Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное учреждение высшего образования

«Пермский национальный исследовательский политехнический университет»

ПНИПУ

**Лабораторная работа**

**«Бинарные деревья»**

Выполнила:

студентка группы ИВТ-23-2Б

Сычева Евгения Андреевна

Проверила:

Доцент кафедры ИТАС

О. А. Полякова

2024 г.

**Постановка задачи:**

1. Сформировать идеально сбалансированное бинарное дерево, тип информационного поля указан в варианте.

2. Распечатать полученное дерево.

3. Выполнить обработку дерева в соответствии с заданием, вывести полученный результат.

4. Преобразовать идеально сбалансированное дерево в дерево поиска.

5. Распечатать полученное дерево.

**Вариант 20:** Тип информационного поля double. Найти среднее арифметическое элементов дерева.

**Анализ задачи:**

1. Необходимо создать класс Tree со значениями:

data – данные, из которых состоит дерево, типа double

left – указатель на узел слева

right – указатель на узел справа

parent – указатель на предка (родителя)

2. Необходимо прописать для класса следующие функции:

* Tree<T>() - конструктор без значений
* Tree<T>(T dt) – конструктор со значением data
* ~Tree<T>() – деструктор
* void InsertLeft(T dt) – функция, которая вставляет узел со значением data слева
* void InsertRight(T dt) – функция, которая вставляет узел со значением data справа
* void Insert(T dt) - функция, которая вставляет узел со значением data в нужное место в дереве
* void AddLeftTree(Tree<T>\* tree) – функция, которая добавляет поддерево слева
* void AddRightTree(Tree<T>\* tree) – функция, которая добавляет поддерево справа
* Tree<T>\* EjectLeft() – функция, которая извлекает поддерево слева
* Tree<T>\* EjectRight() – функция, которая извлекает поддерево справа
* void DeleteLeft() – функция, которая удаляет поддерево слева
* void DeleteRight() – функция, которая удаляет поддерево справа
* void SetData(T dt) – функция, которая устанавливает данные data для узла
* Tree<T>\* GetRight() – функция, которая возвращает указатель на правое поддерево
* Tree<T>\* GetLeft() – функция, которая возвращает указатель на левое поддерево
* Tree<T>\* GetParent() – функция, которая возвращает указатель на родительский узел
* Tree<T>\* Erase(Tree\* node, T val) - функция, которая удаляет узел с заданным значением
* Tree<T>\* Search(T key) – функция, которая ищет узел по заданному ключу
* Tree<T> Find(T dt) – функция, которая ищет узел с исходным знаяением
* static void ReOrder(Tree<T>\* node) – функция, которая проходит дерево в прямом порядке
* static void InOrder(Tree<T>\* node) - функция, которая проходит дерево в симметричном порядке
* static void PostOrder(Tree<T>\* node) - функция, которая проходит дерево в обратном порядке
* Tree<T>\* CopyTree() – функция, которая копирует дерево
* T GetData() – функция, которая получает данные с узла
* void PrintTree(int level) – функция для горизонтальной печати
* void obh(Tree<T>\* node) – дополнительная функция для вертикальной печати дерева
* void PrintVTree() – функция для вертикальной печати
* int GetAmounthOfNodes() – функция, которая считает количество элементов в дереве
* int GetHeight() – функция, которая считает высоту дерева
* T LevelScan() - функция, которая проходит по каждому уровню дерева начиная с корня и считает сумму их значений
* void DeleteTree() – функция, которая удаляет дерево
* static Tree<T>\* BalancedTree(int count) – функция, которая строит идеально сбалансированное дерево
* static Tree<T>\* BuildBalans(const vector<T> data, int start, int end) – функция, которая рекурсивно строит сбалансированное дерево
* void InOrderTraversal(vector<T>& result) – функция, которая обходит дерево в порядке возрастания и сохраняет значения в векторе
* static Tree<T>\* CreateBalans(Tree<T>\* root) – функция, которая создает дерево поиска из сбалансированное

**Код программы на языке С++:**

Tree.h:

#pragma once

#include <iostream>

#include <vector>

#include <string>

#include <ctime>

#include <list>

#include <fstream>

#include <queue>

#include <algorithm>

using namespace std;

template <class T>

class Tree

{

private:

T data;//Данные типа double

Tree<T>\* left; //Указатель на узел слева

Tree<T>\* right;//Указатель на узел справа

Tree<T>\* parent;//Указатель на предка

public:

Tree<T>()// Конструктор без значений

{

left = right = parent = NULL;

}

Tree<T>(T dt)// Конструктор

{

this->data = dt;

left = NULL;

right = NULL;

parent = NULL;

}

~Tree<T>()// Деструктор

{

DeleteRight();

DeleteLeft();

DeleteTree();

}

void InsertLeft(T dt)// Вставить узел слева

{

Tree<T>\* node = new Tree(dt);

if (this->left != NULL)

{

this->left->parent = node;

node->left = this->left;

}

this->left = node;

node->parent = this;

}

void InsertRight(T dt)// Вставить узел справа

{

Tree<T>\* node = new Tree(dt);

if (this->right != NULL)

{

this->right->parent = node;

node->right = this->right;

}

this->right = node;

node->parent = this;

}

void Insert(T dt)// Вставить узел со значением dt в нужное место в дереве

{

Tree<T>\* current = this;

while (current != NULL)

{

if (dt > current->data)

{

if (current->right != NULL)

{

current = current->right;

}

else

{

current->InsertRight(dt);

return;

}

}

else

{

if (dt < current->data)

{

if (current->left != NULL)

{

current = current->left;

}

else

{

current->InsertLeft(dt);

return;

}

}

else return;

}

}

}

void AddLeftTree(Tree<T>\* tree)// Добавить поддерево слева

{

left = tree;

}

void AddRightTree(Tree<T>\* tree)// Добавить поддерево справа

{

right = tree;

}

Tree<T>\* EjectLeft()// Извлечь поддерево слева

{

if (this->left != NULL)

{

Tree<T>\* temp = this->left;

this->left = NULL;

return temp;

}

return NULL;

}

Tree<T>\* EjectRight()// Извлечь поддерево справа

{

if (this->right != NULL)

{

Tree<T>\* temp = this->right;

this->right = NULL;

return temp;

}

return NULL;

}

void DeleteLeft()// Удалить поддерево слева

{

Tree<T>\* temp = this->EjectLeft();

delete temp;

}

void DeleteRight()// Удалить поддерево справа

{

Tree<T>\* temp = this->EjectRight();

delete temp;

}

void SetData(T dt)// Установить данные для узла

{

data = dt;

}

Tree<T>\* GetRight()//Возвращает указатель на правое поддерево

{

return this->right;

}

Tree<T>\* GetLeft()//Возвращает указатель на левое поддерево

{

return this->left;

}

Tree<T>\* GetParent()//Возвращает указатель на родительский узел

{

return this->parent;

}

Tree<T>\* Erase(Tree\* node, T val)//Функция, которая удаляет узел с заданным значением

{

if (node == NULL)

return node;

if (val == node->data) {

Tree\* tmp;

if (node->right == NULL)

tmp = node->left;

else {

Tree\* ptr = node->right;

if (ptr->left == NULL) {

ptr->left = node->left;

tmp = ptr;

}

else {

Tree\* pmin = ptr->left;

while (pmin->left != NULL) {

ptr = pmin;

pmin = ptr->left;

}

ptr->left = pmin->right;

pmin->left = node->left;

pmin->right = node->right;

tmp = pmin;

}

}

delete node;

return tmp;

}

else if (val > node->data)

{

node->left = Erase(node->left, val);

}

else

node->right = Erase(node->right, val);

return node;

}

Tree<T>\* Search(T key)// Поиск узла по заданному ключу

{

if (data == key)

{

return this;

}

if (left != NULL)

{

Tree<T>\* result = left->Search(key);

if (result != NULL)

{

return result;

}

}

if (right != NULL)

{

Tree<T>\* result = right->Search(key);

if (result != NULL)

{

return result;

}

}

return NULL;

}

Tree<T> Find(T dt)// Поиск узла с исхоным значением

{

if (this == NULL || this->data == data)

{

return this;

}

else

{

if (data > this->data)

{

return this->right->Find(dt);

}

else

{

return this->left->Find(dt);

}

}

}

static void ReOrder(Tree<T>\* node)// Пройти дерево с печатью элементов в прямом порядке(сверху вниз)

{

if (node != NULL)

{

cout << node->GetData() << ' ';

ReOrder(node->left);

ReOrder(node->right);

}

}

static void InOrder(Tree<T>\* node)// Пройти дерево с печатью элементов в симметричном порядке(слева направо)

{

if (node != NULL)

{

InOrder(node->left);

cout << node->GetData() << ' ';

InOrder(node->right);

}

}

static void PostOrder(Tree<T>\* node)// Пройти дерево с печатью элементов в обратном порядке(снизу вверх)

{

if (node != NULL)

{

PostOrder(node->left);

PostOrder(node->right);

cout << node->GetData() << ' ';

}

}

Tree<T>\* CopyTree()// Скопировать дерево

{

Tree<T>\* tree = new Tree<T>(this->data);

if (this->parent != NULL)

tree->parent = this->parent;

if (this->left != NULL)

tree->left = this->left->CopyTree();

if (this->right != NULL)

tree->right = this->right->CopyTree();

return tree;

}

T GetData()// Получить данные с узла

{

if (this != NULL)

return data;

else

return NULL;

}

void PrintTree(int level)// Печать дерева горизонтально

{

if (this != NULL)

{

this->left->PrintTree(level + 1);

for (int i = 1; i < level; i++)

{

cout << " ";

}

cout << this->GetData() << endl;

this->right->PrintTree(level + 1);

}

}

void obh(Tree<T>\* node)// Дополнительная функция для вертикальной печати дерева, прохождение по дереву и запись в файл

{

ofstream f("print.txt");

int amount = node->GetAmounthOfNodes();

queue<Tree<T>\*>q;

q.push(node);

while (!q.empty()) {

Tree<T>\* temp = q.front();

q.pop();

f << temp->data << endl;

if (temp->left) {

q.push(temp->left);

}

if (temp->right) {

q.push(temp->right);

}

}

f.close();

}

void PrintVTree()// Печать дерева вертикально

{

obh(this);

ifstream f("print.txt");

int height = this->GetHeight();

int count = 0;

int\* spaces = new int[height];

spaces[0] = 0;

for (int i = 1; i < height; i++) {

spaces[i] = spaces[i - 1] \* 2 + 1;

}

char str[255];

for (int i = 0, l = height - 1; i < height; i++, l--) {

for (int j = 0; j < pow(2, i); j++) {

if (j == 0) {

for (int u = 0; u < spaces[l]; u++) {

cout << " ";

}

}

else {

for (int u = 0; u < spaces[l + 1]; u++) {

cout << " ";

}

}

if (f.getline(str, 255)) {

cout << str;

}

else {

cout << " ";

}

}

cout << endl;

}

delete[] spaces;

f.close();

}

int GetAmounthOfNodes()// Получить количество элементов дерева

{

if (this == NULL)

{

return 0;

}

if ((this->left == NULL) && (this->right == NULL))

{

return 1;

}

int l = 0, r = 0;

if (this->left != NULL)

{

l = this->left->GetAmounthOfNodes();

}

if (this->right != NULL)

{

r = this->right->GetAmounthOfNodes();

}

return(l + r + 1);

}

int GetHeight()// Получить высоту дерева (считает с текущего узла по направлению к листьям)

{

int h1 = 0, h2 = 0, hadd = 0;

if (this == NULL)

{

return 0;

}

if (this->left != NULL)

{

h1 = this->left->GetHeight();

}

if (this->right != NULL)

{

h2 = this->right->GetHeight();

}

if (h1 >= h2)

{

return h1 + 1;

}

else

{

return h2 + 1;

}

}

T LevelScan()// Проходит по каждому уровню дерева начиная с корня и считает сумму их значений

{

vector<Tree<T>\*> V;

Tree<T>\* p = this;

T sum = 0;

V.push\_back(p);

for (int i = 0; i < this->GetAmounthOfNodes(); i++)

{

if (V.at(i)->left != NULL)

V.push\_back(V.at(i)->left);

if (V.at(i)->right != NULL)

V.push\_back(V.at(i)->right);

}

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

{

sum += V.at(i)->GetData();

}

return sum;

}

void DeleteTree()// Удалить дерево

{

delete this;

}

static Tree<T>\* BalancedTree(int count)//Построить идеально сбалансированное дерево по данной высоте

{

if (count <= 0)

{

return NULL;

}

T dt;

cout << "Введите данные для постройки сбалансированного дерева: ";

cin >> dt;

Tree<T>\* node = new Tree<T>(dt);

node->AddLeftTree(BalancedTree(count / 2));

node->AddRightTree(BalancedTree(count - count / 2 - 1));

return node;

}

static Tree<T>\* BuildBalans(const vector<T> data, int start, int end)// Рекурсивно строит сбалансированное дерево

{

if (start > end)

{

return nullptr;

}

int midle = start + (end - start) / 2;

Tree<T>\* new\_node = new Tree<T>(data[midle]);

new\_node->left = BuildBalans(data, start, midle - 1);

new\_node->right = BuildBalans(data, midle + 1, end);

return new\_node;

}

void InOrderTraversal(vector<T>& result)// Обход дерева в порядке возрастания и сохранение значений в векторе

{

if (left != nullptr) {

left->InOrderTraversal(result);

}

result.push\_back(data);

if (right != nullptr) {

right->InOrderTraversal(result);

}

}

static Tree<T>\* CreateBalans(Tree<T>\* root)// Создание дерева поиска из сбалансированного дерева

{

vector<T> sorted\_data;

root->InOrderTraversal(sorted\_data);

return BuildBalans(sorted\_data, 0, sorted\_data.size() - 1);

}

};

main.cpp:

#include <iostream>

#include "tree.h"

using namespace std;

int main()

{

setlocale(LC\_ALL, "RU");

Tree<double>\* TTree = new Tree<double>(1.2);

TTree->InsertLeft(2.3);

TTree->InsertRight(3.7);

TTree->GetLeft()->InsertLeft(4.3);

TTree->GetLeft()->InsertRight(7.8);

TTree->GetRight()->InsertLeft(9.7);

TTree->GetRight()->InsertRight(5.6);

cout << "Горизонтальный вывод дерева: " << endl;

TTree->PrintTree(1);

cout << endl << endl;

double key = 0;

cout << "Введите ключ для поиска элемента: ";

cin >> key;

if (TTree->Search(key))

{

cout << "Элемент " << key << " найден." << endl;

}

else

{

cout << "Такого элемента нет." << endl;

}

cout << endl << endl;

cout << "Вертикальный вывод дерева:" << endl;

TTree->PrintVTree();

cout << endl << endl;

cout << "Прямой обход дерева:" << endl;

TTree->ReOrder(TTree);

cout << endl << endl;

cout << "Симметричный обход дерева:" << endl;

TTree->InOrder(TTree);

cout << endl << endl;

cout << "Обратный обход дерева:" << endl;

TTree->PostOrder(TTree);

cout << endl << endl;

double sum = TTree->LevelScan();

int l = TTree->GetAmounthOfNodes();

double arithmetic = sum / l;

cout << "Среднее арифметическое число элементов дерева: " << arithmetic << endl << endl;

Tree<double>\* TTree\_Ball = new Tree<double>(1.8);

int n;

cout << "Введите количество элементов для сбалансированного дерева: ";

cin >> n;

Tree<double>\* TTree\_Balans = TTree\_Ball->BalancedTree(n);

cout << endl;

cout << "Горизонтальный вывод сбалансированного дерева:" << endl;

TTree\_Balans->PrintTree(1);

cout << endl << endl;

cout << "Вертикальный вывод сбалансированного дерева:" << endl;

TTree\_Balans->PrintVTree();

cout << endl << endl;

cout << "Преобразуем дерево, в дерево поиска: " << endl;

TTree\_Balans->CreateBalans(TTree\_Balans);

cout << "Горизонтальный вывод дерева поиска:" << endl;

TTree\_Balans->PrintTree(1);

cout << endl << endl;

cout << "Вертикальный вывод дерева поиска:" << endl;

TTree\_Balans->PrintVTree();

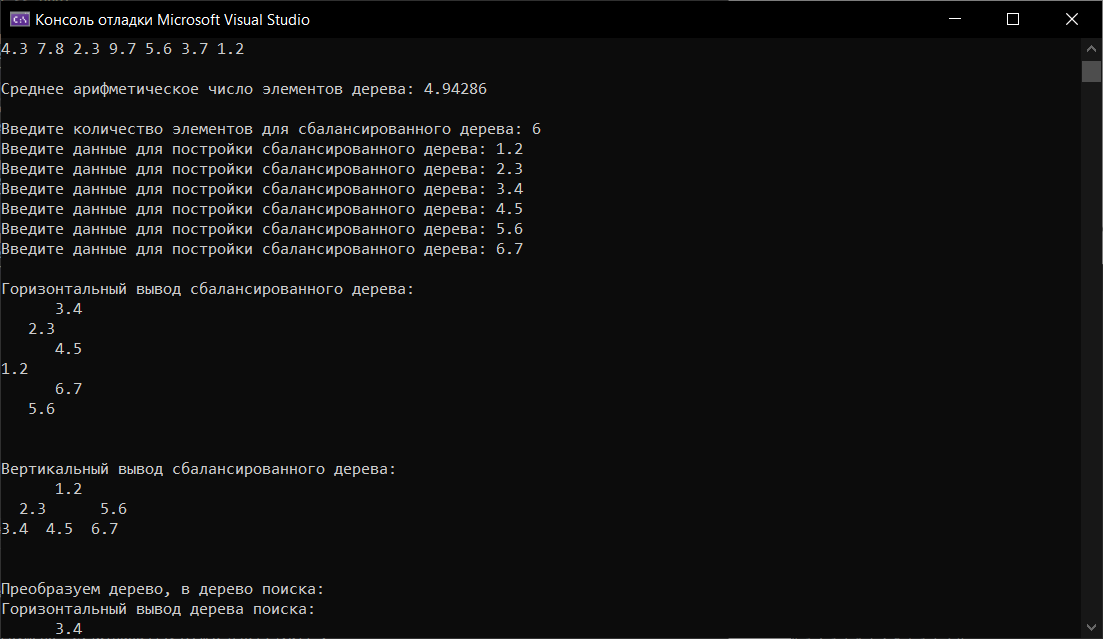
cout << endl << endl;

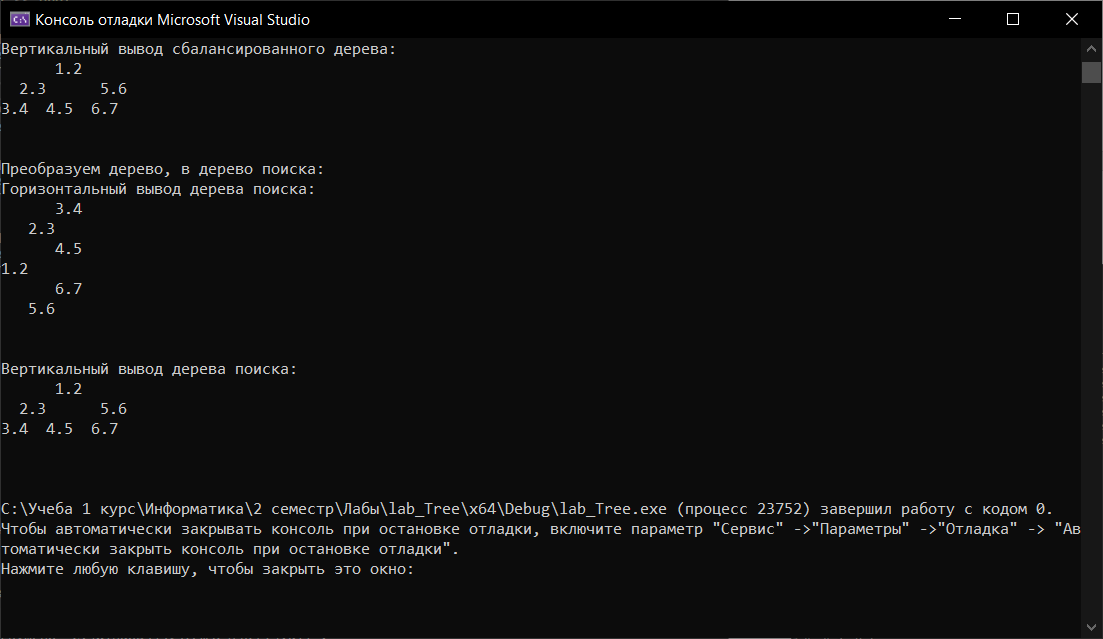
return 0;

}

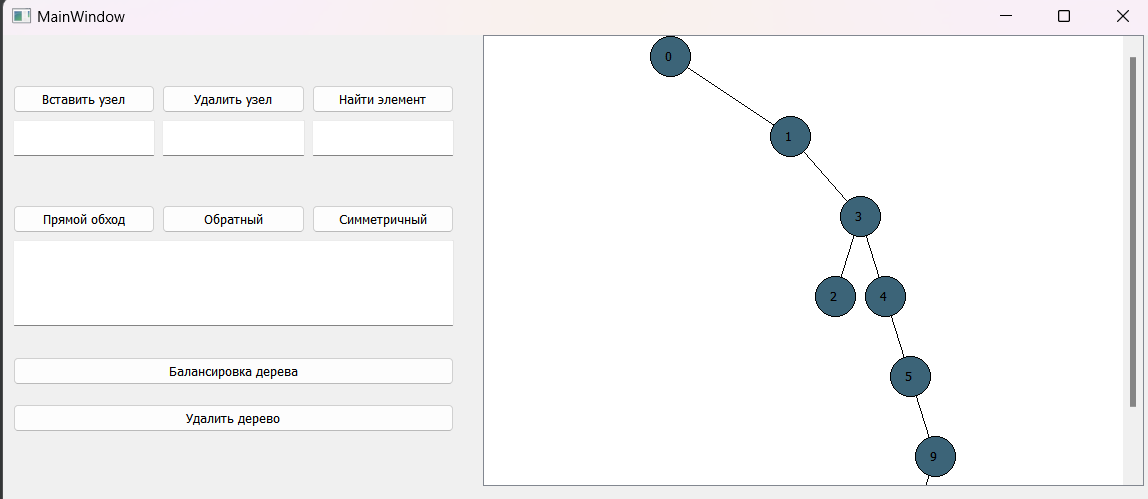
**Скриншоты пропущенного алгоритма через движок:**

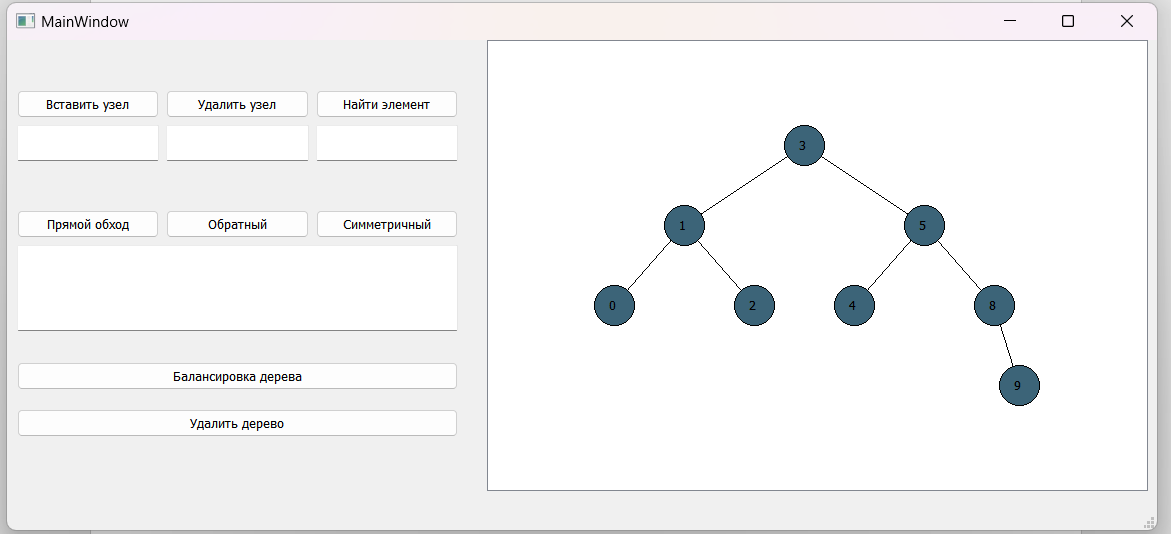
****

****

****

**Результат работы программы на Qt:**





**UML-диаграмма:**

