**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ИС**

ОТЧЁТ

**по лабораторной работе №3**

**по дисциплине «Алгоритмы и Структуры Данных»**

Тема: Деревья

|  |  |  |
| --- | --- | --- |
| Студент гр. 1375 |  | Подберёзский А.Д. |
| Преподаватель |  | Пелевин М.С. |

Санкт-Петербург

2022

**Цель работы:**

Ознакомление с иерархической структурой данных(деревьями) и её подвидами

**Задачи:**

Реализовать:

1. Обычное двоичное дерево (не поиска и не балансирующее) с реализацией обхода этого дерева в глубину (направление обхода не важно, обход через рекурсию).
2. Алгоритм для парсинга скобочной записи дерева. Пример: (8 (9 (5)) (1)) — корень 8, левый сын 9, правый 1, левый сын узла 9 – 5.
3. Реализовать АВЛ дерево (усложненный вариант — КЧ дерево) с реализацией алгоритмов вставки, удаления, поиска, обхода в ширину, обхода в глубину (все 3 вида и использовать итеративный вариант).

Продемонстрировать:

1. Чтение скобочной записи (путем чтения из файла).
2. Шаг 1 или создает двоичное дерево или сообщает об ошибке (неправильные символы, неправильно расставлены скобки, не двоичное дерево вообще и т.п.).
3. Обойти полученное дерево и создать АВЛ (или КЧ) дерево.
4. Продемонстрировать вывод всех узлов 4 способами: в ширину и 3 в глубину.

**Код программы:**

**Файл BinTree.h:**

#pragma once

#include <iostream>

#include <algorithm>

struct node

{

int value;

node\* left;

node\* right;

node(int v)

{

value = v;

left = right = nullptr;

}

};

class BinTree

{

private:

int\* nodes;

int i = 0;

void orderEvent(int value)

{

nodes[i] = value;

++i;

}

void preOrder(node\* a)

{

if (a)

{

orderEvent(a->value);

preOrder(a->left);

preOrder(a->right);

}

}

void inOrder(node\* a)

{

if (a)

{

inOrder(a->left);

orderEvent(a->value);

inOrder(a->right);

}

}

void postOrder(node\* a)

{

if (a)

{

postOrder(a->left);

postOrder(a->right);

orderEvent(a->value);

}

}

bool isNumber(char a)

{

return a >= '0' && a <= '9';

}

node\* parsing(node\* a, std::string s, int& i)//in progress of normal realization

{

if (s[i] == '(')

{

std::string valueStr;

i++;

while (isNumber(s[i]))

{

valueStr += s[i];

i++;

}

if (valueStr.size() != 0)

{

int valueTmp = stoi(valueStr);

a->left = new node(valueTmp);

++counter;

}

else

{

return nullptr;

}

if (s[i] == '(')

{

parsing(a->left, s, i);

}

if (s[i] != ')')

{

return nullptr;

}

i++;

}

if (s[i] == '(')

{

std::string valueStr;

i++;

while (isNumber(s[i]))

{

valueStr += s[i];

i++;

}

if (valueStr.size() != 0)

{

int valueTmp = stoi(valueStr);

a->right = new node(valueTmp);

++counter;

}

else

{

return nullptr;

}

if (s[i] == '(')

{

parsing(a->right, s, i);

}

if (s[i] != ')')

{

return nullptr;

}

i++;

}

return a;

}

node\* findMin(node\* a)

{

return a->left ? findMin(a->left) : a;

}

node\* removeMin(node\* a)

{

if (a->left == 0)

{

return a->right;

}

a->left = removeMin(a->left);

return a;

}

node\* insert(node\* a, int value)

{

if (!a)

{

return new node(value);

}

if (value < a->value)

{

a->left = insert(a->left, value);

}

else

{

a->right = insert(a->right, value);

}

return a;

}

node \*remove(node \*a, int value)

{

if (!a)

{

return 0;

}

if (value < a->value)

{

a->left = remove(a->left, value);

}

else

{

if (value > a->value)

{

a->right = remove(a->right, value);

}

else

{

node\* b = a->left;

node\* c = a->right;

delete a;

if (!c)

{

return b;

}

node\* min = findMin(c);

min->right = removeMin(c);

min->left = b;

return min;

}

}

return a;

}

public:

node\* root;

int counter = 0; //counts number of nodes

BinTree()

{

root = nullptr;

nodes = nullptr;

counter = 0;

}

void add(int value)

{

root = insert(root, value);

++counter;

}

void del(int value)

{

root = remove(root, value);

counter--;

}

void strToTree(std::string s)

{

bool error = 0;

int brackets = 0;

s.erase(std::remove(s.begin(), s.end(), ' '), s.end());

for (int i = 0; i < s.size(); i++)

