Lecture 8: Implementing LinkLists

Contents

1. Mutable Lists in Memory ........................................ 1

2. Classes for Mutable Lists .................................... 2

3. Filling In the LinkList class .................................. 2
   3.1 Representing the Empty List – the null value .......... 3
   3.2 size and addFirst ............................................. 3
   3.3 contains and while Loops ................................... 4

4. Implementing addLast ......................................... 5
   4.1 List Invariants .................................................. 6
   4.2 Fixing addFirst ................................................ 7
   4.3 Checkpoint ....................................................... 7

5. Aside 1: Node as an Inner Class .............................. 8

6. Aside 2: Allowing Any Type of Elements ................... 8
   6.0.1 Integer vs int ............................................... 9

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

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 Mutable Lists in Memory

Last class, we motivated the idea of wanting to have a list whose contents were changed as we add elements. We saw this in practice with Java’s built-in LinkedList class. Today we want to implement this. First, let’s work at the level of heap objects to make sure we’re clear on our goals.

Imagine that we had a list $L_1$ containing the numbers 6 and 9, to which we wanted to add 7 onto the front. Using our original (functional/non-mutating) NodeList implementation, we would get the following heap picture:

If we wanted the name $L_1$ to refer to the new List that includes 7, we had to manually associate the name $L_1$ with the new list. Our goal for today is to have the original code work without us having to modify the association by hand. Specifically, *the object that $L_1$ refers to should stay the same, while the 7 should now be the front of the list.* This suggests that we need a different heap picture in which $L_1$ doesn’t refer to the first node, but instead to another object that in turn refers to the (current) first node. Here’s the new picture:

Our goal today is to implement that new picture. (We will actually implement it *without* the EmptyList object, for reasons that will become clear as we go.)

2 Classes for Mutable Lists

As the previous picture suggests, we need two classes here – one for the Node and one for the overall list. We’ll call that second class LinkList to avoid confusion with the built-in LinkedList.

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

public class LinkList implements IList {
    Node start;
    int eltCount;
}
```
Note that the `next` field within the `Node` class no longer has type `IList`. The `IList` interface is now on the `LinkList` class, since objects of that class will be used as list values (and thus need to support the list methods).

For now, we will work with a subset of the `IList` interface from last week:

```java
public interface IList {
    public int size();
    public void addFirst(int elt);
    public void addLast(int elt);
    public boolean contains(int elt);
}
```

### 3 Filling In the LinkList class

Let’s start with the constructors:

Here’s where we left off our `LinkList` implementation after the last lecture:

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

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

public class LinkList {
    Node start;
    Node end;

    public LinkList() {
        this.start = _________;
        this_eltCount = ______;
    }
}
```

#### 3.1 Representing the Empty List – the null value

First, we need to finish the constructor which creates an empty list. If we were following pure OO design, we would make a class for nodes without content (such as `NoNode`) and use that as the start value for an empty list. In practice, this ends up creating an unnecessary object (we’ll see why it is unnecessary when we implement `contains`), so most OO programmers instead use something called null.

null is a special value in Java (it or a similar value exists in many other languages as well). It can be used anywhere an object (from any class) is expected, and it signifies that there is no actual
object at the moment. It is most commonly used when creating data structures, so you can set up fields and variables until methods get called to populate fields or set variables. Here, we will use null for the initial value of start. We will also use it to indicate when there is no value for the next field (i.e., to signal the end of a list).

We could have just as well set the initial values of start and eltCount when we declared the fields, since neither depends on inputs to the constructor. This version would look like:

```java
public class LinkList implements IList {
    Node start = null;
    int eltCount = 0;

    public LinkList() {}
}
```

Either approach is acceptable, both in 18 and in practice (though the initialization version is more common in practice).

### 3.2 size and addFirst

For size, we will maintain the element count as a variable as we did in the functional version last week. Here’s an extended version of the classes with size and a proposed addFirst method, as well as the null values filled in:

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

    public LinkList() {
        this.start = null;
        this.eltCount = 0;
    }

    public int size() {
        return this.eltCount;
    }

    // here’s a proposed addFirst method
    public void addFirst(int elt) {
        Node newN = new Node(elt, this.start);
        this.start = newN;
    }
}
```

The addFirst code follows the heap modifications needed in the diagram we showed earlier. The first line creates the new Node and its reference to the node for 6. The second line sets up the reference from start to the new node.
3.3 *contains* and *while* Loops

Now we want to write a method to determine whether an item is in a list. Last semester, you would have written this recursively. That’s a little more challenging here because there is no “rest” field or method in the `LinkList` class. Making one would require us to create a new `LinkList` object to wrap around the rest of the nodes after the start node. This seems clumsy and wasteful. So how does one do this instead?

We could consider making a recursive method in the `Node` class, but we will see on Monday why this isn’t really a practical solution. Instead, we will use a different kind of loop in Java, called a *while* loop.

The idea of a *while* loop is that you have some computation that you want to keep repeating until some condition is met. Here’s the `contains` method written with a *while* loop:

