CS2401 – Weeks 14

In this lab assignment, we are going to practice using stacks and queues with binary trees. We hope you enjoy this assignment! Let's get started!

What is the scenario?

In this lab, we are going to play with arithmetic expressions.
You are going to use two main representations of arithmetic expressions: post-fix notation (see class notes for details on what post-fix notation is) and as binary trees (see class notes for details on what such binary trees are like).

You will use the post-fix notation to evaluate your expression.

You will use the binary tree representation to: 1/ evaluation your expression, 2/ print your expression in in-fix notation with parentheses, and 3/ traverse it in level-order fashion to gather the list of all of its values and variables.

Let's build data structures

For this, you are going to have to manipulate / design a few types. We are providing you with the following guidelines:

- Post-fix notation of an arithmetic expression → represented by a string. You will define a type PostfixExpression.
- Binary tree of an arithmetic expression → represented by a binary tree of data, called ExpressionBT that contains:
  - String type: can be "var", "value", or "operator"
  - char operator: can be '+', '-', '*', '/'
  - int value
  - String variable: can be any identifier for a variable, e.g., "x", "y", "z", "x1", "myVar".
  - Note: depending on what type contains, only one of the other 3 attributes will be relevant.

Let's go over the details you will need to implement for each type. Note that written in orange are all the methods and attributes given to you already:

Type: PostfixExpression
Private Attributes: String expression
Methods: Constructors:

- PostfixExpression()
- PostfixExpression(String e)

Accessors / getters:

- getExpression(): returns expression string

Modifiers / setters:

- setExpression(String newExpr): assigns newExpr to expression

Other methods:

- Evaluate(): traverses the expression using IntStack and returns an integer: the integer value of the expression
- Print(): prints out the expression in postfix notation

Type: ExpressionBT
Private Attributes: • String type • char operator • int value • String variable • ExpressionBT left • ExpressionBT right
Methods: Constructors: • ExpressionBT() • ExpressionBT(String[] e) Accessors / getters: • getType() • getValue() • getVariable() • getLeft() • getRight() Modifiers / setters: • setType(String t) • setValue(int v) • setVariable(String var) • setLeft(ExpressionBT b) • setRight(ExpressionBT b) Other methods: • evaluate(): traverses the expression using recursion and returns an integer: the integer value of the expression. Note: only if there are no variables in the expression. If there are variables, print out that you cannot evaluate and return 0 • print(): prints out the expression in infix notation with parentheses, using a BTStack

- allVariables(): void method. It prints out all variables in the tree, if any. If there is no variable, it prints out "no variable in this expression". This method should use a BTQueue.
- includesVariables(): returns true if the expression contains at least one variable, false otherwise

To implement a few of the above methods, you will need to have a few additional types:

- A stack of integers, called IntStack
- A stack of nodes (nodes that form your expression binary tree), BTStack
- A queue of nodes (nodes that form your expression binary tree): BTQueue

You are free to use any implementation you like of the above types, provided that they respect the following signatures of methods:

- For both stacks:
  - Peek(): returns the relevant content of the top element without removing it from the stack
  - Pop(): returns the top element and removes it from the stack
  - Push(data d): adds d on top of the stack
- For the queue:
  - Peek():returns the relevant content of the head element without removing it from the queue
  - Dequeue(): returns the top element and removes it from the queue
  - Enqueue(data d): adds d to the tail of the queue

Note: You may have to implement additional methods such as: isEmpty, isFull, depending on your implementation choices.

Note: we provide you with:

- Some code pertaining to ExpressionBT.java;
- Starter code for PostfixExpression.java;
- Starter code for IntStack.java;
- Starter code for BTStack.java; and
- Starter code for BTQueue.java.

You have to complete all of the above files by following the guidelines provided to you earlier in this document.

You also have to provide 5 test cases for each of the major methods: two Evaluate methods, allVariables, Print, in a file called BTSQTester.java.

Grading: there is a total of 350 pts → you are required to complete 220 pts to get 100%

10 pts Code is deemed to be of high quality

PostfixExpression.java: 40 pts

20 pts Evaluate()

20 pts Print()

ExpressionBT.java: 80 pts

20 pts Evaluate()
20 pts Print()
20 pts allVariables()
20 pts includesVariables()

IntStack.java: 60 pts
20 pts Peek()
20 pts Pop()
20 pts Push(data d)

BTStack.java: 60 pts
20 pts Peek()
20 pts Pop()
20 pts Push(data d)

BTQueue.java: 60 pts
20 pts Peek()
20 pts Dequeue()
20 pts Enqueue(data d)

BTSQTester.java: 40 pts
10 pts 5 test cases for Evaluate (for PostfixExpression)
10 pts 5 test cases for Evaluate (for ExpressionBT)
10 pts 5 test cases for Print (for ExpressionBT)
10 pts 5 test cases for allVariables (for ExpressionBT)

Due date: May 8th at 11:59pm

How to submit?
Check with your own TA for submission guidelines.

Failing to follow submission instructions and guidelines will result in up to 15 points off your overall grade in this lab. So please pay attention.