Assignment 4

DUE DATE: FRIDAY JULY 17, 2020 AT 11:59 PM

Notes:

- Submit ONE .java file for question 1. The .java file must contain ALL of the source code for the question. See the Programming Standards document for how to store several classes in the same file.
- Submit ONE .pdf file for question 2.
- Each filename must have the format `<your last name><your first name>A4Q1.java` (e.g. `SmithJohnA4Q1.java`). Please use your name as shown in UM Learn.
- Do not submit any output. Your code will be run during marking.
- Your program must compile, run, and end normally (not crash or get stuck in an infinite loop) to receive any marks. See the Assignment Information file for some tips on how to make sure your code runs for the markers.
- Assignments must follow the Programming Standards posted in UM Learn.
- Assignment submissions are only accepted via UM Learn. Submissions by email will not be accepted.
- You may submit your assignment multiple times, but only the most recent version will be marked.
- Assignments become late immediately after the posted due date and time. Late assignments will be accepted up to 49 hours after that time, at a penalty of 2% per hour or portion thereof. After 49 hours, an assignment is worth 0 marks and will no longer be accepted.
- The time of the last submission controls the late penalty for the entire assignment.
- These assignments are your chance to learn the material for the tests. *Code your assignments independently*. We use software to compare all submitted assignments to each other, and pursue academic dishonestly vigorously.

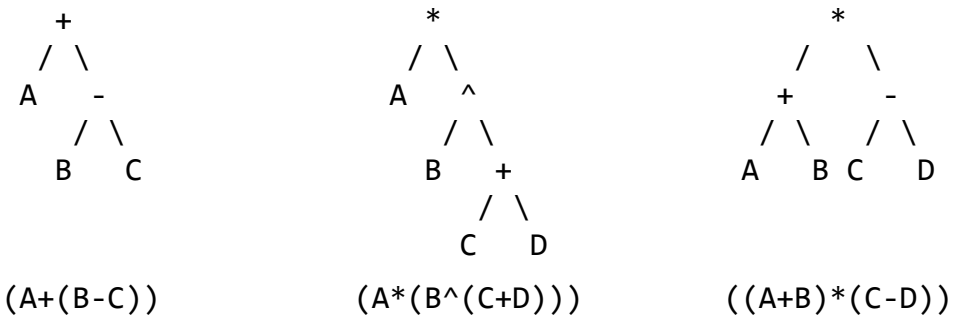
Question 1: Expression Trees [30 marks]

This question can be done after Week 8 of the course but reviewing Week 9 before beginning is recommended.

Read through all of the instructions for question 2 before beginning.

You will use an expression tree to store and manipulate algebraic expressions. Your expression tree will be constructed from nodes, as described below.

Examples of expression trees are shown in the Week 8 class materials. Some additional examples of expression trees are



One advantage of postfix and prefix notations is that parentheses are not needed to specify order of operations. In this question you will create expression trees from both postfix and prefix forms of algebraic expressions. You will also simplify the expression trees and print infix, postfix, and prefix versions of the expressions stored in the trees. Your program will read commands from a file and output the result of executing those commands.

Input

The input file will consist of one command per line, where the command will be one of the 6 options below. If additional information is needed, it will follow on the same line. Possible commands are:

- COMMENT – A line beginning with "COMMENT" should be echoed to the console. No manipulation of the tree is required.
- NEW – A line beginning with "NEW" will construct a new expression tree. Any existing tree will be discarded and replaced with the new tree. The expression that follows "NEW" will be either prefix or postfix notation. Your program should be able to determine which notation is used. All operands and operators will be separated by spaces, so that you may split the line read from file into tokens, where each token contains a String that will correspond to a node in the tree.
  Output a single statement, "New tree constructed", when a tree is successfully created. Please see the sections below on constructing expression trees for some hints on how to read in expressions and construct trees.

- PRINTPREFIX – A line beginning with "PRINTPREFIX" will print the current tree using prefix notation. Similar to the prefix notation used when reading from the input file, separate all operands and operators with a space.
- PRINTPOSTFIX – A line beginning with "PRINTPOSTFIX" will print the current tree using postfix notation. Similar to the postfix notation used when reading from the input file, separate all operands and operators with a space.
- PRINTINFIX – A line beginning with "PRINTINFIX" will print the current tree using infix notation. Infix notation requires parentheses to indicate the order of operations, and you will print a **fully parenthesized expression**, where each operand-operator-operand is enclosed in parentheses. Examples of fully parenthesized expressions are `( ( 8 + 6 ) * ( 4 - 5 ) )` and `( ( ( B + 5 ) * 4 ) ^ 3 )`.
- SIMPLIFY – A line beginning with "SIMPLIFY" will simplify the current tree, following common arithmetic rules, and output the statement "Tree simplified" when complete. Traverse the tree and stop to consider each node containing an operator. (You need to determine which type of traversal is best to use.) Look at the subtree consisting of the operator node and its left and right subtrees, and simplify where possible. Some examples to get you started:
  - If both children of an operator are numeric values, perform the operation and replace that subtree with the result stored in a single node.
  - A * 1 = A. If the operator is * and one of the children is 1, replace the subtree with the other child.
  - A * 0 = 0. If the operator is * and one of the children is 0, replace the subtree with 0.
  - A ^ 1 = A. If the operator is ^ and the right child is 1, replace the subtree with the left child.