```java
class LinkList implements IList {
    /*
     * determine whether list contains given elt
     * @param elt : the item to look for
     */
    public boolean contains(int elt) {
        Node current = this.start;
        while (current != null) {
            // what to do in the non-emptylist case
            if (current.item == elt) {
                return true;
            } else {
                current = current.next;
            }
        }
        return false; // what to do in the emptylist case
    }
}
```

In this code, we’ve set up a variable named `current` to track the `Node` that we’re currently checking in the list. We initialize `current` to `this.start`. Then, as long as we haven’t reached the end of the nodes (as indicated by `null`), we check the given element against the current node’s item, returning if we have found it and advancing current to the next item if we have not. This has the same behavior as the recursive version, we’re just updating the value of `current` rather than using getting the rest of the list.

**So why no EmptyList object?** The *while* loop also shows why a separate class for the end of the list doesn’t really make sense here. The *while* loop needs a condition for terminating the iteration. Thus, we need to ask some question to say “are we done yet” (`current != null` is that question). Having asked the question, we can insert the answer for that case after the *while*.

When we did the `NodeList` version, in contrast, we simply called the method on the `rest` object and *let Java determine which version of the method to run*. One of the major principles in object-oriented programming is to let the language handle type-based questions for you. Asking whether a list is empty or not is a type question, so principled OO programming wouldn’t ask that. However, there
are a handful of limited cases such as writing an `equals` method or using a `while` loop for iteration where the type question has to be in the code. In the case of `LinkList`, we never end up calling a method on the `EmptyList`, so there is no point having a separate class for it.

4 Implementing `addLast`

The `addLast` method, like `addFirst`, inserts a new node into the list, just in the last position. This seems like another situation in which to use the `while` loop – we iterate to the end of the list, then put a new node in place.

*what is the running time of this approach?*

This approach would be linear in the size of the list, in contrast to a constant-time `addFirst`. This raises a question – can we also do `addLast` in constant time? This sounds like a question we asked last week in the context of `size`. In that case, we reduced a traversal to a constant lookup by adding a field to store information about the list. Could we do that here too?

Yes. Just as `LinkList` has a `start` field that references the first item in the list, we could add an `end` field that references the last item. As long as we maintain that `end` field properly, we could use it to implement `addLast` in constant time.

Here’s a proposal of what that might look like:

```java
public class LinkList implements IList {
    Node start;
    Node end: // <---- new field, also initialized to null
    int eltCount;

    public void addLast(int elt) {
        Node newN = new Node(elt, null);
        this.end.next = newN;
        this.end = newN;
    }
}
```

*Are you satisfied with this implementation? Try some test cases (by hand) …*

There are two things we need to think about here. One is that we designed this code with a non-empty list in mind, so we should check that it also makes sense for an empty list. But once we are thinking about an empty list, we have to ask whether the `start` field needs to change as well if we add a new last element. Specifically:

*The start and end fields are not independent of one another when the list is empty. When fields aren’t independent, we have to think about what relationship they should have at all times.*

4.1 List Invariants

Invariants are statements of how a data structure is configured, including how it uses special values like `null`. Every method should preserve the invariants after it runs, which in turn enables every method to assume the invariants when it starts. This reduces the need for certain kinds of error checking.
In our implementation of linked lists, we will maintain the following invariants:

1. When the linked list is empty,
   - \texttt{start} is null
   - \texttt{end} is null

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

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

With this, let’s revisit \texttt{addFirst}.

### 4.2 Fixing \texttt{addFirst}

Assume the invariants were true before the call to \texttt{addFirst} – are they still true afterwards?

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

```java
public void addFirst(int elt) {
    Node newNode = new Node(elt, 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 \texttt{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 \texttt{next} field of the Node for the new element will remain \texttt{null}. It’s useful to have a second constructor in the \texttt{Node} class that only takes the element, and internally sets the \texttt{next} field to \texttt{null} (the source code posted with this lecture shows both constructors in the \texttt{Node} class).
boolean isEmpty() {
    return ((this.start == null) && (this.end == null));
}

public void addLast(int elt) {
    if (this.isEmpty()) {
        this.addFirst(elt);
    } else {
        Node newNode = new Node(elt, null);
        this.end.next = newNode;
        this.end = newNode;
    }
}

4.3 Checkpoint

At this point, we have finished having LinkList implement the IList interface. We’ve seen the use of null and while loops, and the role of invariants in helping us check for errors in our code. And we’ve seen how to have lists that can be shared across two parts of your code. These are all valuable building blocks going forward.

The rest of these notes show two additional modifications that one would usually make to an implementation of linked lists: allowing any type of data in the list, and burying the Node class so that only the LinkList class can use it. We’ll do the second one first.

5 Aside 1: Node as an Inner Class

Now that we have addFirst on linked lists, we never create nodes directly. Only the LinkList 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 LinkList 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 LinkList 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 LinkList implements IList {
    Node start;
    Node end;
    int eltCount;

    // the inner class
    class Node {
        int item;
        Node next;
        ...
    }
}
The posted source code shows this version of the class layout.

6 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 LinkList class with the type parameter.

```java
public class LinkList<T> implements IList<T> {
    Node start;
    Node end;
    int eltCount;

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

    // the IList methods
    public void addFirst(T elt) {
        ...
    }

    public boolean contains(int elt) {
        Node current = this.start;
        while (current != null) {
            if (current.item.equals(elt)) { // note change here!
                return true;
            } else {
                current = current.next;
            }
        }
        return false;
    }
}
```
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).

We also had to change the contains method to compare objects with equals rather than ==, since the latter otherwise only compare memory locations.

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

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

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

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.

### 6.0.1 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 Double vs double.

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