МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ  
федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА № 43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ассистент |  |  |  | М.А.Мурашова |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ № 6 |
| «АВЛ – деревья поиска» |
| по курсу: Структуры и алгоритмы обработки данных |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4134К |  |  |  | Опарин С.Н. |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2022

1. ***Цель работы***

Целью работы является изучение деревьев поиска и получение практических навыков их использования.

1. ***Задание***

Разработать на языке программирования высокого уровня программу, которая должна выполнять следующие функции:

− добавлять элементы в сбалансированное дерево поиска;

− удалять элементы из сбалансированного дерева поиска;

− искать элементы в дереве поиска с выводом количества шагов, за которое осуществляется поиск;

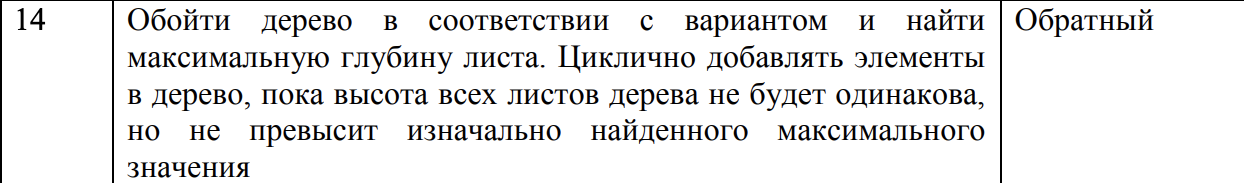
− выводить дерево на экран (любым способом доступным для восприятия);

− выводить список, соответствующий обходу вершин, в соответствии с вариантом задания;

− осуществлять операцию, заданную в таблице 6.

Количество элементов и порядок их ввода при создании сбалансированного дерева поиска определяется по согласованию с преподавателем.

1. ***Вариант задания***



1. ***Листинг программы, реализующей алгоритм:***

**Файл tree.h**

#ifndef tree\_h

#define tree\_h

#include <iostream>

#include <vector>

#include "node.h"

using namespace std;

class Tree {

private:

Node \*root = NULL;

vector<double\*> heightVector;

public:

Tree() {}

Node\* getRoot();

bool add(double data);

void removeElem(double data);

Node\* balanceTree(Node\* ptr);

void find(double data);

Node\* findElem(Node\* p,double data, size\_t &steps);

vector<double\*> getHeight();

void getHeightRecursion(Node\* ptr, int);

void levelOrder(Node\* ptr);

void show();

void showRecursion(string prefix, Node\* ptr, bool isLeft);

};

Node\* Tree::getRoot() {

return root;

}

bool Tree::add(double data) {

if (root == NULL) {

root = new Node(data);

} else {

Node \*newNode = root;

int height = 1;

while (true) {

if (root -> data == data) {

return false;

} else if (root -> data < data) {

if (root -> right != NULL) {

root = root -> right;

} else {

root -> right = new Node(data);

root -> right -> height = height;

break;

}

} else {

if (root -> left != NULL) {

root = root -> left;

} else {

root -> left = new Node(data);

root -> left -> height = height;

break;

}

}

height++;

}

root = newNode;

root = balanceTree(root);

}

return true;

}

void Tree::removeElem(double data) {

root = remove(root, data);

}

Node\* Tree::balanceTree(Node \*ptr) {

if (ptr -> left != NULL) {

ptr -> left = balanceTree(ptr -> left);

}

if (ptr -> right != NULL) {

ptr -> right = balanceTree(ptr -> right);

}

ptr = balanceNode(ptr);

return ptr;

}

void Tree::find(double data) {

size\_t steps = 0;

Node\* find\_element = findElem(root, data, steps);

if (find\_element != NULL) {

cout << "Для поиска потребовалось " << steps << " шагов" << endl;

showRecursion("", find\_element, false);

} else cout << "Такого элемента не существет." << endl;

}

Node\* Tree::findElem(Node\* p, double \_data, size\_t &steps) {

if (!p) return NULL;

if (\_data == p->data) {

return p;

}

if (\_data < p->data) {

++steps;

return findElem(p->left, \_data, steps);

}

else {

++steps;

return findElem(p->right, \_data, steps);

}

}

vector<double\*> Tree::getHeight() {

heightVector.clear();

getHeightRecursion(root, 1);

return heightVector;

}

void Tree::getHeightRecursion(Node\* ptr, int height = 1) {

if (ptr != NULL) {

if (ptr -> left == NULL && ptr -> right == NULL) {

double\* arr = new double[2];

arr[0] = ptr -> data;

arr[1] = height;

heightVector.push\_back(arr);

}

else {

if (ptr -> left != NULL) getHeightRecursion(ptr -> left, height + 1);

if (ptr -> right != NULL) getHeightRecursion(ptr -> right, height + 1);

}

}

}

void Tree::levelOrder(Node\* ptr) {

if (ptr == NULL) return;

cout << ptr -> data << " ";

levelOrder(ptr -> left);

levelOrder(ptr -> right);

}

void Tree::show() {

showRecursion("", root, false);

}

void Tree::showRecursion(string prefix, Node\* ptr, bool isRight) {

if (ptr != NULL) {

cout << prefix;

cout << (isRight ? " ├──" : " └──" );

cout << " " << ptr -> data << endl;

showRecursion(prefix + ((isRight) ? " │ " : " "), ptr -> right, true);

showRecursion(prefix + ((isRight) ? " │ " : " "), ptr -> left, false );

}

}

#endif /\* tree\_h \*/

**Файл node.h**

#ifndef node\_h

#define node\_h

struct Node {

double data;

int height;

Node \*left;

Node \*right;

Node(int d) { data = d; left = right = NULL; height = 1; }

};

unsigned char height(Node\* ptr) {

return ptr? ptr -> height: 0;

}

int balanceFactor(Node\* ptr) {

return height(ptr -> right) - height(ptr -> left);

}

void fixHeight(Node\* ptr) {

unsigned char hl = height(ptr->left);

unsigned char hr = height(ptr->right);

ptr -> height = (hl > hr? hl: hr) + 1;

}

Node\* rotateRight(Node\* ptr) {

Node\* tmp = ptr->left;

ptr->left = tmp->right;

tmp->right = ptr;

fixHeight(ptr);

fixHeight(tmp);

return tmp;

}

Node\* rotateLeft(Node\* tmp) {

Node\* ptr = tmp -> right;

tmp -> right = ptr -> left;

ptr -> left = tmp;

fixHeight(tmp);

fixHeight(ptr);

return ptr;

}

Node\* balanceNode(Node\* ptr) {

fixHeight(ptr);

if (balanceFactor(ptr) == 2) {

if (balanceFactor(ptr -> right) < 0)

ptr->right = rotateRight(ptr -> right);

return rotateLeft(ptr);

}

if (balanceFactor(ptr) == -2) {

if (balanceFactor(ptr -> left) > 0)

ptr -> left = rotateLeft(ptr -> left);

return rotateRight(ptr);

}

return ptr;

}

Node \*findMin(Node \*ptr) {

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

}

Node\* removeMin(Node\* ptr) {

if(ptr->left == 0)

return ptr->right;

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

return balanceNode(ptr);

}

Node\* remove(Node\* ptr, int \_data) {

if( !ptr ) return 0;

if( \_data < ptr->data )

ptr->left = remove(ptr->left, \_data);

else if( \_data > ptr->data )

ptr->right = remove(ptr->right, \_data);

else {

Node\* l = ptr->left;

Node\* r = ptr->right;

delete ptr;

if( !r ) return l;

Node\* min = findMin(r);

min->right = removeMin(r);

min->left = l;

return balanceNode(min);

}

return balanceNode(ptr);

}

#endif /\* node\_h \*/

**Файл main.cpp**

#include <iostream>

#include <cmath>

#include <time.h>

#include <iomanip>

#include "tree.h"

#define RANDOM\_MIN -100

#define RANDOM\_MAX 100

using namespace std;

int main() {

bool GLOBAL\_BREAK = false;

string menuStr;

int menuInt;

time\_t t;

srand((unsigned) time(&t));

Tree tree;

while (true) {

if (GLOBAL\_BREAK) break;

cout << "-------------------------------------------------------\n";

cout << "1. Добавить элемент" << endl;

cout << "2. Генерация рандомных значений" << endl;

cout << "3. Удалить элемент" << endl;

cout << "4. Поиск" << endl;

cout << "5. Обратный обход дерева" << endl;

cout << "6. Вывод глубин листов (minh, maxh)" << endl;

cout << "7. Выровнять листы (minh == maxh)" << endl;

cout << "0. Выход" << endl << endl;

cout << "Ваш выбор: ";

cin >> menuStr;

if (!menuStr[0] || menuStr[1]) continue;

menuInt = atoi(&menuStr[0]);

cout << endl;

switch (menuInt) {

case (1): {

int n;

cout << "Количество новых элементов: ";

cin >> n;

for (int i = 0; i < n; i++) {

while (true) {

int elem;

cout << "Введите значение "<< i <<"-го элемента: ";

cin >> elem;

if (!tree.add(elem)) {

cout << endl<< "Такой элемент уже существует" << endl << endl;

} else break;

}

}

tree.show();

break;

}

case (2): {

double a;

cout << "Количество новых элементов: ";

cin >> a;

for (int i = 0; i < a; i++) {

tree.add(RANDOM\_MIN + rand() % (RANDOM\_MAX - (RANDOM\_MIN) + 1));

}

tree.show();

break;

}

case (3): {

double a;

cout << "Удаляемый элемент: ";

cin >> a;

tree.removeElem(a);

tree.show();

break;

}

case (4): {

double a;

cout << "Элемент: ";

cin >> a;

tree.find(a);

break;

}

case (5): {

tree.levelOrder(tree.getRoot());

cout << endl << endl;

tree.show();

break;

}

case (6): {

vector<double\*> vec = tree.getHeight();

int minKey = 0;

int maxKey = 0;

cout << "Все листы: ";

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

cout << "\t" << vec.at(i)[0] << ":" << vec.at(i)[1] << " ";

if (vec.at(i)[1] < vec.at(minKey)[1])

minKey = i;

if (vec.at(i)[1] > vec.at(maxKey)[1])

maxKey = i;

}

cout << endl;

cout << "minh = " << vec.at(minKey)[0] << ":" << vec.at(minKey)[1] << endl;

cout << "maxh = " << vec.at(maxKey)[0] << ":" << vec.at(maxKey)[1] << endl;

tree.show();

break;

}

case (7): {

vector<double\*> vec;

int minKey, maxKey;

vec = tree.getHeight();

maxKey = minKey = vec.at(0)[1];

for (int i = 1; i < vec.size(); i++) {

if (vec.at(i)[1] > maxKey) {

maxKey = vec.at(i)[1];

}

else if (vec.at(i)[1] < minKey)

minKey = vec.at(i)[1];

}

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

if (vec.at(i)[1] < maxKey) {

int f = vec.at(i)[0];

if (f > 0) {

tree.add(f + 1);

}

else if(f < 0) {

tree.add(f - 1);

}

}

}

tree.show();

break;

}

case 0: {

GLOBAL\_BREAK = 0;

break;

}

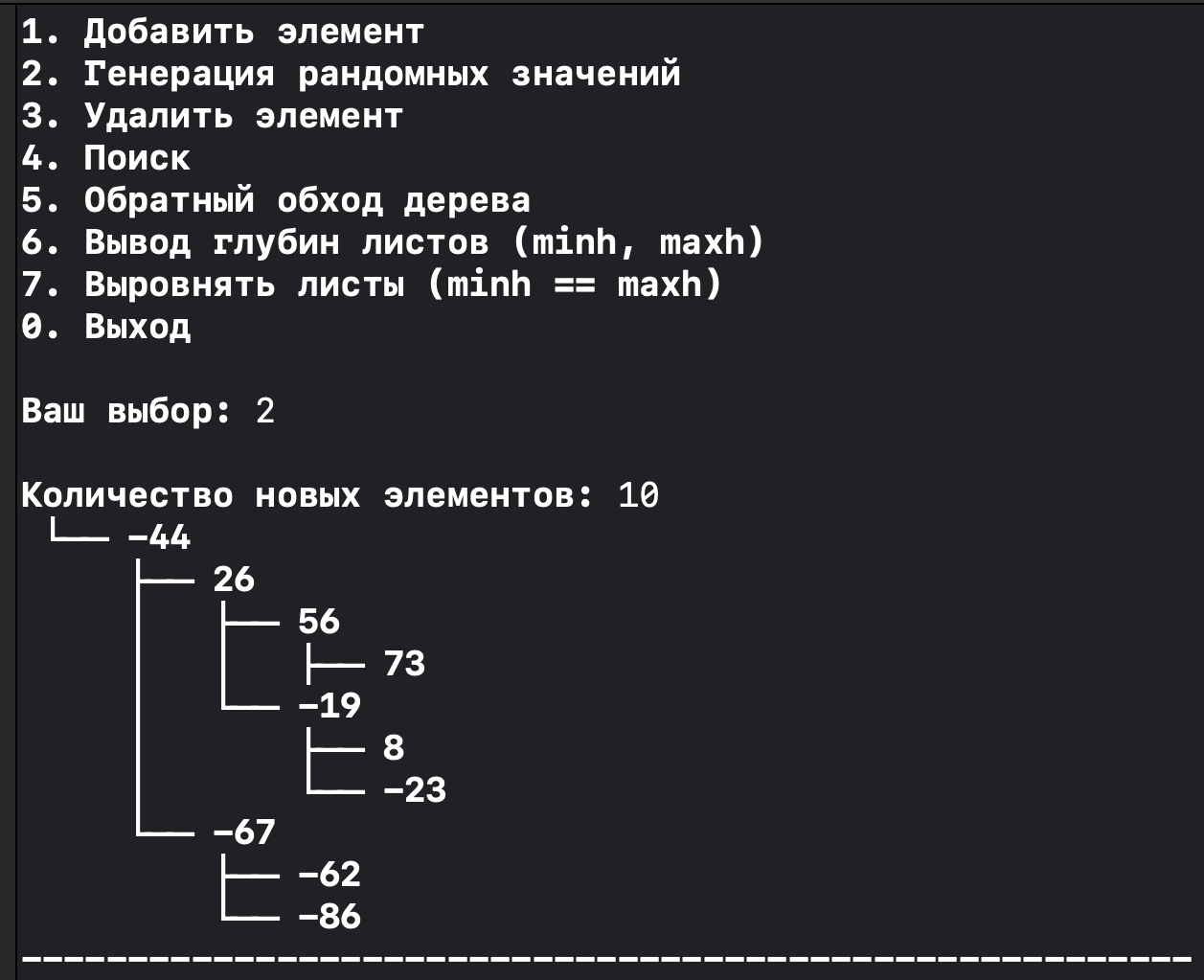
}

}

return 0;

}

1. ***Примеры***

******

1. ***Вывод***

В ходе лабораторной работы я изучил деревья поиска и получил практические навыки их использования