Lab 3: Double Or Nothing

Pre-Lab Notes

Lab submissions will **not be accepted late** – they must be submitted by **midnight the day after they're assigned** (to account for time zone issues). Some latitude might be allowed (I won't disqualify a submission that's mere minutes late!) but that's purely discretionary.

Labs are treated differently from assignments – you're encouraged to work with fellow students, ask questions of the TA during your lab section, and generally collaborate to complete your work. You should still submit your own work, **including citing with whom you collaborated in your submission**, but using answers developed by others will not be treated as plagiarism.

Nevertheless, you are still encouraged to make a good faith effort to complete your labwork, in order to exercise your understanding of the topics it covers. Lab submissions are marked purely 1 or 0 by whether the TA believes you at least made an honest attempt, even if you didn't succeed, so the value is in what the attempt teaches you rather than simply having the right answer.

1. The Array List and the Node List

We recently covered the Array List in lecture, discussing how it is the full data structure version of the primitive array, complete with an ADT and concrete class implementations in languages like Java. In particular, we discussed the idea of a linear set of data elements, known as a sequence, and how arrays and lists are different ways to implement that idea.

However, we did not explore the other side of that coin – if Array List is the formal version of the arrays we studied earlier, what is the formal version of the primitive linked list? These are known as Node Lists.

Today's lab will give you some practice with both the Array List and the Node List, by asking you to implement each of them. You won't simply be using the completely default version of either class, however, since these already exist in Java. Instead, each part of the lab will ask you to make a modification in your implementation, giving us a reason to get some practice in with each of these data structures and their inner workings.

This lab is also partly practice for assignment 2, both in terms of the content covered and also the format of the submission.

1.1 The Node List ADT

To give you a quick overview of the Node List data structure, then, a (capital L) List is defined as a linear sequence of data elements stored as **positions**, to contrast with an array's index. A position is relative, meaning that each element is defined as being before or after some other element, like the head being at the start of the sequence and the tail being at the end. Despite the relative nature of positions, from moment to moment they can still be turned into a number if need be – we can talk about adding something in the third position meaning that it belongs behind the first two positions, for example.

In a Node List, each position is filled with an object, known as a Node, which stores the data element belonging in that position as well as a reference (or references) to other positions and their nodes. The formal methods in a Node List's ADT include:

-F**irst**: Returns the node at the start of the sequence.

-L**ast**: Returns the node at the end of the sequence.
-P**rev**: Returns the node previous to a given node.
-N**ext**: Returns the node after a given node.
-S**et**: Replace the element stored in the node at a given position, returning the old element.
-**addFirst**: Insert a new node as the first node of the sequence.
-**addLast**: Insert a new node as the last node of the sequence.
-**addBefore**: Insert a new node into the sequence before a given position.
-**addAfter**: Insert a new node into the sequence after a given position.
-**Remove**: Removes the node at a given position from the sequence and returns it.

This isn't to say that a fully-featured Node List class might not want to implement other functions, of course, nor does it comment on how to implement them (which may vary on the needs of the programming language or the specific situation). For example, this ADT doesn't even clarify whether the list is singly-linked or doubly-linked – it doesn't define the node class at all. Any class claiming to be a concrete implementation of the Node List ADT, however, must at least cover the methods outlined above.

2. Outline

Below are two requests, one for a customized ArrayList class, and one for a customized NodeList class. In each case, you should write a separate file for the data structure being built, though they may share the same test class (likely a main class with a main function that prints text output to the terminal).

2.1 Changing Arrangements

At the end of the lecture on Array Lists, we briefly discussed how the standard Java ArrayList class handles the array's need for a fixed size by doubling its current capacity whenever it is asked to add a new element beyond the current maximum. This might be a useful technique for other data structures to emulate, but why stop there? Why not also shrink the Array List whenever it drops to below half of the current maximum capacity?

For this part of the lab, write a version of the ArrayList class (from scratch, not inheriting or implementing ArrayList or any other class) whose add and remove functions will double or halve the stored array, if the number of stored elements exceeds the maximum or drops below half the maximum. To simplify matters, we'll let this version of Array List simply store an array of integers (rather than any sort of object). Our attention is also focused on the add and remove functions, so the other functions of the ADT can just be filled in as-needed to support those two and any testing you do.

Demonstrate your work in a test class with a main method that prints both the ArrayList's size (as in filled elements) as well as the size (as in capacity) of the actual array stored within it.

2.2 Adaptation

While we know from section 1.1 of this lab that code for a doubly-linked list of nodes is, in essence, an implementation of the Node List ADT, we also know from our unit on the Adapter design pattern that “essentially the same” isn't equal to “exactly the same” in the world of programming objects. If the class hierarchy for a software system being laid down by a developer calls for a NodeList class, sending them a LinkedList class won't work.

For this second lab question, you're to implement your own NodeList class that fulfills the

NodeList ADT by adapting a standard Java LinkedList class of doubly-linked nodes. Your custom NodeList class doesn't need to extend or implement any other Java class, for now – it only needs to make concrete the ADT of the Node List class as described in 1.1, by including a LinkedList of nodes. You'll also want to define your own node class to fill the LinkedList with, which for simplicity's sake we'll say will carry a simple String variable as a data element..

Once your NodeList class is complete, be sure to run it with a test class that will try each of its functions. Each of them should produce the desired effect, such as being able to add String words as new nodes, print the Strings from the stored nodes, remove certain nodes, etc.

3. Deliverables

All lab submissions will be done through CourSys at https://courses.cs.sfu.ca/. For this lab, that should include:

1. A .zip archive of your code, organized into a single Java project. To ensure compatibility, it's recommended to use IntelliJ while developing your code, then using the export function under File -> Export -> Project To .zip File. **Don't forget to include a test file**, a Main.java with a main method that prints the results of a few runs of your functions to the terminal would be best.

Be sure you tidy up your code before submitting! Check the Java code style guide posted on the course website, and do your best to provide helpful comments.