Lab 6: Arithmetic Expression Tree

In this lab you will complete a program that builds and traverses an Arithmetic Expression Tree, then outputs the resulting tree using the drawing software Graphviz (short for Graph Visualization Software).

ArithmeticExpression class:

```
#ifndef ARITHMETICEXPRESSION_H
#define ARITHMETICEXPRESSION_H
#include <iostream>
#include <cstdlib>

using namespace std;
struct TreeNode{
    char data;
    char key;
    TreeNode* left;
    TreeNode* right;
    TreeNode(char data, char key):data(data),key(key),left
(0),right(0){}
};

class arithmeticExpression{
  private:
    string infixExpression;
    TreeNode* root;

  public:
    /* Initializes an empty tree and sets the infixExpression
    to the value of parameter passed in. */
    arithmeticExpression(const string &);
    /* Implementation of destrucor is optional.
       The destructor should deallocate all the nodes in the
tree. */
    //~arithmeticExpression();

    /* Converts the infixExpression to its equivalent postfix
string
       and then generates the tree and assigns the root node
to the
       root data field. The key for the first node is 'a',
'b' for the second node and so on. */
    void buildTree();

    /* Calls the recursive infix function. */
    void infix();

    /* Calls the recursive prefix function. */
    void prefix();

    /* Calls the recursive postfix function. */
    void postfix();

    /* Calls the recursive visualizeTree function and generat
es the .png file using system call. */
    void visualizeTree(const string &);

 private:
    /* Helper function that returns an integer according to
       the priority of the given operator. */
    int priority(char);

    /* Helper function that returns the postfix notation equi
valent
       to the given infix expression */
    string infix_to_postfix();
    /* Helper function that outputs the infix notation of the
arithmetic expression tree
       by performing the inorder traversal of the tree.
       An opening and closing parenthesis must be added at th
e
       beginning and ending of each expression. */
    void infix(TreeNode *);

    /* Helper function that outputs the prefix notation of th
e arithmetic expression tree
       by performing the preorder traversal of the tree. */
    void prefix(TreeNode *);

    /* Helper function that outputs the postfix notation of t
he arithmetic expression tree
       by performing the postorder traversal of the tree. */
    void postfix(TreeNode *);

    /* Helper function for generating the dotty file. This is
a recursive function. */
    void visualizeTree(ofstream &, TreeNode *);
};
#endif
```

Here is the implementation of the functions `priority`, `infix_to_postfix` and `visualizeTree`. You should implement the rest of functions.

```
int arithmeticExpression::priority(char op){
    int priority = 0;
    if(op == '('){
        priority =  3;
    }
    else if(op == '*' || op == '/'){
        priority = 2;
    }
    else if(op == '+' || op == '-'){
        priority = 1;
    }
    return priority;
}

string arithmeticExpression::infix_to_postfix(){
    stack<char> s;
    ostringstream oss;
    char c;
    for(unsigned i = 0; i< infixExpression.size();++i){
        c = infixExpression.at(i);
        if(c == ' '){
            continue;
        }
        if(c == '+' || c == '-' || c == '*' || c == '/' || c
== '(' || c == ')'){ //c is an operator
            if( c == '('){
                s.push(c);
            }
            else if(c == ')'){
                while(s.top() != '('){
                    oss << s.top();
                    s.pop();
                }
                s.pop();
            }
            else{
                while(!s.empty() && priority(c) <= priority
(s.top())){
                    if(s.top() == '('){
                        break;
                    }
                    oss << s.top();
                    s.pop();
                }
                s.push(c);
            }
        }
        else{ //c is an operand
            oss << c;
        }
    }
    while(!s.empty()){
        oss << s.top();
        s.pop();
    }
    return oss.str();
}

void arithmeticExpression::visualizeTree(const string &output
Filename){
    ofstream outFS(outputFilename.c_str());
    if(!outFS.is_open()){
        cout<<"Error opening "<< outputFilename<<endl;
        return;
    }
    outFS<<"digraph G {"<<endl;
    visualizeTree(outFS,root);
    outFS<<"}";
    outFS.close();
    string jpgFilename = outputFilename.substr(0,outputFilena
me.size()-4)+".jpg";
    string command = "dot -Tjpg " + outputFilename + " -o " +
jpgFilename;
    system(command.c_str());
}
```

main.cpp test harness

Use this main.cpp file for testing your program.

```
#include <iostream>
#include "arithmeticExpression.h"

using namespace std;

int main(){
    string expr1 = "a+b*c";
    arithmeticExpression ex1(expr1);
    ex1.buildTree();
    cout<<"expression 1: "<<expr1<<endl;
    cout<<"infix: "; ex1.infix(); cout<<endl;
    cout<<"prefix: "; ex1.prefix(); cout<<endl;
    cout<<"postfix: "; ex1.postfix(); cout<<endl;
    //ex1.visualizeTree("expr1.dot");
    cout<<endl;

    string expr2 = "(a+b)*(c-d)";
    arithmeticExpression ex2(expr2);
    ex2.buildTree();
    cout<<"expression 2: "<<expr2<<endl;
    cout<<"infix: "; ex2.infix(); cout<<endl;
    cout<<"prefix: "; ex2.prefix(); cout<<endl;
    cout<<"postfix: "; ex2.postfix(); cout<<endl;
    //ex2.visualizeTree("expr2.dot");
    cout<<endl;

    string expr3 = "a + b * c - ( d * e + f ) * g";
    arithmeticExpression ex3(expr3);
    ex3.buildTree();
    cout<<"expression 3: "<<expr3<<endl;
    cout<<"infix: "; ex3.infix(); cout<<endl;
    cout<<"prefix: "; ex3.prefix(); cout<<endl;
    cout<<"postfix: "; ex3.postfix(); cout<<endl;
    //ex3.visualizeTree("expr3.dot");
    cout<<endl;

    return 0;
}
```

Graphviz

`NOTE`: each TreeNode contains a "key" value which is a unique character used to generate the .dot file.

`Graphviz Dot files` - graphs represented in .dot file can be displayed int the Graphviz viewer called dotty. We will call dotty of a .dot file from within your C++ program using the system call. But first familiarize yourself with both the input file format and the corresponding output using the dotty viewer.

`Installation`

To install graphviz package you may open up a terminal and execute the following commands:

1. sudo apt-get update
2. sudo apt-get install graphviz

`Input file`

Here are three input file samples:

`1. simple_bst.dot:` This is the most basic method for producing a tree using Graphviz. The identifier for each node is that node's label, for example "key25" represents that the node has a key value of 25.

```
digraph G1 {
  // nodes and edges
  key12 -> key8;
  key12 -> key16;
  key8  -> key1;
  key8  -> key11;
  key16 -> key13;
  key16 -> key25;
}
```

Line 1 gives the graph name (G1) and type (digraph). The lines that follow creates nodes, edges, or subgraphs and set attributes. You may use Quotes to protect punctuation and white spaces. A node is created when its name first appears in the file (e.g. key12). An edge is created when nodes are joined by the edge operator `->` . To visualize the above graph, copy and paste the above code into a new file and save it as "simple_bst.dot" file and then run the following command:

```
dot -Tpng simple_bst.dot -o graph.jpg
```

Once you execute the above command into a terminal, a visualization of the graph is going to be created into an image called graph.jpg.

`2. height_color.dot:` The following code contains the representation for a tree (called G2) with nodes' labels that indicate the node's key and the node's height in the tree.

```
digraph G2 {

  // nodes
  a [color = lightblue, style = filled, label="key=4, h=2"];
  b [color = lightblue, style = filled, label="key=2, h=1"];
  c [color = lightblue, style = filled, label="key=6, h=1"];
  d [color = lightblue, style = filled, label="key=1, h=0"];
  e [color = lightblue, style = filled, label="key=3, h=0"];
  f [color = lightblue, style = filled, label="key=5, h=0"];
  g [color = lightblue, style = filled, label="key=7, h=0"];

  // edges
  a -> b;
  a -> c;
  b -> d;
  b -> e;
  c -> f;
  c -> g;
}
```

`3. inorder_color.dot:` The following code is the representation for G3 tree with nodes' labels that indicate the node's key and the node's order in which it is visited in an inorder traversal of the tree.

```
digraph G3 {

  // nodes
  a [color = red, peripheries=2, style = filled, label="key= 4, in=4"];
  b [color = red, peripheries=2, style = filled, label="key= 2, in=2"];
  c [color = red, peripheries=2, style = filled, label="key= 6, in=6"];
  d [color = red, peripheries=2, style = filled, label="key= 1, in=1"];
  e [color = red, peripheries=2, style = filled, label="key= 3, in=3"];
  f [color = red, peripheries=2, style = filled, label="key= 5, in=5"];
  g [color = red, peripheries=2, style = filled, label="key= 7, in=7"];

  // edges
  a -> b;
  a -> c;
  b -> d;
  b -> e;
  c -> f;
  c -> g;
}
```