addFirst(newelt)
end

public IList addFirst(int newelt) {
   this.start = new NodeList(newelt, new EmptyList());
   return this;
end

public IList addFirst(int newelt) {
  this.start = new NodeList(newelt, this.start);
  return this.start;
end
```

Stop and Think: what do you like or dislike about each of these options?

Actually, none of these is quite right. We need to return a `LinkList`, which neither the first nor third do. The second achieves that, but always puts the new element on a new empty list, rather than the existing list. The right solution is:

```
  public IList addFirst(int newelt) {
  this.start = new NodeList(newelt, this.start);
  return this;
end
```

Why make the new `NodeList` here, rather than use the `addFirst` method on the existing classes? We certainly could have done that, but this version is more conventional, as it makes the structure of the computation a bit clearer. (We'll see another reason in the next lecture as we continue to work on this code.)

2 Adding to the End of the List

Java's built-in `LinkedList` class also provides a method for adding an element to the end of the list. This is pretty handy in practice. What would that look like? Here's a proposal, showing code for each of the `LinkList`, `NodeList`, and `EmptyList` classes:

```
public class EmptyList extends AbsList {

  public IListInternal addLast(int newelt) {
    return new NodeList(newelt, this);
  }
}

public class NodeList extends AbsList {
  int first;
  IListInternal rest;
```

```
  public IListInternal addLast(int newelt) {
    this.rest = this.rest.addLast(newelt);
    return this;
  }
}

public class LinkList {
  public IList addLast(int newelt) {
        this.start = this.start.addLast(newelt);
        return this;
   }
}
```

This is a straight-up recursive solution, similar in style to what you would have implemented on the functional lists. We can't use `for` loops to do this because they only work on Java's built-in classes (we'll show how to make them work on classes we write a bit later in the course).

2.1 A Timing Check

Consider the following two expressions. What is the running time of each (assuming `someList` is a `LinkList`)?

```
1: someList.addFirst(4).addFirst(6)
2: someList.addLast(6).addLast(4)
```

`addFirst` takes constant time: no matter how large the list gets, the time to create a new `NodeList` and connect it to the `start` field is the same. In contrast, `addLast` takes linear time because we have to walk all the way to the end of the list to insert the element. That's unfortunate – we really want the two operations to take the same amount of time if possible. So what might we do?

3 Speeding up addLast

Stop and Think: why do we *need* linear time to add to the end of the list?

The only computation we are doing in `addLast` is getting to the end of the list. We're not doing any computation along the way. That suggests that storing a bit of additional information about where the end of the list is could speed up `addLast`. In fact, that is what we will do next class. We will modify `LinkList` to look like the following:

```
public class LinkList {
  IList start;  // the start of the actual list underneath
  IList end;   // the end of the actual list underneath

  LinkList(){}
}
```

Once we figure out how to work with the `end` field, we'll get a fast implementation of `addLast`. And it will raise some other interesting issues along the way ...

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`.