Lecture 9: Optimizing List Operations

Contents

1 Speeding up `addLast` **1**

1.1 Key Takeaways 3

2 A Side Note on Testing and Debugging **3**

3 Speeding up `length` **4**

3.1 Key Takeaways 5

4 Can we speed up `remEltOnce?` **5**

5 Loops versus Recursion in List Methods **5**

6 Recap **6**

Motivating Question

How can we reduce the time required by key list operations?

Objectives

By the end of this lecture, you will know:

- How to add fields to speed up certain list operations
- Which kinds of operations can be optimized with fields
- Pitfalls to watch out for as you optimize through fields

1 Speeding up `addLast`

Last lecture, we observed that `addLast` required linear time because we had to get to the end of the list in order to insert the new element. We also asked whether we could speed up that process by adding an `end` field to the `LinkList` class that would always refer to the last list node in memory.

As a reminder, here is the `LinkList` class from last time, with the `start` field that we use to implement (the constant time) `addFirst` operation:

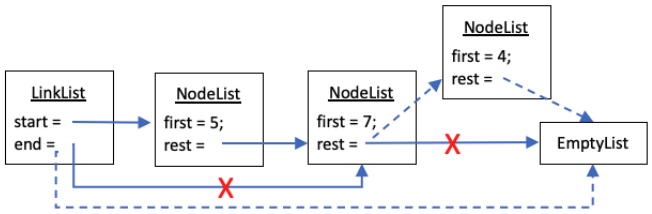
```
public class LinkList implements IList {
  IListInternal start = new EmptyList();
```

```
  IListInternal end = this.start;

  LinkList(){}

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

Conceptually, the following diagram shows the idea of how `addLast` should work. The solid arrows exist before `addLast` runs. Afterwards, the dashed arrows should have been added and the arrows with Xs over them should have been removed.



Corresponding, a first attempt at `addLast` might look like:

```
public IList addLast(int newelt) {
  NodeList newN = new NodeList(newelt, this.end.rest);
  this.end.rest = newN;
  this.end = newN;
  return this;
}
```

As an exercise, match up the lines of code to the arrows that each one creates and/or deletes in the diagram.

If we enter this code into Java, we get an error that `rest` is not defined for `this.end`. Like `start`, `end` will need to be an `IListInternal`, allowing for an `EmptyList`. The interface doesn't have a `rest` field (`EmptyList` by definition has no `rest`). Our diagram assumed that the list wasn't empty. Our code therefore needs to handle that case separately.

You Try It: Draw out the diagram for the empty case before proceeding. Then try to correct the code.

Here's the next version of the code:

```
public LinkList addLast(int newelt) {
  if (this.isEmpty()) { // end and start both refer to EmptyList
    this.end = new NodeList(newelt, this.end);
    this.start = this.end;
  } else { // end must already refer to a NodeList
    NodeList newN = new NodeList(newelt, this.end.rest);
    this.end.rest = newN;
    this.end = newN;
```

```
    }
    return this;
}
```

Did you catch having to update the `start` reference when the list is empty? Drawing out the diagram up front was intended to help you spot that issue.

Unfortunately, Java still isn't happy: it is still complaining that `rest` might not be available on `this.end`. But we've checked that with the **`if`** statement – we know that `this.end` is a `NodeList`, so what's Java's problem???

The checker that reviews your code before running in Java is limited in what it can check. It doesn't look at the content of your code. It looks just at whether your uses of names are consistent with their types. Your code says `end` is an `IListInternal`, so Java complains.

In this case, we need to tell the Java compiler/checker that we, the programmers, know that `end` will be a `NodeList` within the **`else`** case. We do this by writing the following:

```
((NodeList)this.end).rest
```

In parentheses before `this.end`, we put the type that we know the name will have if running the code gets to this point. When we include this annotation, Java will check our code assuming the indicated type, but if the actual `end` object is not a `NodeList` when this line runs, Java will halt our program with a runtime exception. This is called *casting* in Java. You can't use it to change the class of an object. You are only using it to tell the compiler that you know something more about the actual object in that situation.

Once we add the casting, `addLast` will be complete. You can see the final result in the posted code file (the one labeled *lec09*).

1.1 Key Takeaways

The takeaway from this segment is

- if you are traversing a data structure to get to a fixed point (like the end of the structure), you can reduce that computation to constant time by maintaining the current value of that fixed point in a field/variable.

But that said,

- Be careful to **maintain all of your fixed variables whenever you update one of them** (as we saw with having to adjust `start` when we adjusted `end`.

We really can't stress this second point enough. Being attentive to this could save you considerable time on debugging your code later. Writing out diagrams showing the old and new data structures and writing test cases are both good ways to catch these dependencies as you go.

That second takeaway should raise a warning – `addFirst` updated `start`. Might `end` also need to be updated in that case? Think about it (then see the posted code for the answer).

2 A Side Note on Testing and Debugging

Kathi was feeling lazy when she wrote up the code for this lecture. She was tired, and rushing, and confident in what she was doing. But she knew she shouldn't post solution code without some sort of testing, so she threw the following into the `main` method:

```
LinkList list3 = new LinkList().addLast(6).addLast(8);
System.out.println(list3);
```

After running the code, Java printed out the list contents:

```
[8, 6, ]
```

(never mind the extra comma at the end – look at the order of the items!)

45 minutes later, Kathi found the problem. The **`if`** statement to check whether the list is empty refers to `this.isEmpty`, which is a method in the `NodeList`/`EmptyList` classes. And in the `NodeList` class, that method was written to return **`true`** (oops!). How did *that* happen? Because when Kathi wrote the `NodeList` class, she was tired and rushing, and copied over code from the `EmptyList` class to get started. But then she forgot to change the return value on the method after copying it.

And, since she was tired and rushing, she never bothered to test the `isEmpty` method when she wrote it (after all, who could get *that* wrong after years of teaching CS).

And thus, the three minutes it would have taken to quickly check `isEmpty` the first time around turned into 45 minutes of frustration later on. Kathi assumed the error must have been in the `addLast` method since that's the one she was just writing. It was a long wild goose chase until she started tracing the code execution (which you can do either with print statements or the debugger).

This is a true story. We know that many of you resist writing tests because you are tired and rushing and feel it is a pointless activity. All programmers experience that, Kathi and the TAs included. But when things go wrong, you appreciate how a couple of simple sanity-check tests as you code can save you a LOT of time and frustration.

3 Speeding up `length`

The `length` method is also linear time: can we improve its performance? Unlike with `addLast`, `length` isn't searching for a fixed value. But it is instead computing a value that builds up incrementally in a predictable way: every time we add an item to the list, the length increases by 1. So why not maintain a field with the current number of items, and return that instead of computing the `length` via a traversal?

This is a good idea, and it is indeed what actual list implementations will do. The posted code shows the new field (`eltCount`) and how we update it in `addFirst` and `addLast`.

The tricky part is maintaining this value when we *remove* items. Our `remEltOnce` method doesn't require that the given element already be in the list, so calling this method may or may not reduce the size of the list. We therefore have to do something extra to maintain `eltCount` on removal. There are a couple of options:

- Keep the linear-time `length` method in the `IListInternal` classes, and call it at the end of each removal to update `eltCount`.
- Have `remEltOnce` return both the updated node chain and the revised element count back to the `LinkList` class. This would return a new class to store the two return values. (Knowing how to do this is a good exercise for those who are interested.)
- Have `NodeList` maintain its own `eltCount` (or similar) field: `remEltOnce` would update the value of this field for each object as it checks whether a node should be removed. The `NodeList` objects in our current version already update their `rest` fields, so updating the local `eltCount` would fit into the current code structure nicely.

The posted code currently does the first option. The third is the best option (which we leave as an exercise for you to try). The second is heavierweight, but trying it out would help you know how to return multiple values from a method if you ever had to do that.

3.1 Key Takeaways

Methods that traverse data to build up a value incrementally can often be optimized by doing the incremental computation as the data are constructed. But think carefully about which methods might require you to update the computed value.

One good way to check for needed updates is to test combinations of methods together. In this case, test that the length of a list is as expected upon each method that might change the number of nodes. Merely testing that the list *contents* are as expected isn't sufficient: we have to test that methods that depend on the contents also weren't affected.

4 Can we speed up `remEltOnce`?

While we're looking at linear-time methods to speed up, can we do anything about `remEltOnce`? That also traverses the list.

In this case, linear time is the best we can do. Until the method is called with a specific element, we don't know which element will be needed. We wouldn't know which element to hang onto in an additional field, and we have to search the whole structure in order to find the element. Not all computations can be optimized like we did for `addLast` and `length`.

5 But Is It All Really Working?

We warned you earlier that you have to be really careful once you use variables to track key parts of your data structure (such as the `end`) of the list. Breaking dependencies among objects in memory becomes really easy if we aren't careful.

Take a look at the tests for our new `length` method in the `lec09` code bundle: they seem to run fine.

Now, change the uses of `list4` to `list1` in the `length` tests (commenting out the two lines in that section that define `list4` as a new list – just reuse the existing `list1`. What happens? Why? If

you really understand this, you'll be able to write down a couple of sentences that explain what's happening here.

Stop and Think: What do you take away from this?

6 Loops versus Recursion in List Methods

As an aside, some students have asked about whether to use loops or recursion to traverse lists when needed. As an illustration to compare the two, here are two versions of a `length` method (both in the `NodeList` class):

```
// recursion
public int length() {
  return 1 + this.rest.length();
}

// while loop
public int length2() {
  int count = 0;
  NodeList curr = this;

  while (!curr.isEmpty()) { // ! is not
    count = count + 1;
    curr = curr.rest;
  }
  return count;
}
```

In the second version, we maintain a variable `curr` (for "current") that moves down the sequence of nodes, adding `1` as each node is visited. This is the same computation that occurs in the recursive version. In the recursive version, the name `this` refers to each node in succession (recall that `this` always names the object from which the method being run was called).

Ultimately, which version you write is largely a style preference. The recursive version has the benefit of simplicity: it doesn't have to check whether you are at the end of the list because that happens naturally in the `EmptyList` object at the end. And there are no variables for the programmer to (remember to) maintain. (Forgetting to maintain variables is where a great many programming errors lie.)

That said, the `while` version is what you are more likely to see in internships and online postings. The reasons are partly historical (see the notes on `null` from today for details). Some will also argue that the `while` version is faster because it takes a few computation steps to call methods (a constant number per call). While this is true, this amount of cost will rarely matter in practice (the exception being if you in a setting where ounce of computation savings matters).

For this course, you may choose which approach to take on all assignments.

7 Recap

These notes have focused on improving the running-time performance of a linked list implementation. The strategy of adding fields to speed up later computations is common in programming.

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`.