CSJ61B Lecture 4

Wednesday, February 12, 2020

Last time in 61B, we designed an `IntList`:

```
public class IntList {
    public int first;
    public IntList rest;

    public IntList(int f, IntList r){
        first = f;
        rest = r;
    }
}
```

The above is an example of a "naked" data structure, which is functionl, but users will need to know references well, and be able to think recursively. Let's make it a little more convenient to use.

To redesign our class, let's give the variables some new names:

```
public class IntNode {
    public int item;
    public IntNode next;

    public IntNode(int i, IntNode n) {
        item = i;
        next = n;
    }
}
```

Now if you say this looks similar to `IntList`, it is. It's exactly the same, except for some different variable names. Now that this is done, we'll make another class:

```
public class SLList {
    public IntNode first;

    public SLList(int x) {
        first = new IntNode(x, null);
    }

    public static void main(String[] args) {
        SLList L = new SLList(10);
    }
}
```

What did this do? Well, we've made a new "naked" recursive list, with a method that hides the "naked"-ness:

```
IntNode X = new IntNode(10, null);
SLList Y = new SLList(10);
```

The next programmer who uses `SLList` will not need to worry about `null` and references.

Now let's make some methods.

```
public void addFirst(int x) {
    first = new IntNode(x,first);
}

public void getFirst() {
    return first.item;
}
```

Suppose we wanted to make a list of the numbers 5, 10, 15. Previously:

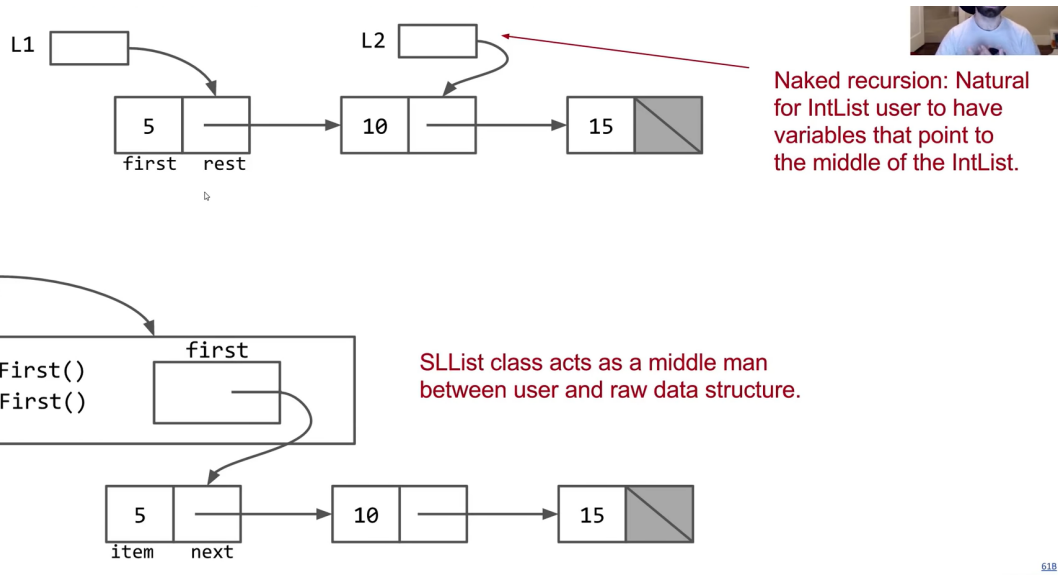
```
IntList L = new IntList(15, null);
L = new IntList(10, null);
L = new IntList(5, null);
```

And now:

```
SLList N = new SLList(15);
L.addFirst(10);
L.addFirst(5);
```

You can see this just looks a little neater, especially if you're a novice. For naked recursion, it is natural for the `IntList` user to have variables that point right to the middle of the `IntList`.

However, the `SLList` class acts as a middle man between the user and the raw data structure.



Access Control

You might notice that while we've designed this layer that acts as the middleman between the user and the raw data structure, there's nothing stopping the user from just directly accessing the data structure and changing it him/herself.

Well, we can protect our data using the `private` keyword:

```
public class SLList {
    private IntNode first;

    public SLList(int x) {
        first = new IntNode(x, null);
    }

    public static void main(String[] args) {
        SLList L = new SLList(10);
    }
}
```

The Java compiler will actually enforce the `private` keyword: any methods outside of the `SLList` class will not be able to change details.

