

CS231 Fall 2013 Lecture 9

Stephanie R Taylor

September 23, 2013

1 Administrative Topics

- Return the quizzes
- Here is a list of terms that will appear in lab today, some of which we will cover today, some of which we will cover on Wednesday. There are enough instructions in the lab to write the code. Terms: iterator, implements, interface, inner class, generic, linked list, and node.

2 Review of what we want from lists

Let's think about a general-purpose List class that does everything you might ever want to do to with a List. Here is what comes to mind:

- add new data at the end
- insert new data other than at the end
- remove data from anywhere
- check size of the list
- print out the data in the list as a string
- clear

- test equality
- check if it contains a specific value
- see what is in the i-th position
- find where a specific value is
- replace one of the values with another one
- get a sublist of its data

3 Improving the IntList class

- Now that we can add numbers to the list, let's see how many we can add and how quickly we can do it. (i.e. Let's see how efficient our MyIntList class is.)
- Let's create an empty list and then add 1 million integers to it. (1 million is not exorbitant since 1 zettabyte of data is generated each year.) [Run it in class]
- Why was MyIntList so slow? We needed to keep creating new bigger arrays and copying the data from the old array to the new bigger array. Let's see if we can calculate how long it will take.
- Let's count how many array copies are done: $(1 + 2 + \dots + 999,999) = ?$ [approx 500 billion elements copied]
- Is that a big deal? Let's assume Java can do a billion copies per second. How long will this take? 500 seconds = 8 minutes, and that's not counting all the time taken by the other instructions in the program.
- Why is it taking so long? Too much copying! Instead let's make BetterIntList class. We will make the array bigger than we need and keep track of which slots in the array are full and which are empty. Start with an array with 10 empty slots. After filling them, create a new array twice as big and copy the old data over. Then we'll have a half-full array and so we can gradually fill the second half. When that array is

full, again create a new array twice as big and copy the old data into it, and we'll again have a half-full array.

- Let's look at the code for such a class (see BetterIntList class source code)
 - How do we keep track of how full the array is? We can't use `data.length` anymore. [a new field `size`] That's right. Every time we add an element, we increment `size` and every time we remove an element we decrement `size`.
 - Run it. Why is it so fast? How many array copies does it do? $[1 + 2 + 4 + 8 + \dots + 500000] = 1$ million. Time spent array copying: 1 millisecond.
 - What would happen if we wanted to insert a new element in the middle of the list?
 - Let's now implement `insert(int x, int index)`. [do it in class].
 - What about the fact that insert at front is so slow? Can we speed it up like we sped up the `add()` method? What do you suggest?
 - When copying to new larger array, copy to middle of array so that can add both on front and on end. But what about inserting in the middle?
 - If order is important, then there is no easy way using arrays to speed it up.
 - Conclusion: If inserting in the middle is what you will be doing most of the time instead of adding at the end, then what you need is a new data structure!

4 Linked Lists

Idea: Think of all data as being stored in node objects. You somehow need to keep track of which data nodes follow which other data nodes in a list.

One solution: Store the order info in some other object independent of the data, such as an array.

Better Solution: Let the data nodes keep track of it.

4.1 Class for Nodes

The idea is to have an object, called a `ListNode`, that contains some instance variables containing information (like our `int`) and an extra instance variable referring to another object of the same type `StringNode`. For variety, let's store strings in our list instead of integers. In this example, let's store only 3-char airport codes. For example, look at:

```
class StringNode {
    String data;
    StringNode next;

    public StringNode (String d, StringNode n) {
        this.data = d;
        this.next = n;
    }
}
```

Notice that a `StringNode` contains an instance variable named `next` that refers to another `StringNode`. So you could have a long list of `StringNode` objects, each referring to the next `StringNode` objects. Does that mean a `StringNode` object can contain a reference to itself? [Yes, although it is usually not useful to do so.] The `String` in a `StringNode` is an arbitrary bit of data we will be using as a running example for this chapter. In general, list nodes can contain any kind of data (including other lists).

Let's build a list by moving around pointers:

```
// create three new nodes, unlinked
StringNode node1 = new StringNode("PWM", null);
StringNode node2 = new StringNode("ORD", null);
StringNode node3 = new StringNode("OAK", null);
node1.next = node2;
node2.next = node3;
```

-or-

if we don't need `node2` and `node3` variables any further-

```
StringNode node1 = new StringNode("PWM", new StringNode("ORD", new StringNode("OAK",
```

What is the value of `node1.next.data`? How about `node1.next.next.data`?

Let's generalize. How would we go about adding a node to the head of the list?

- Make a new node with the right data, and with its next field pointing to the old head of the list
- Update the head-of-list pointer to point to the new node

What about finding the length of the list? We would need a node pointer that would start at the head, then follow the next links until we get to the end (it will be null). As we do this we need to update a counter. Then return the value of the counter.

Moral: There is lots of code necessary to manage the list.

4.2 Class for LinkedLists

So we should have a class `StringLinkedList` that managed `StringNodes`. This code is a sort of container for `StringNodes`. So let's define it like this:

```
public class StringLinkedList {

    // field
    StringNode head;

    private class StringNode {
        // StringNode methods here
    }

    // StringLinkedList methods here
}
```

In this case, both classes can be in the same file. The `StringLinkedList` class will have methods that let you add and remove Strings from the list. It will use `StringNodes`. But the outside world never needs to know that. So `StringNode` is going to be an *inner* class. And it will be private.