COMP 2522
Object Oriented Programming 1
Lab 7

Yue Wang
yue_wang@bcit.ca

Due at or before 11:59:59 PM on Sunday, March 26, 202

1 Introduction

Welcome to your next COMP 2522 lab.

In this week's lab, you will implement a Node class and a DoublyLinkedList class in order to better understand how to traverse a doubly linked list and add/remove nodes from it.

2 Submission requirements

This lab must be submitted to D2L at or before 11:59:59 PM on Sunday, March 26, 2023.

Don't wait until the last minute as you never know what might happen (computer issues, no Internet, power outage, ...). Submit often, and please make sure your comments are short and clear and specific.

You must not ignore the code style warnings produced by IntelliJ. **You must style your code so that it does not generate any IntelliJ warnings.**

When code is underlined in red, IntelliJ is telling us there is an error. Mouse over the error to (hopefully!) learn more.

When code is underlined in a different colour, we are being warned about something. We can click the warning triangle in the upper right hand corner of the editor window, or we can mouse over the warning to learn more.

3 Grading scheme

Figure 1: This lab, like all labs, will be graded out of 5

Your lab will be marked out of 5. Consult the rubric available on D2L. On a high level, expect that marks will be assigned for:

1. (3 point) Meeting the functional requirements in this lab, i.e., does the code do what it is supposed to do?

2. (1 point) Meeting non-functional Java OOP design requirements such as encapsulation, information hiding, minimizing mutability, maximizing cohesion and minimizing coupling, making good choices about associations and thinking about Demeter, etc.

3. (1 point) Writing code that is self-documenting and well-styled that generates no warnings from IntelliJ or Checkstyle, with correct and complete Javadocs, using short and atomic methods correctly encapsulated in classes that are well defined.

4 Style Requirements

Continuing with this lab, there are code style requirements that you must observe:

1. Your code must compile.
2. Your code must not generate any IntelliJ code warnings or problems.
3. Your code must execute without crashing.
4. Your code must not generate any Checkstyle complaints (unless they are complains I have specifically said you can ignore).
5. Don't squeeze your code together. Put a blank space on either side of your operands, for example. I will be assessing the readability and clarity of your code.
6. All of your program classes must be in package ca.bcit.comp2522.xxx (replace xxx as required by the assignment, lab, quiz, etc.). For example, today's work should go in the ca.bcit.comp2522.labs.lab0X package.
7. All classes require Javadoc comments including the @author tag AND the @version tag. Class comments go after the package and import statements. There should be no blank lines between a class comment and the class it describes.
8. Public constants require Javadoc comments, private constants do not. Constants should be the first thing in your class.
9. Constants should be static and final, are often public, and should be followed by instance variables.
10. Instance variables have private or protected visibility, never public or default visibility.
11. Public and protected methods require Javadoc comments including @param tag(s)the @return tag, and the @throws tag where needed (we won't worry about throws until we talk about exceptions in depth).
12. A method's comment must begin with verbs describing what the method does, i.e., Calculates, Returns, Sets, Prints, etc. Note that we use present tense in Java – Returns, not Return. Prints, not Print.
13. The @return and @param tags go AFTER the description.
14. Private methods require non-Javadoc comments (the comments that start with a slash and a single asterisk).
15. Do comment complicated logical blocks of code inside methods with sparse, clear inline comments.
16. Do not use magic numbers (you must use constants instead). Remember that a magic number is any numeric literal. A constant can be local in a method (use the final keyword with it) or class-level (make it static and ALL_CAPS).
17. All method parameters that are object references must be made final (so we don't forget parameters are passed by value):
    (a) Nice to prevent erroneous assignments, and necessary if parameter is referenced by inner class, but that is perhaps a little advanced for now.

(b) References made final mean that the reference, once pointing to an object, cannot be changed to point at a different object.

18. Consider making your methods final:
    (a) Making a method final prevents subclasses (those that inherit the method) from changing its meaning.
    (b) Final methods are more efficient (the methods become inline, thus avoiding the stack and generating overhead).
19. Data and methods that work together must be encapsulated in the same class.
20. Code duplication must be minimized.
21. The values of local variables that are primitives are set when they are declared, and local variables are not declared until they are needed.
22. Every class that stores state needs an equals method, a hashCode method, and a toString method.
23. In general, we enforce a fairly hard maximum method length of 20 lines of code. Aim for 10 (excluding braces).

5 Implementation requirements

In class, we designed a doubly linked list using nodes. Each node has three instance variables, including an Object, a pointer to the previous node, and a pointer to the next node. A doubly linked list keeps track of the head (first node) and tail (last).

Create a Node class and a DoublyLinkedList class to implement this data structure. The following methods must be added to your DoublyLinkedList class:

1. Add three different methods for adding a new node to the doubly linked list:
    (a) prepend(Object data) creates a Node for the data and adds the new Node to the beginning of the list of Nodes
    (b) append(Object data) creates a Node for the data and adds the new Node to the end of the list of Nodes
    (c) add(Object data, int index) creates a Node for the data and inserts the new Node at the specified index. If the specified index is too large or if it is negative, throw an IndexOutOfBoundsException from the method.
2. public int size( ) which returns the number of nodes in the list.
3. public void clear( ) which reduces the list to length zero. There is an easy way to do this and a not-so-easy way to do this. Take advantage of garbage collection and do it the easy way.
4. public Object get(int index) returns a reference to the Object at the specified index. If the specified index is too large or if it is negative, throw an IndexOutOfBoundsException.
5. public Object remove(int index) removes the Object at the specified index. If the specified index is too large or if it is negative, throw an IndexOutOfBoundsException. Otherwise, remove the Node from that index and extract and return the data inside it. Consider all the cases, i.e., when the index = 0, when the index = size( ) - 1, when the index is too large, and when the index is somewhere in between.

Good luck, and have fun!