|  |  |  |
| --- | --- | --- |
| American University of Sharjah  College of Engineering  Department of Computer Science & Engineering  P. O. Box 26666, Sharjah, UAE |  | **Instructors:** Dr. Michel Pasquier  **Lab Instructor:** Praveena Kolli  **Office:** EB2-12  **Phone**: 971-6-5152352  **e-mail**: pkolli@aus.edu  **Semester**: Spring 2021 |

**CMP305L - Data Structures and Algorithms Lab**

**Lab. Assignment 10 – Trees and Binary Trees**

***Objectives***

* To understand Trees and Binary Trees
* To implement recursive traversals on trees

***Instructions***

* Do not use any static or global variable. Use recursion!

***Note:***

***Lab:*** Exercises 1 and 2 (10 marks)

***Bonus*:** Exercise 3 (1 mark)

***Exercise 1:***

Write the following recursive functions that work on a general tree. The *TreeNode* structure is provided to you.

1. Depth of a node: that is the number of edges from the tree's root node to that node.
2. Height of a node: that is the number of edges on the longest downward path from that node to a leaf node.

You may use the provided main() function to continue creating nodes and the tree structure so as to test the above functions.

Code:

#include<iostream>

using namespace std;

template <typename Object>

struct TreeNode

{

Object element;

TreeNode \*firstChild;

TreeNode \*nextSibling;

TreeNode(const Object & e = Object{ }, TreeNode \* first = nullptr, TreeNode \* next = nullptr)

: element{ e }, firstChild{ first }, nextSibling{next}{ }

};

template <typename Object>

int depth(TreeNode<Object>\* root, TreeNode<Object>\* node, int dep = 0) {

if (root == node) return dep;

if (root == nullptr) return 0;

int dep1 = 0, dep2 = 0;

if (root->nextSibling != nullptr) {

dep1 = depth(root->nextSibling, node, dep);

}

if (root->firstChild != nullptr) {

dep2 = depth(root->firstChild, node, dep + 1);

}

if (dep1 > dep2) return dep1;

else return dep2;

}

template <typename Object>

int height(TreeNode<Object> \*node, int d=0)

{

int hSibling = -1, hChild = -1;

if(node->nextSibling != nullptr)

hSibling = height(node->nextSibling, d);

if(node->firstChild != nullptr)

hChild = height( node->firstChild, d + 1);

int hMax = (hSibling > hChild) ? hSibling : hChild;

return (hMax > d) ? hMax : d;

}

int main()

{

TreeNode<char> \*root = new TreeNode<char>('A', nullptr,nullptr);

TreeNode<char> \*b = new TreeNode<char>('B', nullptr, nullptr);

TreeNode<char> \*c = new TreeNode<char>('C', nullptr, nullptr);

TreeNode<char> \*d = new TreeNode<char>('D', nullptr, nullptr);

TreeNode<char> \*e = new TreeNode<char>('E', nullptr, nullptr);

TreeNode<char> \*f = new TreeNode<char>('F', nullptr, nullptr);

TreeNode<char> \*g = new TreeNode<char>('G', nullptr, nullptr);

TreeNode<char> \*h = new TreeNode<char>('H', nullptr, nullptr);

TreeNode<char> \*i = new TreeNode<char>('I', nullptr, nullptr);

TreeNode<char> \*j = new TreeNode<char>('J', nullptr, nullptr);

TreeNode<char> \*k = new TreeNode<char>('K', nullptr, nullptr);

TreeNode<char> \*l = new TreeNode<char>('L', nullptr, nullptr);

TreeNode<char> \*m = new TreeNode<char>('M', nullptr, nullptr);

TreeNode<char> \*n = new TreeNode<char>('N', nullptr, nullptr);

TreeNode<char> \*o = new TreeNode<char>('O', nullptr, nullptr);

TreeNode<char> \*p = new TreeNode<char>('P', nullptr, nullptr);

TreeNode<char> \*q = new TreeNode<char>('Q', nullptr, nullptr);

root->firstChild = b;

b->nextSibling = c;

c->nextSibling = d;

d->firstChild = h;

d->nextSibling = e;

e->firstChild = i;

e->nextSibling = f;

f->firstChild = k;

f->nextSibling = g;

g->nextSibling = n;

n->firstChild = o;

o->nextSibling = p;

o->firstChild = q;

cout << "The height of the tree is: " << height(root) << endl;

cout << "The depth of node Q is: " << depth(root, q) << endl;

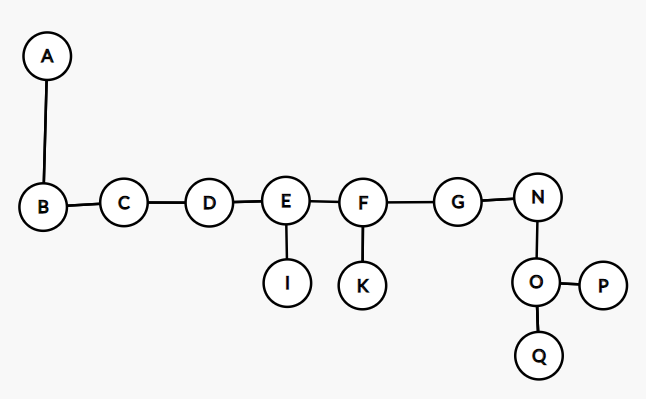
cout << "The depth of node O is: " << depth(root, o) << endl;

cout << "The depth of node D is: " << depth(root, d) << endl;

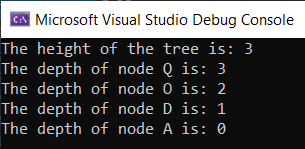
cout << "The depth of node A is: " << depth(root, root) << endl;

}

The tree looks like:



Screenshot:



***Exercise 2:***

Write the following recursive functions that work on a binary tree. The*BinaryNode*structure is provided to you in appendix. The function *MakeTree* is also given, which takes an algebraic expression as string and creates a matching tree representation.

