CS 249 Project 2  
Linked Lists, Stacks, and Queues

# Overview

This project requires the completion of the following tasks:

* Write a linked list as an ADT
* Write a queue as an ADT
* Write a stack as an ADT
* Use these ADTs to create a program which can process infix or postfix calculator operations efficiently.

## Unit Testing

This project requires that for each part you must run the included unit tests to validate and verify the correctness and integrity of your programs, as well as to drive a test-driven development approach. Remember that test-driven development means that test and compilation errors are not necessarily a bad thing; you should use them as an indication of you what code you need to create or fix next.

Start by downloading the interface and unit tests files provided for this assignment and setting-up JUnit for your specific Java development setup. Below are some quick tips for getting started running JUnit tests in several common development setups.

After you turn in your assignment you will be graded using these tests, as well as additional unit tests, so make sure you are thorough in your implementation.

### IntelliJ:

* Create a new project and add the interface and test files to the *src* directory
* Open up one of the test files from within IntelliJ and click on one of the lines underlined with an error
* Press ***alt+enter*** (***option+enter*** on mac) and select the option to import the JUnit library into your project
* You should now be able to run each of the tests (***right-click*** the file and choose *run*), or all of the tests (***right-click*** on the project and choose to run all tests) once you create the classes required for this project

### Eclipse

* Create a new project and add the interface and test files to the project *src* directory (you might need to ***right-click*** and refresh the *src* icon in Eclipse)
* ***Right-click*** on the project in the Package Explorer panel, go to ***Build Path->Add Libraries***, and select JUnit
* You should now be able to run the unit tests by ***right-clicking*** on the project or test files under Project Explorer, or by going to the ***run->run as*** menu

### Command Line

For a command line development environment, you will need to perform several steps:

* Download the required JUnit JAR files to your project directory
* Add the JAR files to your project class path when compiling and running
* Create your own driver class (a class with a main method) to run the tests and print out the results

# Writing Advanced Data Types

In the first step we will write and test our data structures, starting with linked lists. Remember that these advanced data structures have two sections: the public section which would normally define the interface entirely, and a private section which contains implementation details that would be private outside of this course. These private elements are only exposed to allow us to use unit testing.

**In this assignment you may not use any java collections or sorting algorithms, or arrays.**

1. Write a linked list ADT class called LinkedList satisfying the provided interface from ILinkedList.java
   * Your linked list nodes must be objects of a class which implements the INode interface. This interface is provided inside of the ILinkedList interface as an inner interface.
   * It is recommended that you store a link to both the first and last nodes in the linked list.
   * Verify completion with the LinkListTest.java unit tests.
2. Write a queue ADT called Queue satisfying the provided interface from IQueue.java
   * This queue must be built using the linked list from step 1 as the underlying list structure.
   * Verify with the QueueTest.java unit tests.
3. Write a stack ADT called Stack satisfying the provided interface from IStack.java
   * This queue must be built using the linked list from step 1 as the underlying list structure.
   * Verify with the StackTest.java unit tests.

# Writing an Infix and Postfix Calculator

Typically mathematical equations are expressed using what is called infix notation, where operators appear in-between their operands. The following equations are all presented in infix notation.   
3 + 5  
4 – 2 + 7  
3 + 2 \* 5 / 10 + 11

Postfix notation is another way to write equations in which the operators come after their operands. This notation is commonly used for computation in engineering fields. Many HP calculators utilize postfix notation, as an experienced user can enter equations much faster than using infix notation. The same equations listed earlier could be presented in postfix form as follows.  
3 5 +  
4 2 – 7 +  
 3 4 5 \* 10 / + 11 +

Solving the final equations in postfix is done as follows, where the green highlight shows the portion of the equation that is computed to reach the bold portion in the next step.  
3 4 5 \* 10 / + 11 +  
3 **20** 10 / + 11 +  
3 **2** + 11 +  
**5** 11 +  
**16**

One advantage of postfix notation is the freedom from order of operations. Compare the last of these three equations. Where in infix notation you must remember to multiply and divide before adding, in postfix notation there is no need to consider this. Consider the following two equations in infix notation.  
2 \* 3 + 1  
2 \* (3 + 1)  
Which translate into the following postfix notation.  
2 3 \* 1 +  
2 3 1 + \*  
Demonstrating how postfix notation can represent more complex equations without the use of parenthesis.

