Programming Assignment #4

Binary Tree and Its Application

Educational Objectives: In this assignment you are to gain experience with the design and implementation of a binary tree and its application in converting postfix expressions into infix expressions, as well as practice with developing recursive algorithms.

Task: Implement a binary expression tree and use the tree to convert postfix expressions into infix expressions

Project Requirements:

In this project, you are asked to develop a binary expression tree and use the tree to convert postfix expressions into infix expressions. In this project, a postfix expression may contain 4 types of operators: multiplication (*), division (/), plus (+), and minus (-). We assume that multiplication and division have the same precedence, plus and minus have the same precedence, and multiplication and division have higher precedence than plus and minus. All operators are left-associative (i.e. associate left-to-right).

Binary Expression Tree:. Build a binary expression tree class called "BET". Your BET class must have a nested structure named "BinaryNode" to contain the node-related information (including, e.g., element and pointers to the children nodes). In addition, BET must at least support the following interface functions (you may have more in your implementation but do not change the test driver):

Public interface

- **BET()**: default zero-parameter constructor. Builds an empty tree.
- **BET(const string& postfix)**: one-parameter constructor, where parameter "postfix" is string containing a postfix expression. The tree should be built based on the postfix expression. Tokens in the postfix expression are separated by spaces.
- **BET(const BET&)**: copy constructor -- makes appropriate deep copy of the tree
- **~BET()**: destructor -- cleans up all dynamic space in the tree

- **bool buildFromPostfix(const string& postfix)**: parameter "postfix" is string containing a postfix expression. The tree should be built based on the postfix expression. Tokens in the postfix expression are separated by spaces. If the tree contains nodes before the function is called, you need to first delete the existing nodes. Return true if the new tree is built successfully. Return false if an error is encountered.
- **const BET & operator= (const BET &)**: assignment operator -- makes appropriate deep copy
- **void printInfixExpression()**: Print out the infix expression. Should do this by making use of the private (recursive) version
- **void printPostfixExpression()**: Print the postfix form of the expression. Use the private recursive function to help
- **size_t size()**: Return the number of nodes in the tree (using the private recursive function)
- **size_t leaf_nodes()**: Return the number of leaf nodes in the tree. (Use the private recursive function to help)
- **bool empty()**: return true if the tree is empty. Return false otherwise.

Private helper functions (all the required private member functions must be implemented recursively):

- **void printInfixExpression(BinaryNode *n)**: print to the standard output the corresponding infix expression. Note that you may need to add parentheses depending on the precedence of operators. You should not have unnecessary parentheses.
- **void makeEmpty(BinaryNode* &t)**: delete all nodes in the subtree pointed to by t.
- **BinaryNode * clone(BinaryNode *t)**: clone all nodes in the subtree pointed to by t. Can be called by functions such as the assignment operator=.
- **void printPostfixExpression(BinaryNode *n):** print to the standard output the corresponding postfix expression.
- **size_t size(BinaryNode *t)**: return the number of nodes in the subtree pointed to by t.
- **size_t leaf_nodes(BinaryNode *t)**: return the number of leaf nodes in the subtree pointed to by t.

Make sure to declare as `const` member functions any for which this is appropriate

Conversion to Infix Expression:. To convert a postfix expression into an infix expression using a binary expression tree involves two steps. First, build a binary expression tree from the postfix expression. Second, print the nodes of the binary expression tree using an in-order traversal of the tree.

The basic operation of building a binary expression tree from a postfix expression is similar to that of evaluating postfix expression. Refer to Section 4.2.2 for the basic process of building a binary expression tree from a postfix expression.

Note that during the conversion from postfix to infix expression, parentheses may need to be added to ensure that the infix expression has the same value (and the same evaluation order) as the corresponding postfix expression. Your result should not add unnecessary parentheses. Tokens in an infix expression should also be separated by a space. The following are a few examples of postfix expressions and the corresponding infix expressions.

postfix expression	infix expression
4 50 6 + +	4 + (50 + 6)
4 50 + 6 +	4 + 50 + 6
4 50 + 6 2 * +	4 + 50 + 6 * 2
4 50 6 + + 2 *	(4 + (50 + 6)) * 2
a b + c d e + * *	(a + b) * (c * (d + e))

Other Requirements:

- Analyze the worst-case time complexity of the private member function **makeEmpty(BinaryNode* & t)** of the binary expression tree. Give the complexity in the form of Big-O. Your analysis can be informal; however, it must be clearly understandable by others. Name the file containing the complexity analysis as "analysis.txt".
- You can use any C++/STL containers and algorithms
- If you need to use any containers, you must use the ones provided in C++/STL. **<u>Do not use the ones you developed in the previous projects.</u>**
- Create a makefile that will compile the provided driver program (see below) with your class, into an executable called "proj4.x"

- Your program MUST check invalid postfix expressions and report errors. We consider the following types of postfix expressions as invalid expressions: 1) an operator does not have the corresponding operands, 2) an operand does not have the corresponding operator. Note that an expression containing only a single operand is a valid expression (for example, "6"). In all other cases, an operand needs to have an operator.

The sample driver program (proj4_driver.cpp) is attached to this assignment and included in Canvas. The driver accepts input from terminal, or the input is redirected from a file that contains the postfix expressions to be converted. Each line in the file (or typed by user) represents a postfix expression. We assume that the tokens in a postfix expression are separated by space.

Submitting

Tar up all of your C++ source files and header files that you develop for this project, as well as the makefile and the analysis.txt file into one tar archive, and submit online via Canvas,in the "Assignments" section. Use the Assignment 4 link to submit. Make sure to tar your files correctly.

Your tar file should be named in this format, all lowercase:

*lastname_firstname*_p4.tar

Example: My tar file would be: **Gaitros_david_p4.tar**