PROJECT 11

IMPLEMENTING AND MANIPULATING EXPRESSION TREES

Course: CSCI 301

Name: Ziqi Zhao

Section Number: 1

Project Number: 11

Due Date: Wednesday, May 1

1. **Design Document**

**Introduction:**

This program is using a binary tree to compute expressions. We can call it expression tree. An expression tree represents an arithmetic expression interior nodes hold operators, leaves hold operands. We can evaluate the expression such a tree represents, and print the expression in prefix, infix, postfix notation. In this project, it can read a file which contains some contents of expressions with fully- parenthesized. Implement these contents to an expression tree and compute the value with these expressions. I create a class named Exptree, it’s a pointer-based binary tree.

**Data Structure:**

This program used a class named Exptree to implement a pointer-based binary tree to represent the expressions. Struct the Node in the project. Contain int operand, char optr, Node \* left, Node \* right and Node \* root.

Use fstream to open and read a file.

**Function:**

Exptree( ); Constructor function, initialized empty.

~Exptree( ); Destructor, delete node.

void build(istream & in\_f); Build the tree corresponding to a fully-parenthesized infix expression from the file.

int value( ); Evaluate the expression such a tree represents.

void prefix( ); Preorder traversal and print expression of the tree.

void postfix( ); Postfix traversal and print expression of the tree.

void infix( ); Inorder traversal and print expression of the tree.

void destory(Node \* p); Private destructor function. Base case: the tree is empty; Smaller solution: delete left subtree first, then delete right subtree, finally delete the root.

Node \* r\_build(istream & in\_f); Private build node function, called by the public build function. Base case: only one operand in the expression, build it as a node and set the left and right pointers to NULL.

int r\_value(Node \* r); Private evaluate function. Base case: r only have an operand; Smaller solution: evaluate the value of a subtree.

void r\_prefix(Node \* r); Preorder traversal of the tree, print root first, then left node, finally right node.

void r\_postfix(Node \* r); Postorder traversal of the tree, print left node first, then right node, finally root.

void r\_infix(Node \* r); Inorder traversal of the tree, print left node first, then root, finally right node.

**The main program:** I create an expression tree, a fstream and a string as parameters to call the functions. I also use ifstream to open and read the file.

1. **Code**

===============================================================

// Name : expTree.cpp

// Author : Ziqi Zhao

// Course : CSCI 301

// Section : 1

// Project : 11

// Due Date : Wednesday, May 1

// Description : This is a program used a pointer-based binary tree to

// implement and exercise expressions. Expressions represented in a binary

// tree are easy to compute recursively and when we print expressions, they

// can be represented to preorder, inorder and postorder. We also can

// compute the value of expression recursively.

//===============================================================

#include <iostream>

#include <fstream>

#include <stdlib.h>

#include <cstring>

#include <string.h>

using namespace std;

class Exptree {

public:

//Constructor, initialized empty.

Exptree() {root = NULL;}

//Destructor.

~Exptree();

//Build the tree corresponding to a fully parenthesized

//infix expression from the file stream.

void build(istream & in\_f);

//Evaluate the expression such a tree represents.

int value();

//Preorder traversal and print expression of the tree.

void prefix();

//Postorder traversal and print expression of the tree.

void postfix();

//Inorder traversa and print expression of the tree.

void infix();

private:

//Declartions.

struct Node {

int operand;

char optr;

Node \* left;

Node \* right;

};

Node \* root;

//private destructor function, delete nodes of the tree

//base case: the tree is empty

//smaller solution: delete left subtree, then right subtree, finally root

void destroy(Node \* r);

//private build node function, called by th public one

//base case: only one operand in the tree, just build it as a node,

//left and right pointers are NULL;

Node \* r\_build(istream & in\_f);

//private evaluate function

//precondition: the parameter r is not NULL

//base case: r only have operand

int r\_value(Node \* r);

//print the expression in preorder

void r\_prefix(Node \* r);

//print the expression in postorder

void r\_postfix(Node \* r);

//print the expression in inorder

void r\_infix(Node \* r);

};

Exptree::~Exptree() {

destroy(root);

}

void Exptree::destroy(Node \* r) {

if (r != NULL) {

destroy(r->left);

destroy(r->right);

delete r;

}

}

void Exptree::build(istream & in\_f) {

destroy(root);

root = r\_build(in\_f);

}

Exptree::Node \* Exptree::r\_build(istream & in\_f) {

char ch;

Node \* temp;

in\_f >> ch;

if (isdigit(ch)) {

temp = new Node;

temp->operand = ch - '0';

temp->left = NULL;

temp->right = NULL;

}

else {

temp = new Node;

temp->left = r\_build(in\_f);

in\_f >> temp->optr;

temp->right = r\_build(in\_f);

in\_f >> ch;

}

return temp;

}

int Exptree::value(){

return r\_value(root);

}

int Exptree::r\_value(Node \* r) {

if (r->left == NULL) {

return r->operand;

}

else {

int l\_val = r\_value(r->left);

int r\_val = r\_value(r->right);

if (r->optr == '+')

return l\_val + r\_val;

else if (r->optr == '-')

return l\_val - r\_val;

else if (r->optr == '\*')

return l\_val \* r\_val;

else

return l\_val / r\_val;

}

}

void Exptree::prefix() {

r\_prefix(root);

}

void Exptree::r\_prefix(Node \* r) {

if (r->left == NULL) {

cout << r->operand <<' ';

}

else {

cout << r->optr << ' ';

r\_prefix(r->left);

r\_prefix(r->right);

}

}

void Exptree::postfix() {

r\_postfix(root);

}

void Exptree::r\_postfix(Node \* r) {

if (r->left == NULL) {

cout << r->operand << ' ';

}

else {

r\_postfix(r->left);

r\_postfix(r->right);

cout << r->optr << ' ';

}

}

void Exptree::infix() {

r\_infix(root);

}

void Exptree::r\_infix(Node \* r) {

if (r->left == NULL) {

cout << r->operand << ' ';

}

else {

cout << " ( ";

r\_infix(r->left);

cout << r->optr <<' ';

r\_infix(r->right);

cout << " ) ";

}

}

int main() {

Exptree \* tree = new Exptree();

fstream file;

string name;

char buffer[256];

cout << "Enter input file name: ";

cin >> name;

cout << endl;

file.open(name.c\_str(), ios::in);

while (!file.eof()) {

file.getline(buffer,256, '\n');

tree->build(file);

cout << "Preorder: ";

tree->prefix();

cout << endl;

cout << "Inorder: ";

tree->infix();

cout << endl;

cout << "Postorder: ";

tree->postfix();

cout << endl;

cout << "Value = " << tree->value() << endl;

cout << endl;

}

file.close();

return 0;

}

1. **User Document**

This program named **expTree.cpp** is used to evaluate expressions. The user can compute the value about the expressions in this file and print them in different three orders. I also create a file named **data.txt and data1.txt**, they contain some expressions.

The program’s name is expTree.cpp

To compile the program, simply enter:

g++ expTree.cpp

To run it, enter:

a.out

Please enter your input after the program prompts the command line:

Enter input file name:

Then you should type the file’s name-data.txt and data1.txt. Then you can get your results.

1. **Tests**

**Test1:**

csci2>a.out

Enter input file name: data.txt

Preorder: 3

Inorder: 3

Postorder: 3

Value = 3

Preorder: + + \* 2 5 - 9 1 + 7 / 6 3

Inorder: ( ( ( 2 \* 5 ) + ( 9 - 1 ) ) + ( 7 + ( 6 / 3 ) ) )

Postorder: 2 5 \* 9 1 - + 7 6 3 / + +

Value = 27

Preorder: / 9 3

Inorder: ( 9 / 3 )

Postorder: 9 3 /

Value = 3

Preorder: - 7 6

Inorder: ( 7 - 6 )

Postorder: 7 6 -

Value = 1

Preorder: \* + - \* 5 3 / 4 2 / 3 1 7

Inorder: ( ( ( ( 5 \* 3 ) - ( 4 / 2 ) ) + ( 3 / 1 ) ) \* 7 )

Postorder: 5 3 \* 4 2 / - 3 1 / + 7 \*

Value = 112

Preorder: + 1 1

Inorder: ( 1 + 1 )

Postorder: 1 1 +

Value = 2

**Test2:**

csci2>a.out

Enter input file name: data1.txt

Preorder: + 0 9

Inorder: ( 0 + 9 )

Postorder: 0 9 +

Value = 9

Preorder: \* 3 7

Inorder: ( 3 \* 7 )

Postorder: 3 7 \*

Value = 21

Preorder: + + 7 3 / \* 2 9 + 2 1

Inorder: ( ( 7 + 3 ) + ( ( 2 \* 9 ) / ( 2 + 1 ) ) )

Postorder: 7 3 + 2 9 \* 2 1 + / +

Value = 16

Preorder: \* + / 4 2 - 9 5 + 7 \* 6 8

Inorder: ( ( ( 4 / 2 ) + ( 9 - 5 ) ) \* ( 7 + ( 6 \* 8 ) ) )

Postorder: 4 2 / 9 5 - + 7 6 8 \* + \*

Value = 330

csci2>

1. **Summary**

From this program, I learn how to use binary tree to implement expressions. We can call it expression tree. An expression tree is a binary tree in which interior nodes hold operators and leaves hold operands. When we read a file from computer, the program can build a tree for an expression, then compute the value of the expression and print this expression in different orders-prefix, infix, postfix.

Binary Tree is a useful ADT, almost all functions can be recursive computation, that makes our code look more concise and clear.