Nodes for Expression Trees

The nodes in the expression tree will have the ability to hold three types of information: operators, variables, or constants. Possible operators will be +, -, *, and ^. We will omit division so that we can deal only with integers.

The fields in your nodes should be:

- A type that identifies the node as an operator, variable, or a number. Something similar to `enum NodeType{OPERATOR, VARIABLE, NUMBER;}` is appropriate.
- A char that will store the operator for operator nodes. The only valid characters are '+', '-', '*', and '^'.
- A String that will store the variable name for variable nodes.
- An int that will store the value for numerical nodes.
- Two Node references, that refer to the left child and the right child. For leaf nodes these will be `null`.

Note: Even though we will use '^' as the symbol (e.g. x ^ y to represent x^y), use the pow() method in your program. ^ in Java is a bitwise XOR.

The Expression Tree Class

Your expression tree class must have methods that allow it to carry out the expected commands. Recursion should be used when appropriate.

A Queue to store Nodes

You will need a queue of Nodes (i.e. a queue where each item stored in the queue is a Node) to aid in the construction of an expression tree from prefix notation.

You may choose the underlying implementation for your queue that you think is most appropriate (linked list or array). The implementation that you choose should be hidden from a user of the class. That is, the user will enqueue and dequeue Nodes, and will not know how they are managed inside the queue.

Your queue must have the following methods:

- A constructor that creates and empty queue.
- A `boolean isEmpty()` method that returns a boolean, indicating whether the queue is empty.
- A `boolean enqueue(Node toAdd)` method that will insert the given Node into the queue. This method will return true if the enqueue is successful, and return false if the enqueue fails.
- A `Node dequeue()` method that will dequeue and return the Node at the front of the queue. This method removes the returned Node from the queue. This method should return `null` if the user tries to dequeue from an empty queue.
- A `Node peek()` (or front) method that returns the Node at the front of the queue. This method does **not** remove the returned Node from the queue. This method should return `null` if the user tries to peek at an empty queue.

A Stack to store Nodes

You will need a stack of Nodes (i.e. a stack where each item stored in the stack is a Node) to aid in the construction of an expression tree from postfix notation.

As for the queue class, you may choose the underlying implementation for your queue that you think is most appropriate (linked list or array). The implementation that you choose should be hidden from a user of the class. That is, the user will push and pop Nodes, and will not know how they are managed inside the stack.

Your queue must have the following methods:

- A constructor that creates and empty stack.
- A `boolean isEmpty()` method that returns a boolean, indicating whether the stack is empty.
- A `boolean push(Node toAdd)` method that will insert the given Node into the stack. This method will return true if the push is successful, and return false if the push fails.

- A `Node pop()` method that will pop and return the Node on the top of the stack. This method removes the returned Node from the stack. This method should return `null` if the user tries to pop from an empty stack.
- A `Node peek()` (or top) method that returns the Node on the top of the stack. This method does **not** remove the returned Node from the stack. This method should return `null` if the user tries to peek at an empty stack.

Constructing An Expression Tree From Postfix Notation

Constructing an expression tree from postfix notation is similar to evaluating a postfix expression (discussed in Week 6, slides #142-157). However, instead of storing operands on the stack, you store entire subtrees. Read along the postfix expression from left to right, as we did when evaluating a postfix expression. When you encounter an operand:

- Make a Node that holds the operand.
- Push that Node onto the stack.

When you encounter an operator:

- Pop two operands/subtrees (B and C) off the stack.
- Create a new Node (A) that holds the operator.
- Attach B (the first node popped) as the right child of A.
- Attach C (the second node popped) as the left child of A.
- Push A onto the stack. Note that this effectively pushes the entire subtree onto the stack, because the stack holds node A, and A is linked to B and C.

When you're done evaluating the postfix expression, pop the one remaining item off the stack, which will be the root node of the tree that depicts the postfix expression.

