Due : Oct. 10 (10/10), 13:00
Late submission due: Oct. 11 (10/11), 13:00

Overview

This assignment consists of four parts. Happy holiday.

General Notes

- *Read this homework guideline carefully.* If you do not follow the guidelines, you may receive a 0 regardless of whether your code works or not.
- Do not use any IDEs (Eclipse, IntelliJ IDEA, etc.)
  - We recommend Sublime Text (Linux/Mac/Windows), Atom (Linux /Mac/Windows), Notepad++ (Windows), or TextWrangler (Mac).
  - IDEs often create a "package" of your code, which breaks the autograder.
  - **If you know how to fix the package problem**, you can use any IDE you want. However, we will not answer any questions related to this problem since we have already recommended a solution.
- Do not change any method or class signatures. If your code changes any class or method names or signatures, you will receive an automatic 0.
- 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.
- To ensure that your code will be accepted by the autograder, you should submit your code on YSCEC, download it again, recompile it and check the provided test suite. This way, you know that the file you are submitting is the correct one.
- You can use any course materials. However, if you do not cite the source you referred to, it might be checked as copied code. Please write the material (and page) you referred in comments.

1 Stack

A stack is a LIFO (Last-In First-Out) data structure. The stack can be applied to multiple programs. Implement a stack structure, and use it to implement the following program.

1.1 Stack

You will implement the stack data structure based on linked list in Stack.java file. The stack should be able to store an arbitrary type (given at the construction) of data. You should use the Node class we provided.

You must implement the following methods as well as its constructor.

`push`: Push an item onto the top of the stack.

`pop`: Remove an item from the top of the stack and return that item. If the stack is empty, raise an exception.

`size`: Return the number of items currently in the stack.

1.2 LISP

In a LISP-like input, multiple types of brackets must exist in pairs. The types of brackets used in the input are square brackets (‘[’,‘]’), curly braces, (‘{’,‘}’), and parentheses (‘(’,‘)’). If the brackets do not match, the program should notify that the input is incorrect. This input must follow the four conditions:

1. A pair of brackets contains a left bracket and a right bracket of the same type.
2. A left bracket must precede the paired right bracket.
3. There are no two pairs containing the same bracket.
4. Pairs of brackets must not intersect each other.
   e.g. ({)} is incorrect.

Implement a method `checkBracketBalance` that verifies whether or not the input meets all of the above four conditions in the LISP.java file. You may assume that the input will only consists of brackets with no space between them.

2 Postfix calculator

Postfix notation is a representation for mathematical expressions. For example, the expression $5 + 10 \times 4$ would be written as $5\ 10\ 4\ \times\ +$. Stacks and queues can be used to evaluate expressions written in postfix notation.

Here you will use a stack for evaluating postfix expressions in the PostfixCalc.java file. You will implement the `evaluate` method, which returns the value of the given postfix expression.

For this assignment, we guarantee that the following conditions hold for inputs:

- An input contains $+$, $-$ and $*$ (addition, subtraction and multiplication, respectively) as operators.
- A term (number) in input is always non-negative integer.
- A given input is always valid postfix expression of length at least 1.
- Terms and operators are separated by a single space.
- Every intermediate result of a given input will never overflow/underflow Java's `int` type.

3 Cafe

In Yonsei, there is an excellent cafe called Hanul-Sam. It serves several types of coffee for the students. Unfortunately, Hanul-Sam can handle at most two students at a time, and thus the students should wait in line to get their coffee. If the waiting time is too long, we should expand Hanul-Sam.

You will implement a program that simulates the services of Hanul-Sam and computes the total waiting time. The simulation starts at time 0. Implement the following methods with reference to the above process in Cafe.java file.

`arrive`: Notify that a student has arrived. The student's ID, arrival time, and the time that takes to make coffee is given as integers. You can assume that the arrival time is greater than the current simulation time and will come in ascending order.

`serve`: Simulate until at least one coffee is ready to serve: serve one coffee to a student and return the student's ID. If there are two coffees ready at the same time, you should serve any one of them and serve the other coffee at the next call.

`stat`: Return the total waiting time of the served students.

The running time of the `arrive` and `serve` methods must be $O(1)$.

4 k-ary tree

4.1 k-ary tree

A k-ary tree is a tree whose each node can have at most k children.

You will implement a k-ary tree that stores an arbitrary type (given at the construction) of data in Tree.java file. The details of each methods are in the source file. You must implement the following methods as well as its constructor:

`root`: Return the root node of the tree.

`add`: Insert the given node at the possible leftmost empty slot of smallest depth and return the inserted node. You may assume that the leftmost child is at index 0 of its parent, except root.

`detach`: Detach the subtree that has the given node as the root. The subtree itself remains intact.

`arity`: Return k (the number of possible children for a node).

`size`: Return the number of nodes in the tree.

`height`: Return the height of the tree.

`isEmpty`: Return whether or not the tree is empty.

4.2 Tree traversal

Implement a program that traverses the *k*-**ary tree** in three ways. You will implement this alongside with your tree implementation in the Tree.java file. This subproblem requires that the tree implementation works.

Preorder visits the root node of the subtree before visiting its child nodes.

Postorder visits the child nodes before visiting the root of the subtree.

Depthorder visits all node of the same depth from left to right before visiting the next depth.

You must implement the following methods as well as its constructor:

`preOrder()`: Return the sequence of items visited by preorder traversal.

`postOrder()`: Return the sequence of items visited by postorder traversal.

`depthOrder()`: Return the sequence of items visited by traversing the tree by depthorder.

General Directions

- Write your name and student ID number in the comment at the top of the files you submit.
- Implement all of the required methods.
- You should not import anything that is not already included in the file.
- Pay careful attention to the required return type and edge cases.
- All the codes we provide can be found in src/base directory. If you are unsure what a class/method exactly does, please refer to the code.
- You are free to implement any algorithm that you wish, but you must code it yourself. If you referred to any course materials, you must write the name of the material and page in comments. We will only be testing that your code produces a correct result and terminates in a reasonable amount of time.

Submission Procedure

You *must* make a zip file for submission using Gradle build tool (refer to Compiling section). For this assignment, the zip file will contain only the following six files:

- Stack.java
- LISP.java
- PostfixCalc.java
- Cafe.java
- Tree.java
- your_student_id_number.txt

You must rename 2020xxxxxx.txt to your actual student ID number. Inside of that text file, you must include the following text and write your name at the bottom. Please be sure to write all the following text including the last period.

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

If this file is missing, you will get 0 on the assignment. It should be named *exactly* your student id, with no other text. For example, *2020123456.txt* is correct while something like *2020123456_pa3.txt* will receive 0.

Compiling

This assignment uses Gradle build tool to automate compiling and testing procedure. The following command will test your Java code against the provided testcases:

```
% ./gradlew -q runTestRunner
```

The following command will zip the files for your submission. The zip file will be named with your student id (the name of .txt file) and will lie in "build" directory. Be aware that the command will be interrupted if your pledge does not comply the guideline.

```
% ./gradlew -q zipSubmission
```

Since the testrunner blocks the standard output from printing, it is hard to test your code fragment while writing the code. For this purpose, we also provide an empty Main class. As this file is not for submission, you may use any features Java provides. The following command will run the Main class instead of the testcases.

```
% ./gradlew -q runMain
```

On Windows, try `gradlew.bat` instead of `./gradlew` if you met an error. Moreover, you may omit the '`-q`' option to review the compile log.

Testing

We have provided small testcases (src/test) for you to check your code. You can test your code by the means mentioned above.

Note that the testcases we will use to grade your code is `very 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.