{

if (s[i] == '(')

{

brackets++;;

}

else if (s[i] == ')')

{

brackets--;

}

else if (!isNumber(s[i]))

{

error = 1;

break;

}

}

if (s.size() != 0 && !error && !brackets)

{

int i = 0;

std::string tmp;

while (isNumber(s[i]))

{

tmp += s[i];

i++;

}

add(stoi(tmp));

root = parsing(root, s, i);

}

}

int\* preOrder()

{

if (root)

{

delete []nodes;

nodes = new int[counter];

preOrder(root);

i = 0;

return nodes;

}

}

int\* inOrder()

{

if (root)

{

delete[]nodes;

nodes = new int[counter];

inOrder(root);

i = 0;

return nodes;

}

}

int\* postOrder()

{

if (root)

{

delete[]nodes;

nodes = new int[counter];

postOrder(root);

i = 0;

return nodes;

}

}

};

void printTree(node\* a, int lvl)

{

if (a)

{

printTree(a->right, lvl + 1);

for (int i = 0; i < lvl; i++)

{

std::cout << " ";

}

std::cout << a->value << "-<\n";

printTree(a->left, lvl + 1);

}

}

**Файл AVL.h:**

#pragma once

#include <iostream>

#include <algorithm>

struct avlNode

{

int value;

int height;

avlNode\* left;

avlNode\* right;

avlNode(int v)

{

value = v;

left = right = nullptr;

height = 1;

}

};

class AVL

{

private:

int\* nodes = 0;

int i;

void orderEvent(int value)

{

nodes[i] = value;

++i;

}

void preOrder(avlNode\* a)

{

if (a)

{

orderEvent(a->value);

preOrder(a->left);

preOrder(a->right);

}

}

void inOrder(avlNode\* a)

{

if (a)

{

inOrder(a->left);

orderEvent(a->value);

inOrder(a->right);

}

}

void postOrder(avlNode\* a)

{

if (a)

{

postOrder(a->left);

postOrder(a->right);

orderEvent(a->value);

}

}

int getHeight(avlNode\* a)

{

return a ? a->height : 0;

}

int getBalanceFactor(avlNode\* a)

{

return getHeight(a->right) - getHeight(a->left);

}

void editHeight(avlNode\* a)

{

int heightLeft = getHeight(a->left);

int heightRight = getHeight(a->right);

a->height = (heightLeft > heightRight ? heightLeft : heightRight) + 1;

}

avlNode \*rightRotate(avlNode\* a)

{

avlNode\* q = a->left;

a->left = q->right;

q->right = a;

editHeight(a);

editHeight(q);

return q;

}

avlNode \*leftRotate(avlNode\* a)

{

avlNode\* b = a->right;

a->right = b->left;

b->left = a;

editHeight(a);

editHeight(b);

return b;

}

avlNode \*balance(avlNode\* a)

{

editHeight(a);

if (getBalanceFactor(a) == 2)

{

if (getBalanceFactor(a->right) < 0)

{

a->right = rightRotate(a->right);

}

return leftRotate(a);

}

if (getBalanceFactor(a) == -2)

{

if (getBalanceFactor(a->left) > 0)

{

a->left = leftRotate(a->left);

}

return rightRotate(a);

}

return a;

}

avlNode \*findMin(avlNode\* a)

{

return a->left ? findMin(a->left) : a;

}

avlNode \*removeMin(avlNode\* a)

{

if (a->left == 0)

{

return a->right;

}

a->left = removeMin(a->left);

return balance(a);

}

avlNode \*insert(avlNode\* a, int value)

{

if (!a)

{

return new avlNode(value);

}

if (value < a->value)

{

a->left = insert(a->left, value);

}

else

{

a->right = insert(a->right, value);

}

return balance(a);

}

avlNode \*remove(avlNode\* a, int value)

{

if (!a)

{

return 0;

}

if (value < a->value)

{

a->left = remove(a->left, value);

}

else

{

if (value > a->value)

{

a->right = remove(a->right, value);

}

else

{

avlNode\* b = a->left;

avlNode\* c = a->right;

delete a;

if (!c)

{

return b;

}

avlNode\* min = findMin(c);

min->right = removeMin(c);

min->left = b;

return balance(min);

}

}

return balance(a);

}

public:

avlNode\* root;

int counter;

AVL()

{

root = nullptr;

nodes = nullptr;

counter = 0;

i = 0;

}

void add(int value)

{

root = insert(root, value);

++counter;

}

void del(int value)

{

root = remove(root, value);

counter--;

}

int\* preOrder()

{

if (root)

{

delete []nodes;

nodes = new int[counter];

preOrder(root);

i = 0;

return nodes;