Hint: Before writing any code, run through this algorithm on paper for a few examples. For example, the postfix expression DEF+* should produce the tree for the infix expression D*(E+F).

Constructing An Expression Tree From Prefix Notation

To construct an expression tree from prefix notation, make use of a queue of nodes to temporarily store subtrees as you build the full tree. Begin by reading the prefix expression from left to right and placing the entire expression into the queue. That is, for each operand or operator, create a node containing that token and put it into the queue. Until the queue contains only a single item, remove the item at the front of the queue:

- If the item is an operand or an operator that already has children, enqueue it again at the back/end of the queue without any modification.
- If the item is an operator without children, examine the next two items in the queue.
    - If they are both operands or operators with children, remove them from the queue.
        - The first item removed is the left child of the operator.

        - The second item removed is the right child of the operator.
        - Enqueue the operator again at the back/end of the queue (it is now linked to two children).
    - If they are not both operands, do not remove them from the queue (you'll process them on the next iterations of the loop), and enqueue the operator again at the back/end of the queue.

Hint 1: Note that while operator nodes should always have two non-null children in a valid tree, you will sometimes want to queue operator nodes temporarily without children while building the tree.

Hint 2: Notice that we want to be able to peek at (examine) two items in the queue without dequeuing them. For this assignment (even though it is not a real queue operation), add a `peek2` method to your Queue class. This method should return (without removing) the second item in the queue (and return null if a second item does not exist).

Hint 3: Before writing any code, run through this algorithm on paper for a few examples. For example, the prefix expression *D+EF should produce the tree for the infix expression D*(E+F).

Application (Main) Class – named <your last name><your first name>A4Q1

Your application class should process commands from an input file, as described above, until the end of the file. You may assume that the data file will be named `A4Q1input.txt` and will be located in the same directory as your .java file.

Sample Program Input

```
COMMENT Starting tests...
NEW C 3 + 5 4 - *
PRINTINFIX
SIMPLIFY
PRINTINFIX
PRINTPOSTFIX
PRINTPREFIX
COMMENT End of tests.
```

Sample Program Output

```
Starting tests...
New tree constructed
( ( C + 3 ) * ( 5 - 4 ) )
Tree simplified
( ( C + 3 ) * 1 )
C 3 + 1 *
* + C 3 1
End of tests.
```

Additional Notes

- You may assume that the input file is free of errors. That is, you may assume that the file contains one command per line, that all tokens within a line will be separated by a space, and that only valid commands are provided. You may also assume that given expressions are valid expressions (i.e. the number and order of operands and operators are such that they construct a valid algebraic expression).

- Place all classes for question 1 in the same file.

- Submit ONE .java file containing all the source code for question 1. The .java file MUST be named <your last name><your first name>A4Q1.java (e.g. SmithJohnA4Q1.java). Do not submit any output. Your code will be run during marking. Make sure that your code compiles without errors (see the Assignment Information file for some common issues).

[Programming Standards are worth 8 marks]

Question 2: 2-3-4 Trees [12 marks]

This question requires material from Week 10.

Similar to the examples seen in Week 10 for 2-3 and 2-3-4 trees, you will redraw a 2-3-4 tree after each insertion listed below. No code is required for this question.

In this question we are looking at the type of 2-3-4 tree that contains data in **both** interior and leaf nodes. Recall that 2-3-4 trees contain nodes with up to 3 data items/4 children per node.

Begin with an empty 2-3-4 tree. Insert the values listed below, in the order listed, **re-drawing the entire tree after EACH insertion**. (You may also draw an intermediate stage when node splitting is required, if you find that helpful.) Use a **top-down** 2-3-4 tree, where full nodes are split on the way down to the insertion point.

You may use Photoshop, Powerpoint, or other software to draw your trees, or you may draw your trees by hand on paper and scan or photograph your solution. Whatever method you choose, convert your solution into a pdf file. Your answer must be legible after conversion to pdf, and be well-organized so that it can be easily marked.

1. Begin with an empty 2-3-4 tree.
2. Insert 50.
3. Insert 20.
4. Insert 40.
5. Insert 45.
6. Insert 70.
7. Insert 60.
8. Insert 58.
9. Insert 65.
10. Insert 47.
11. Insert 90.
12. Insert 80.
13. Insert 100.
14. Insert 69.
15. Insert 67.
16. Insert 68.
17. Insert 95.

Your pdf file must be named `<your last name><your first name>A4Q2.pdf` (e.g. `SmithJohnA4Q2.pdf`). Submit your pdf file to the Assignment 4 dropbox in UM Learn, along with your solution to question 1.