1. A function that prints the tree in *Infix* equivalent format (cf. example hereafter):

template <typename Object>

void inFix(BinaryNode<Object>\* node)

1. A function that prints the tree in *Postfix* equivalent format (cf. example hereafter):

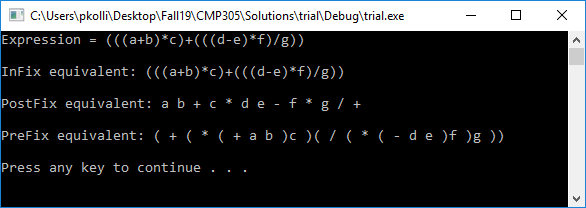
template <typename Object>

void postFix(BinaryNode<Object>\* node)

1. A Function that prints the tree in *Prefix* equivalent format (cf. example hereafter):

template <typename Object>

void preFix(BinaryNode<Object>\* node)



You should use the provided main() to test the three functions.

Code:

#include <iostream>

using namespace std;

template <typename Object>

struct BinaryNode

{

Object element;

BinaryNode \*left;

BinaryNode \*right;

BinaryNode(const Object & theElement, BinaryNode \*lt = nullptr, BinaryNode \*rt = nullptr)

: element{ theElement }, left{ lt }, right{ rt } { }

};

template <typename Object>

void MakeTree(string expr, BinaryNode<Object> \*node, int& pos);

template <typename Object>

void preFix(BinaryNode<Object>\* node);

template <typename Object>

void postFix(BinaryNode<Object>\* node);

template <typename Object>

void inFix(BinaryNode<Object>\* node);

int main()

{

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

int index = 0;

BinaryNode<char>\* root = new BinaryNode<char>(' ');

int pos = 0;

MakeTree(a, root, index);

cout << "Expression = " << a << "\n\n";

cout << "InFix equivalent: ";

inFix(root);

cout << "\n\nPostFix equivalent: ";

postFix(root);

cout << "\n\nPreFix equivalent: ";

preFix(root);

cout << "\n\n";

return 0;

}

//A recursive function that takes an expression and makes the tree

template <typename Object>

void MakeTree(string expr, BinaryNode<Object> \*node, int& pos)

{

if (expr[pos] == '(')

{

pos++;

node->left = new BinaryNode<char>('0');

node->left->left = NULL;

node->left->right = NULL;

MakeTree(expr, node->left, pos);

MakeTree(expr, node, pos);

return;

}

if (expr[pos] >= 'a' && expr[pos] <= 'z')

{

node->element = expr[pos];

pos++;

return;

}

if (expr[pos] == '+' || expr[pos] == '-' || expr[pos] == '\*' || expr[pos] == '/')

{

node->element = expr[pos];

pos++;

node->right = new BinaryNode<char>('0');

node->right->left = NULL;

node->right->right = NULL;

MakeTree(expr, node->right, pos);

}

if (expr[pos] == ')')

{

pos++;

return;

}

}

template <typename Object>

void preFix(BinaryNode<Object>\* node)

{

if(node->element=='\*'|| node->element == '/'|| node->element == '+'|| node->element == '-')

cout << "(";

cout << node->element << " ";

if (node->left != nullptr) {

preFix(node->left);

}

if (node->right != nullptr) {

preFix(node->right);

cout << ")";

}

}

template <typename Object>

void inFix(BinaryNode<Object>\* node)

{

if (node->left != nullptr) {

cout << "(";

inFix(node->left);

}

cout << node->element;

if (node->right != nullptr) {

inFix(node->right);

cout << ")";

}

}

template <typename Object>

void postFix(BinaryNode<Object>\* node)

{

if (node->left != nullptr) {

postFix(node->left);

}

if (node->right != nullptr) {

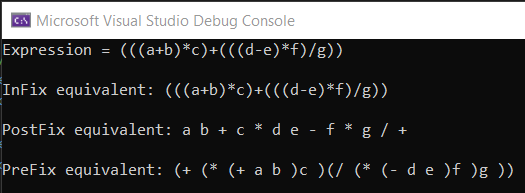
postFix(node->right);

}

cout << node->element<<" ";

}

Screenshots:



***Bonus:***

***Exercise 3:***

In the Array-based implementation of Binary Tree, the left child of the node at index i is at index (2\*i + 1) and the right child of the node at index (2\*i+ 2). The parent of the node at index i is at index ((i-1)/2).

Write the following functions,

1. Construct a binary tree from the given array

template <typename Object>

void ArrayToTree(BinaryNode<Object> \*&tree, Object arr[], int size, int startIndex)

1. Construct an array from the given binary tree

template <typename Object>

void treeToArray(BinaryNode<Object>\* tree, Object values[], int& index)