Why do we restrict access? It hides implementation details from users of your class, as a method of abstraction, and reduces the learning curve. In the meantime, it is also safe for you to change the method of implementation yourself.

It is analogous to a car. There are some public parts the user should worry about, like the pedals and wheel, but think about the electric car. The fuel line, analogous to a `private` method, was changed to a motor, but the user does not have to worry about this and can instead just drive.

Despite the term **access control**, it has nothing to do with protecting against hackers, spies etc, it is just a way to tell other programmers you don't have to mess with it.

It is convention in Java that if you make something `public`, the expectation is that you will never remove it. Like in a car, if you removed the steering wheel, people would not be able to use your car. Similarly, removing `public` classes breaks backward compatibility with your code.

Nested classes

In Java, you can move classes inside of other classes:

```
public class SLList {
    private static class IntNode {
        public int item;
        public IntNode next;
        public IntNode(int i, IntNode n) {
            item = i; next = n;
        }
    }
    private IntNode first;
    public SLList(int x) {
        first = new IntNode(x, null)
    }
}
```

Nested classes are useful when a class doesn't stand on its own, and is obviously subordinate to another class. For example, if another class should never use the nested class (like `IntNode`), you can just set it to private, as other classes probably won't need to manipulate `IntNode`.

If the `IntNode` class never uses the the `SLList` class, you can also add the word `static` to the class declaration of `IntNode`.

Static classes result in a minor savings of memory, but other reasons will be discussed later.

> **A really minor detail**
>
> When you make a private nested class, the access modifiers for its variables (`item` and `next` for example) become irrelevant.

`addLast`

Let's give our `SLList` class some additional powers: in particular, `addLast` and `size`.

```
public void addLast(int x) {
    IntNode p = first;
    while (p.next != null) {
        p = p.next;
    }
    p.next = new IntNode(x, null);
}
```

This code runs without error, but right now it's not easy to get the last item. What's another way we can confirm it added? We could check its size!

`size()`

You'll see the `SLList` data structure itself is not inherently recursive. The common way to get around this is to first build a helper function:

```
/** Returns the size of the list starting at P.*/
private static int size(IntNode p) {
    if (p.next == null) {
        return 1;
    }
    return 1 + size(p.next);
}

public int size() {
    return size(first);
}
```

You will very frequently see this system when you're working with data structures: in Hug's words, a `private` `static` method that speaks the language of the gods, and a public method that mortals can see.

Caching

There are a couple of problems with our code as it is, one of which is that it's pretty slow. Let's do a quick question to see how slow it is:

If `size` takes 2 seconds on a list size of 1000, how long will it take for a list of size 1 million? It will take 1000 times as long!

Python's `len` function does something similar to this, but it takes the same amount of time irrespective of the size. How can we make our function much faster?

Instead of calculating size every time we ask for it, we can just save this information as we add to the list We can a new `private` variable that has info about size, and set all of our methods such as `addFirst` and `addLast` to tally every time thyey add to the list. Now we can just write:

```
public int size() {
    return size;
}
```

This behavior of maintaing a special variable is called **caching**,. However, there ain't no such thing as a free lunch: we did have to do extra work to make it possible.

The `SLList` class also acts as a convenient storage location for our `size` variable, which is much tougher to do with the naked `IntList` class.

The Empty List

So far, we've seen that `SLList` has a faster `size()` method that would have been convenient for `IntList`, and it's also simpler to use, has a more efficient `addFirst` method, and helps prevents errors.

Another benefit is that it can also be easier to represent the empty list, by setting the `front` to `null`. Let's create an alternative constructor for that.

```
public SLList() {
    first = null;
    size = 0;
}
```

There's actually a pretty tiny bug hiding in this code, and that is that if you try to add to the end of an empty list, the code will error. This isn't easily obvious, but think that `p` doesn't have a `next` variable if it is `null`.

So how do we fix `addLast`?

