Project 4: Binary Tree and Its Application

Due: 04/05/2018

Educational Objectives: Experience with binary tree and its application in converting postfix expressions into infix expressions, experience in developing recursive algorithms.

Statement of Work: 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, and plus and minus have the same precedence. Moreover, multiplication and division have higher precedence than plus and minus. All operators are left associative.

Binary Expression Tree:. Name your binary expression tree class as "BET". Your BET must have a nested structure named "BinaryNode" to contain the node related information (including, e.g., element and pointers to children nodes). In addition, BET must at least support the following interfaces.

Public interfaces

- **BET()**: default zero-parameter constructor.
- **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 space.
- **BET(const BET&)**: copy constructor.
- **~BET()**: destructor.
- **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 space. 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.
- **void printInfixExpression()**: call the private version of the printInfixExpression function to print out the infix expression.
- **void printPostfixExpression()**: call the private version of the printPostfixExpression function to print out the postfix expression.
- **size_t size()**: call the private version of the size function to return the number of nodes in the tree.
- **size_t leaf_nodes()**: call the private version of the leaf_nodes function to return the number of leaf nodes in the tree.
- **bool empty()**: return true if the tree is empty. Return false otherwise.

Private interfaces **(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. Called by functions such as the destructor.
- **BinaryNode * clone(BinaryNode *t) const**: clone all nodes in the subtree pointed to by t. 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.

Conversion to Infix Expression:. To convert a postfix expression into an infix expression using binary expression tree involves two steps. First, you need to build a binary expression tree from the postfix expression. Second, you need to print the nodes of the binary expression tree using inorder 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. They all involve the use of stack to hold intermediate results. Essentially, when you encounter an operand, you create a node to contain the operand and push it into a stack. When you encounter an operator, you pop out the corresponding operands from the stack, and build a new tree, and then push the new tree into the stack. After you have processed all tokens in the postfix expression, the stack has the binary expression tree. Please refer to Section 4.2.2 for building binary expression from 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. You cannot 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. You cannot use the ones you developed in the previous projects.

- 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.

A driver program to test the BET implementation is provided to you. It 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.

Provided Code

The tar file contains the following files:

1. proj4_driver.cpp: the driver program.
2. proj4.x: executable code (compiled on linprog.cs.fsu.edu)

Deliverable Requirements

Turn in the tar file containing all c++ source files and header files that you may develop for this project, the makefile, , and the analysis.txt online at Canvas.