**Lab 04**

**Name:** Will Townsend

**Class:** COSC 320 – Section 751

**Date:** 09/27/2020

**Lab Report:**

This lab took a little more thought in applying the different traversals due to the way the provided functions were implemented. This lab took me around 1 hour and 30 minutes (30 minutes to implement the code and 1 hour to debug). I can confirm I did this lab 100% independently without help from outside sources or classmates.

**Lab:**

**Task 1:**

1.1) a\*b

\*

a b

1.2) a+b\*c

\*

+ c

a b

1.3) a+b\*c/d-e

\*

+ -

a b / e

c d

**Task 2:**

2.1:

Pre: \*ab

Post: ab\*

2.2:

Pre: \*+abc

Post: ab+c\*

2.3:

Pre: \*+ab-/cde

Post: ab+cd/e-\*

Answer: The postfix outputs always have the root at the end of the output and prefix outputs have the root at the beginning. The post will always have the operands before the operator in a level above that operand and pre will always have the operator before the operand when above those operands (or on the identical level in some cases).

**Task 3:**

One of the problems that I didn’t see with my original scan of the infix2postfix class is the function postfix’s parameter. I didn’t realize at first that in order to work it needed to look like an equation instead of the infix ordering. I realized it after looking through it a second time and fix the parameters in my main accordingly.

**Task 4:**

The way this was implemented made it so you had to put in the tree in its postfix order. Although, the comments never explicitly stated when doing it the other ways it would return a segfault because there was nothing to pop off the stack s of subtrees.

**Task 5:**

void prefixoutput(tnode<char> \*exp){

if (exp){

printf(“%c ”, exp->nodeValue);

prefixoutput(exp->left);

prefixoutput(exp->right);

}

}

**Task 6:**

lab\_04.cpp:

#include<stdio.h>

#include"d\_except.h"

#include"d\_expsym.h"

#include"d\_tnodel.h"

#include"d\_tnode.h"

#include"inf2pstf.h"

void prefixoutput(tnode<char> \*exp);

tnode<char> \*buildExpTree(const string& exp);

int main(){

puts("Lab 04: Tree Traversal");

std::string tree1="ab+cd-ef\*+/";

std::string tree2="hi/j-kl\*m/+";

std::string tree3="vw-x/yz+-";

tnode<char>\* one = buildExpTree(tree1);

tnode<char>\* two = buildExpTree(tree2);

tnode<char>\* three = buildExpTree(tree3);

infix2Postfix ex1 = infix2Postfix("(a+b)/((c-d)+(e\*f))");

infix2Postfix ex2 = infix2Postfix("((h/i)-j)+((k\*l)/m)");

infix2Postfix ex3 = infix2Postfix("((v-w)/x)-(y+z)");

puts("Example 1:");

displayTree(one,3);

std::cout<<"Prefix: ";

prefixoutput(one);

std::cout<<"\nPostfix: "<< ex1.postfix()<<std::endl;

puts("\nExample 2:");

displayTree(two,3);

std::cout<<"Prefix: ";

prefixoutput(two);

std::cout<<"\nPostfix: "<< ex2.postfix()<<std::endl;

puts("\nExample 3:");

displayTree(three,3);

std::cout<<"Prefix: ";

prefixoutput(three);

std::cout<<"\nPostfix: "<< ex3.postfix()<<std::endl;

return 0;

}

void prefixoutput(tnode<char> \*exp){

if(exp){

printf("%c ",exp->nodeValue);

prefixoutput(exp->left);

prefixoutput(exp->right);

}

}

tnode<char> \*buildExpTree(const string& exp)

{

// newnode is the address of the root of subtrees we build

tnode<char> \*newnode, \*lptr, \*rptr;

char token;

// subtrees go into and off the stack

stack<tnode<char> \*> s;

int i = 0;

// loop until i reaches the end of the string

while(i != exp.length())

{

// skip blanks and tabs in the expression

while (exp[i] == ' ' || exp[i] == '\t')

i++;

// if the expression has trailing whitespace, we could

// be at the end of the string

if (i == exp.length())

break;

// extract the current token and increment i

token = exp[i];

i++;

// see if the token is an operator or an operand

if (token == '+' || token == '-' || token == '\*' || token == '/')

{

// current token is an operator. pop two subtrees off

// the stack.

rptr = s.top();

s.pop();

lptr = s.top();

s.pop();

// create a new subtree with token as root and subtees

// lptr and rptr and push it onto the stack

newnode = new tnode<char>(token,lptr,rptr);

s.push(newnode);

}

else // must be an operand

{

// create a leaf node and push it onto the stack

newnode = new tnode<char>(token);

s.push(newnode);

}

}

// if the expression was not empty, the root of the expression

// tree is on the top of the stack

if (!s.empty())

return s.top();

else

return NULL;

}

Output:

Lab 04: Tree Traversal

Example 1:

/

+ +

a b - \*

c d e f

Prefix: / + a b + - c d \* e f

Postfix: a b + c d - e f \* + /

Example 2:

+

- /

/ j \* m

h i k l

Prefix: + - / h i j / \* k l m

Postfix: h i / j - k l \* m / +

Example 3:

-

/ +

- x y z

v w

Prefix: - / - v w x + y z

Postfix: v w - x / y z + -