Binary Expression Tree Project

Project Objectives

To create a `binaryExpressionTree` class which:

- Inherits from the `binaryTreeType` class.
- Correctly handles the fact that the base class is an abstract class.
- Uses the instance variables from the base class.
- Implements the `evaluateExpressionTree` recursively but does not provides access to the interior nodes of the expression tree.

Report Instructions

- Read the problem requirements in the project instructions section below.
- Plan the project.
- Build your project in the IDE of your choice. When completed and ready to turn in, store your project in a GitHub repository. Put the URL to the repository and the name of the IDE you used for development at the beginning of your report.
- When your project is ready to be graded write a project report with the following format:
  - URL to GitHub repository
  - Name of IDE used for development
  - Name of program (title, number, or both)
  - Your name
  - Name of your partner (if you have one)
  - Planning notes – UML diagrams, IPO charts for member functions, handwritten notes, etc.
  - Answer the following reflection questions:
    1. What did you find most challenging with this program?
    2. What problems did you encounter and how did you solve them?
    3. What did you learn from writing this program?

Your project report must be in .pdf format. Upload your project report to the assignment in Canvas and submit.

Project Instructions

In the module on stacks, we looked at a program which used a stack to evaluate a postfix expression. For this project we will use a type of binary tree called an **expression tree** rather than a stack to perform the evaluation.

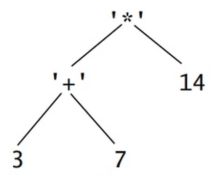
In a simple expression tree, there are two kinds of nodes:

(a) Leaf nodes, which contain the operands for the info field;

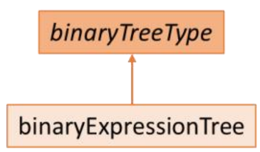
(b) Non-leaf nodes, which contain the operator and have exactly two child nodes (one for each operand).

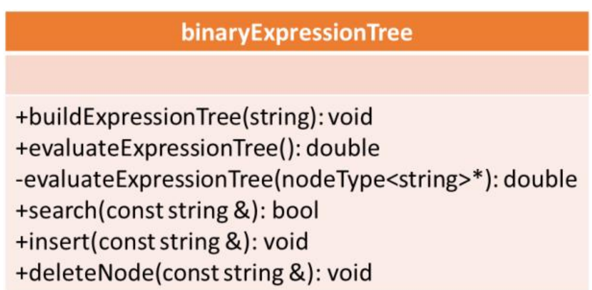
For example, the postfix expression 3 7 + 14 * evaluates to 140 ((3 + 7) * 14).

Here is what the expression tree for the postfix expression above would look like:



For this project, create a `binaryExpressionTree` class which inherits from the `binaryTreeType` class and has a function to take a postfix expression as an input and build an expression tree and another function which evaluates the expression tree and returns the result. Here is the UML diagram for this class:





Here are things to note:

The `binaryExpressionTree` class inherits from the `binaryTreeType` class which is a template. However, the info fields of the nodes in the `binaryExpressionTree` class will be strings. The `Type` parameter for the `binaryTreeType` template can be fixed as a `string`, so that the `binaryExpressionTree` class is not a template.

The `binaryTreeType` class is abstract. Therefore, the `binaryExpressionTree` class must implement the pure virtual functions `search`, `insert`, and `deleteNode`. These functions do not need to do anything, but they must be implemented.

The `expressionTreeType` class does not have any instance variables. It will store the pointer to the expression tree in the `binaryTreeType` instance variable, `root`. This variable is protected so that the `binaryExpressionTree` class can access it directly.

The `nodeType struct` is also a template, so specify the `<string>` data type anywhere where it is being used (for example, the parameter to private version of `evaluateExpressionTree` is a pointer to a `nodeType<string> struct`).

The `evaluateExpressionTree` function is recursive and is overloaded so that the public version does not take any parameters. It simply calls the private version and passes it the root of the binary expression tree.

This class supports postfix expressions containing only the operators `+`, `–`, `/`, `*`. The postfix expressions are expected to be well-formed.

The `evaluateExpressionTree` function requires a stack. For this project, use the library stack class. The header to include is `<stack>`. It has `push()`, `pop()`, and `top()` functions and uses a linked list so that it only has an `empty()` function. This class is a template, so the `Type` parameter should be specified as a pointer to a `nodeType<string> struct`.

To parse the postfix expression string, use the tokenizer function that was demonstrated in the Postfix Expression Calculator example in the module on stacks.

Here is the algorithm for building the expression tree:

Initialize a stack of pointers to binary tree nodes
Get a postfix expression (*this will be an input to the* `evaluateExpressionTree` *function*)
Convert the string to a character array (*include* `<cstring>`)

```
//Convert postfixExpress to a cstring, expression
char * expression = new char [postfixExpression.length()+1];
strcpy (expression, postfixExpression.c_str());
```

For each token in the expression:
- If token is a number
  - Create a node
  - Convert the token from a character array to a string
  - Store the string in the info field
  - Push the new node onto the stack
- else if token is an operator
  - Create a node and store the operator in the info field
  - If stack is not empty
    - Use `top()` to get a reference to the node at the top of the stack
    - Store the reference in the `rLink` field of the new node.
    - Use `pop()` to remove the node from the stack
    - If stack is not empty
      - Use `top()` to get a reference to the node at the top of the stack
      - Store the reference in the `lLink` field of the new node
      - Use `pop()` to remove the node from the stack
      - Push the new node back onto the stack
    - else
      - Error – Stack is empty
  - else
    - Error – Stack is empty

else
Error – unsupported token
Get the next token
if stack is not empty
Pop the expression tree from the stack and store in root
If stack is not empty
There was an error, set root back to null

Note that you can use the postorder traversal from the base class to view the expression tree after you build it.

The algorithm to evaluate the expression tree is recursive:

(a) If the tree has only one node (which must be a leaf), then return the info field of the node (this is the base case).

(b) If the tree has more than one node and the root contains an operator, first evaluate the left subtree (recursive call), then evaluate the right subtree (recursive call) and apply the operator to the results.

In the example above, the left subtree evaluates to 3+7, which is 10. The right subtree evaluates to 14. So, the entire tree evaluates to 10*14, which is 140.

Algorithm for evaluating the expression tree:

(*p is a pointer to a node*
x is the left operand of an expression
y is the right operand of an expression
op is the operator
The function, evaluate(p) is recursive)
if p is a leaf then
return the value stored at p (*value is stored as a string, convert to double using* `stod()` *before returning*)
else
op is the info field of p
x = evaluate(left(p))
y = evaluate(right(p))
Evaluate the expression x op y and return the result

For the main function to test the class, read the postfix expressions from a file and write the results to a file using the Postfix Expressions Calculator example from the module on stacks as a guide. Here are some notes about this program:

Since the postfix expressions are passed to the `evaluateExpressionTree` function as strings, read them using `getline()`. The `evaluateExpressionTree` function should take care of converting the string to a cstring for the tokenizer.

Inside the loop where you are reading the strings from the input file, call the `destroyTree()` function from `binaryTreeType` to clear out the current expression tree. Do this before you read the next string.

The input file contains only well-formed postfix expressions. The mal-formed expressions have been removed.