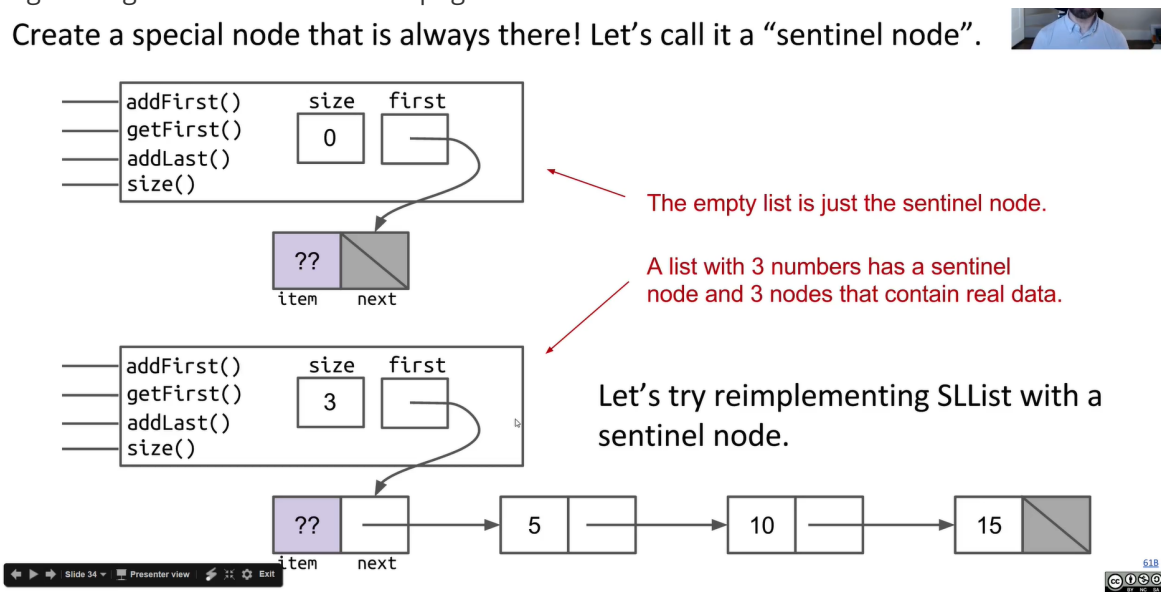
```
public void addLast(int x) {
    size += 1;

    if (first == null) {
        first = new SLList(x);
        return;
    }
    IntNode p = first;
    while (p.next != null) {
        p = p.next;
    }
    p.next = new IntNode(x, null);
}
```

The above is a possible solution, but there is a slightly easier way of doing this, which is more elegant. Simple code is usually not good code, and these special cases are not simple.

How do we fix this? Well, we can make all `SLList`s the "same" -- they should have the same structure.

We can represent an empty list using a **sentinel** node. When we add to it, stuff will get added to the end, but the sentinel node will still be there.

![image-20200212175429386](C:\Users\winst\AppData\Roaming\Typora\typora-user-images\image-20200212175429386.png



Let's implement this:

```
public class SLList {
    private static class IntList {...}

    private IntNode sentinel;
    private int size;

    public SLList() {
        sentinel = new IntNode(63, null);
        size = 0;
    }

    public SLList(int x) {
        sentinel = new IntNode(63, null); //Arbitary value
        sentinel.next = new IntNode(x, null);
        size = 1;

    public void addFirst(int x) {
        sentinel.next = new IntNode(x, sentinel.next);
        size += 1;
    }

    public void getFirst() {
        return sentinel.next.item;
    }

    public void addLast(int x) {
        size += 1;
        IntNode p = sentinel;

        while (p.next != null) {
            p = p.next;
```

```
        }

        p.next = new IntNode(x, null);
    }
}
```

This feels harder than before, but in larger systems, this will save you a lot of time in not having to worry about special cases.

Invariants

We can generalize this idea of simplification using the term **invariant**. An invariant is a condition that is guaranteed to be true during code execution, assuming there are no bugs in your code. An `SLList` with a sentinel node has at least the following invariants:

- The `sentinel` reference always points to a `sentinel` node.
- The first node, if it exists, is always at `sentinel.next`.
- The `size` variable is always the total number of items that have been added.

Invariants make it easier to reason about code. We can assume they are true to simplify our code (for example, `addLast` no longer needs to worry about nulls), and we must also write methods that preserve these invariants.