}

}

int\* inOrder()

{

if (root)

{

delete[]nodes;

nodes = new int[counter];

inOrder(root);

i = 0;

return nodes;

}

}

int\* postOrder()

{

if (root)

{

delete[]nodes;

nodes = new int[counter];

postOrder(root);

i = 0;

return nodes;

}

}

};

void printAVL(avlNode\* a, int level)

{

if (a)

{

printAVL(a->right, level + 1);

for (int i = 0; i < level; i++)

{

std::cout << " ";

}

std::cout << a->value << "-<\n";

printAVL(a->left, level + 1);

}

}

**Файл fileFunc.h:**

#pragma once

#include <cassert>

#include <iostream>

#include <fstream>

bool inputInFile(const char\* filename)

{

assert(filename != nullptr);//checking name of file

std::ofstream file;//opening file

file.open(filename);

assert(file.is\_open() == true);//checking if file is opened

std::string str;

std::cin >> str;

file << str << std::endl;

file.close();

return true;

}

bool readingFromFile(const char\* filename, std::string &inputstring)

{

assert(filename != nullptr); //checking name of file

std::ifstream file;//opening file

file.open(filename);

assert(file.is\_open() == true); //checking if file is opened

file >> inputstring;

file.close();//closing file

return true;

}

**Файл main.cpp:**

#include <iostream>

#include <string>

#include "AVL.h"

#include "BinTree.h"

#include "fileFunc.h"

int main(int argc, char\* argv[]) {

BinTree tree;

std::string s;

int choose = 0;

std::cout << "What type of insert do you want:" << std::endl

<< "1 - constant string" << std::endl

<< "2 - string from console" << std::endl

<< "3 - string from file" << std::endl;

std::cin >> choose;

if (choose == 1)

{

s = "6 (2(1)(4(3)(5))) (7(9(8)))";

}

else if (choose == 2)

{

std::cout << "Insert tree(without first and last brackets and spaces):" << std::endl;

std::cin >> s;

}

else if (choose == 3)

{

char filename[] = "input.txt";

std::cout << "Standart name of file is 'input.txt'" << std::endl;

readingFromFile(filename, s);

std::cout << "Your string is:" << std::endl << s << std::endl;

}

tree.strToTree(s);

AVL avlTree;

int\* treeNodes = tree.preOrder();

for (int i = 0; i < tree.counter; ++i)

{

avlTree.add(treeNodes[i]);

}

std::cout << "Tree:\n";

printTree(tree.root, 1);

std::cout << "\n\nAVL Tree:\n";

printAVL(avlTree.root, 1);

std::cout << "\n\n";

int \*avlTreeNodes = avlTree.preOrder();

std::cout << "Preorder - ";

for (int i = 0; i < avlTree.counter; i++)

{

std::cout << avlTreeNodes[i] << " ";

}

avlTreeNodes = avlTree.inOrder();

std::cout << "\nInorder - ";

for (int i = 0; i < avlTree.counter; i++)

{

std::cout << avlTreeNodes[i] << " ";

}

avlTreeNodes = avlTree.postOrder();

std::cout << "\nPostorder - ";

for (int i = 0; i < avlTree.counter; i++)

{

std::cout << avlTreeNodes[i] << " ";

}

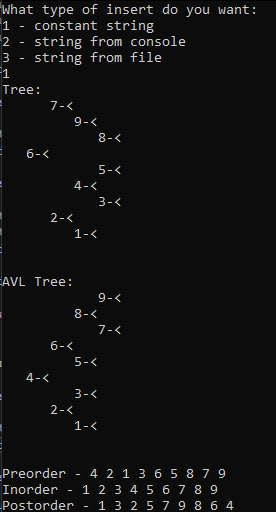
std::cout << "\n";

system("pause");

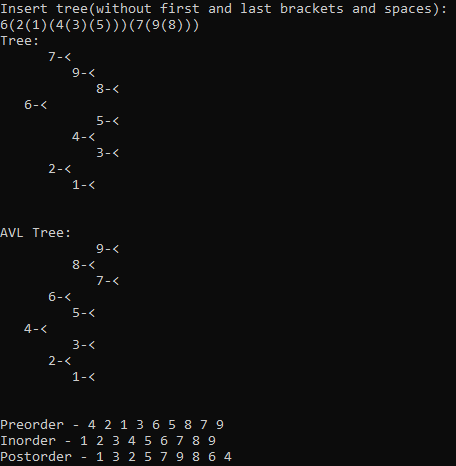
}

**Пример работы программы:**

Пример работы с константной записью 6 (2(1)(4(3)(5))) (7(9(8))):



Пример работы с записью из консоли:

  
Пример работы с записью из файла:

