Due : Oct. 13 (10/13), 13:00

Overview

This assignment consists of four parts: implementing a stack, implementing a queue, evaluation of prefix notation expressions, and evaluation of postfix notation expressions.

General Notes

- Do not use Eclipse. We recommend Sublime Text (Linux/Mac/Windows), Atom (Linux/Mac/Windows), Notepad++ (Windows), TextWrangler (Mac), or VSCode (Linux/Mac/Windows).
  - Code that contains the line "package ..." at the beginning of the file breaks our autograder and Eclipse automatically adds that line.
- Do not change any method or class signatures. You should only edit inside of the functions. If your code changes any class or method names or signatures, you will receive an automatic 0. You should not implement any other functions or instance variables besides the ones that are provided, unless explicitly allowed.
- Make sure your code compiles. Non-compiling code will automatically receive a 0. If you have a problem that is causing you to not be able to compile, it may be better to just comment out the incorrect code and return a dummy value (something like null or -1) so the rest can compile.
- Make sure that your code does not print out anything (there should be no System.out.println in your code). You will receive an automatic 0 if your code outputs something to STDOUT during the tests.
- To ensure that your code will be accepted by the autograder, you should submit your code on YSCEC, download it again, unzip it, recompile it and check the provided test suite. This way, you know that the file you are submitting is the correct one.

Introduction

In this assignment, you will implement a generic stack and queue based on a linked list. We have provided a fully functoning linked list implementation in the form of a class file. The linked list is singularly linked, linear, has a head and tail pointer, and implements the following methods:

- `LinkedList()`
- `void insert(int index, T data)`
- `void remove(int index)`
- `void clear()`
- `T getData(int index)`
- `int getSize()`
- `String toString()`

You should not edit any code in the `LinkedList.java` file.

Stack

Stacks are a commonly used *LIFO* (Last In First Out) data structure. In this homework, you will implement a generic stack using the provided linked list implementation. The methods to be implemented are:

- `Stack()`
- `T peek()`
- `T pop()`
- `void push(T data)`
- `void clear()`
- `int getSize()`
- `boolean isEmpty()`

The descriptions of these methods can be found in the `Stack.java` file. Note that **all** of these methods should run in $O(1)$ time.

Queue

Queues are another common data structure, but are *FIFO* (First In First Out), unlike stacks. Again, using the provided linked list implementation, you will implement a generic queue that has an optional maximum capacity. The methods to be implemented are:

- `Queue()`
- `Queue(int capacity)`
- `T peek()`
- `T dequeue()`
- `void enqueue(T data)`
- `void clear()`
- `int getSize()`
- `boolean isEmpty()`
- `boolean isFull()`

The descriptions of these methods can be found in the `Queue.java` file. Note that **all** of these methods should run in $O(1)$ time.

Postfix Calculator

As you learned in the lecture, *postfix notation* is a representation for mathematical expressions. For example, the expressions $5+10\times4$ would be written as 5 10 4 $\times$ +. Stacks and queues can be used ot evaluate expressions written in postfix notation. Here you will use your stack and queue implementations to evaluate postfix expressions. You will implement only two methods:

- `PostfixCalculator()`
- `int evaluate(String exp)`

The descriptions of these methods can be found in the PostfixCalculator.java file. For this assignment, we will guarantee several things:

- We will only use the operators $+$ $-$ $\times$.
- We will only consider non-negative integers as terms.
- All inputs to solve are guaranteed to be valid postfix notation expressions of length at least 1.
- Terms and operators will be separated by a single space.
- The expression evaluation will never overflow Java's `int`.

Prefix Notation

In addition to infix and postfix notation, prefix notation is also frequently used. In prefix notation, the operations appear before the terms they operate on. For example, $5 + 10 \times 4$ would be written as $+ \ 5 \ \times \ 10 \ 4$. Again, you will use your stack and queue implementations to evaluate prefix expressions. The methods to be implemented are:

- `PrefixCalculator()`
- `int evaluate(String exp)`

The descriptions of these methods can be found in the PrefixCalculator.java file. For this assignment, we will guarantee several things:

- We will only use the operators $+ \ - \ \times$.
- We will only consider non-negative integers as terms.
- All inputs to solve are guaranteed to be valid prefix notation expressions of length at least 1.
- Terms and operators will be separated by a single space.
- The expression evaluation will never overflow Java's `int`.

General Directions

- Write your name and student ID number in the comment at the top of the java files.
- Implement all of the required functions in the java files.
- You should not import anything that is not already included in the file.
- Pay careful attention to the required return types and edge cases.

Submission Procedure

To submit your homework, simply drag and drop each file individually into the attachment box on the YSCEC assignment page. You should **NOT** zip them or place them in any folder. For this assignment, you should submit only the following files:

- Stack.java
- Queue.java
- PostfixCalculator.java
- PrefixCalculator.java
- ⟨student_id⟩.txt

Inside of the ⟨student_id⟩.txt file, you should include the following text, and write your name at the bottom.

In completing this assignment, I pledge that I have not given nor received any unauthorized assistance.

If this file is missing, you will get a 0 on the assignment. It should be named *exactly* your student id, with no other text. For example, `2017123456.txt` is correct while something like `2017123456_hw1.txt` will receive a 0.

Testing

We have provided a small test suite for you to check your code. You can test your code by compiling and running the tester program. You will always get a notification if you fail any of the tests, but you may not get a notification if you passed.

Note that the test suite we use to grade your code will be much more rigorous than the one provided here (and not necessarily a superset of the provided tests). You should consider making your own test cases to check your code more thoroughly.