Binary Expression Tree

This assignment will allow you to demonstrate your understanding of binary trees, in particular, the Binary Expression Tree.

Attached to this assignment is a solution named Tree. Start with this code. (Do not create a completely different class or find a different implementation on the Internet.) You will need to fill in the details; there are some guidelines and algorithms in the Binary Expression Tree tutorial that you might find useful, as well as the starter code. For parsing, you may use either the recursive algorithm or the while-loop; decide which one you understand best and use it.

I strongly encourage you to do these in order of ‘C’, ‘B’, ‘A’. Each part builds on the earlier portion. use the code to display infix to determine whether your parse is working. There is no reason to evaluate an expression until you know it is in your tree correctly.

## For a grade of ‘C’:

Write a program that will accept a fully parenthesized expression from the user. The program will then build a binary tree to hold the expression. Once the tree is built the program will display the expression in **prefix**, **infix** and **postfix** formats; e.g. an input of ((3\*(8-2))-(1+9)) would produce:

Infix: ((3\*(8-2))-(1+9))

Postfix: 382-\*19+-

Prefix: -\*3-82+19

Your program must support add, subtract, multiply, divide and ‘raise to the power of’ (represented with ‘^’). If you want to add other binary operators (e.g. %) feel free to do so.

Place your code in a loop so that the user can enter as many expressions as desired.

## For a grade of ‘B’:

Evaluate the expression and print its value, e.g.:

((3\*(8-2))-(1+9)) = 8;

You may use the recursive evaluate method that we have discussed, but if you would rather put the expression into a queue or stack (pulled from your tree, of course) and evaluate it pushing and popping the stack, that would be fine as well.

You could also use the evaluate algorithm run in a while-loop. Here you would also need a stack.

Note that evaluation should return a double. Even though all of the values in your expression will be single digit integers, the expression “(3/2)” is expected to evaluate to 1.5, not 1.

Evaluation of the ‘^’ symbol can best be implemented using Math.Pow().

## For a grade of ‘A’:

Add the ability for the expression to contain single character variables, e.g.:

(((x\*(8-2))-(y+9))-x)

After the expression is entered, prompt the user for the value of each variable. Unlike your expression, which can still only contain single digit integers, allow the values for the variables to be doubles so that the user can enter 3.7 for x for example. Use the value entered by the user to evaluate the expression. For example, given the above expression, prompt the user for the values of x and y. Assuming the response is ‘1’ and ‘2.2’ respectively, the expression program should print:

(((x\*(8-2))-(y+9))-x) = -6.2, where x=1.0 and y=2.2

Be nice to the user. Show the values of the variables in the output (like above or some other way you see fit). Also, only ask for ‘x’ once. It isn’t going to change as the expression is evaluated.

Continue to prompt the user for values for the variables until the user decides to quit. This means that given the above expression, you would prompt the user for new values for x and y and display the results. Give the user a way out of this inner loop, of course. Put this code in an outer loop which asks for a different expression.

Do not determine whether the expression has variables in your *Main* method. This must be done in your tree. This means that your tree should provide a ‘GetVariableList’ method that returns a data structure (List? Queue?) containing the variable names and that can be called from *Main*. You’ll want some sort of SetVariable(name, value) method to assign the value that the user inserts. I expect you to use a Dictionary for your variables. If you use some other structure, justify it in your comments.

Note, you **must** **not** ask for the variables during the Parse. Only Main should interact with the user. Loop through the variables asking the user for the values. Again, this should be in an internal loop. The user should be able to evaluate the same expression with different values until she decides to stop. Then prompt for a new expression.

## Questions:

Assuming the number of nodes is ‘n’:

* Roughly, what is the big-o for creating the tree?
* What is the big-o for traversing the tree:
  + in-order?
  + pre-order?
  + post-order?
* What is the big-o for evaluating the tree?

Consider the equivalent expressions (1+(2+(3+(4+(5+(6+(7+8)))))))) and (((1+2)+(3+4))+((5+6)+(7+8))).

* How do these trees differ?
* What are the heights of the root node in each version?
* How is the big-o affected for creating traversing and evaluating the tree?
* Take a look at the pre/post infix representations using your program. Any observations?