Stacks are an ideal tool for working with mathematical notation. Hold a stack of numeric operands, and whenever you are given an operator, pop the previous two operands off the stack, apply the operator, and push the result of the operation back onto the stack. As an example of solving the example equation above using a stack:  
3 (entered 3)  
3 4 (entered 4)  
3 4 5 (entered 5)  
3 20 (entered \*)  
3 20 10 (entered 10)  
3 2 (entered /)  
5 (entered +)  
5 11 (entered 11)  
16 (entered +)

Stacks can also be used to convert from infix notation to postfix notation. This time, instead of using a stack to track the operands we can use it to track the operators. We can also apply order of operations by popping and writing from the operator stack whenever we see a new operator of lower precedence. Consider the following equation.  
2 \* (3 + 4) + 5 \* 6  
To convert this into postfix notation we hold a string to build our result string and use a stack of operators to remember in which operators have not yet been included into our result string. The following table shows the state of our result string and operator stack for each entered operator.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator Stack** | **Posfix String** | **Entered** | **Notes** |
|  | 2 | 2 |  |
| \* | 2 | \* |  |
| ( \* | 2 | ( |  |
| ( \* | 2 3 | 3 |  |
| + ( \* | 2 3 | + |  |
| + ( \* | 2 3 4 | 4 |  |
| \* | 2 3 4 + | ) | When we hit a close parenthesis we write the operators until we reach the open parenthesis. |
| + | 2 3 4 + \* | + | + is lower in the order of operations, so we pop and write the \*. |
| + | 2 3 4 + \* 5 | 5 |  |
| \* + | 2 3 4 + \* 5 | \* | \* is higher in the order of operations than +, so it remains on the stack. |
| \* + | 2 3 4 + \* 5 6 | 6 |  |
|  | 2 3 4 + \* 5 6 \* + | (end) |  |

For this project you will write a program capable of reading equations in both infix and postfix notations.

1. Write a PostfixCalculator class which implements the interface provided in IPostfixCalculator.java.
   * Start by creating stubs and ensuring that the class compiles and that all unit tests in PostfixCalculatorTest.java fail.
   * Implement the inputOperator() and inputOperand() methods such that you can successfully add numbers by inputting two numbers followed by a ‘+’ operator.
   * Expand this to use all operators and ensure you can solve longer postfix equasions with these methods.
   * Implement inputToken, and then inputEquation.
   * Ensure you can pass all unit tests.
2. Write a InfixCalculator class which implements the interface provided in IInfixCalculator.java.
   * Start by creating stubs and ensuring that the class compiles and that all unit tests in InfixCalculatorTest.java fail.
   * Implement the inputOperator(), inputOperand(), and endEquation() methods such that you can convert a simple equation using the ‘+’ and ‘-‘ operators.
   * Expand your implementation to handle ‘\*’ and ‘/’ while respecting order of operations.
   * Expand your implementation to handle parenthesis.
   * Implement inputToken() and convertEquation() such that you can convert an entire equation at once.
   * Implement inputEquation() so you can get the result of a equation in a single call.
   * Ensure you can pass all unit tests.

# Write-up and Submission

* For each part of the project, describe your approach, solutions, describe any difficulties that you encountered, and detail the efficiency of all non-test methods in your code using Big-O notation. Include any requested output and screenshots for each part.
* Add all of your *.java* files and your write-up document to a .zip or .tar.gz archive and submit it on Bb Learn. Make sure all of your java files are in the root of the zip file, rather than inside additional folders.
* You **must** write and submit your own code. You must also clearly identify and online or text sources that you used as references, any collaborators with which you discussed the project (or lack thereof), and how the source was beneficial.

# Grading

This project is worth 100 points.

* 10 Points – Design and code quality
  + Good object-oriented design, consistent comments, white space, indentation, etc.
* 70 Points – Implementation quality
  + Demonstrate that your solution is complete and accurate based on unit tests.
  + 20 points – LinkedList
  + 10 points – Queue
  + 10 points – Stack
  + 15 points – PostfixCalculator
  + 15 points – InfixCalculator
* 20 Points – Writeup and Submission