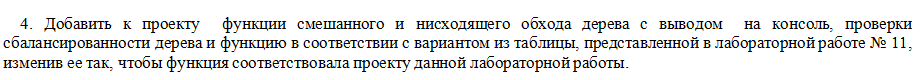
Банкузов Михаил 7 группа Лабораторная работа №12 Вариант 1



#include <iostream>

using namespace std;

struct Tree//дерево

{

int key;//ключ

int n1 = 0, n2 = 0, n3 = 0;

int sum = n1 + n2 + n3;

Tree\* Left, \* Right;// укащатели на левую и правую ветви

};

int sum = 0, result3 = 0, counter2 = 0;

Tree\* makeTree(Tree\* Root);//Создание дерева

Tree\* list(int i, int n1, int n2, int n3);//Создание нового элемента

Tree\* insertElem(Tree\* Root, int key, int n1, int n2, int n3);//Добавление нового элемента

Tree\* delet(Tree\* Root); //Удаление элемента по ключу

void view(Tree\* t, int level);//Вывод дерева

void delAll(Tree\* t);//Очистка дерева

int check\_left(Tree\* tree);// Подсчёт правых

int c = 0;//количество слов

Tree\* Root = NULL;//указатель корня

struct Node { // структура Node для представления узлов дерева

int data;

Node\* left;

Node\* right;

};

Node\* createNode(int value) { // функция createNode для создания новых узлов

Node\* newNode = new Node; // выделение памяти для нового узла

newNode->data = value; // инициализация данных нового узла

newNode->left = NULL;// инициализация указателей на дочерние узлы

newNode->right = NULL; // инициализация указателей на дочерние узлы

return newNode; // возврат указателя на новый узел

}

void insertNode(Node\*& p, int value) {// функция insertNode для вставки новых значений в дерево

if (p == NULL) {// если дерево пусто, то создаем новый узел

p = createNode(value); // инициализация указателей на дочерние узлы

}

else if (value < p->data) { // если значение меньше значения корня, то вставляем его в левое поддерево

insertNode(p->left, value);// инициализация указателей на дочерние узлы

}

else { // если значение больше значения корня, то вставляем его в правое поддерево

insertNode(p->right, value);

}

}

void traverseInOrder(Node\* p, int& sum) { // функция traverseInOrder для обхода дерева в порядке "влево-корень-вправо"

if (p != NULL) { // если дерево не пусто, то обходим его

traverseInOrder(p->left, sum); // обходим левое поддерево

sum += p->data; // суммируем значения всех узлов

traverseInOrder(p->right, sum); // обходим правое поддерево

}

}

void traverseeInOrderNoSum(Node\* p) { // функция traverseInOrder для обхода дерева в порядке "влево-корень-вправо"

if (p != NULL) { // если дерево не пусто, то обходим его

traverseeInOrderNoSum(p->left); // обходим левое поддерево

cout << p->data << " "; // выводим значение корня

traverseeInOrderNoSum(p->right); // обходим правое поддерево

}

}

void traversePostOrder(Node\* p) { // функция traversePostOrder для обхода дерева в порядке "влево-вправо-корень"

if (p != NULL) { // если дерево не пусто, то обходим его

traversePostOrder(p->left); // обходим левое поддерево

traversePostOrder(p->right); // обходим правое поддерево

cout << p->data << " ";// выводим значение корня

}

}

void traversePreOrder(Node\* p) { // функция traversePreOrder для обхода дерева в порядке "корень-влево-вправо"

if (p != NULL) { // если дерево не пусто, то обходим его

cout << p->data << " "; // выводим значение корня

traversePreOrder(p->left); // обходим левое поддерево

traversePreOrder(p->right); // обходим правое поддерево

}

}

int getHeight(Node\* p) { // функция getHeight для вычисления высоты дерева

if (p == NULL) { // если дерево пусто, то его высота равна 0

return 0; // иначе высота дерева равна максимальной высоте его поддеревьев + 1

}

else { // вычисляем высоту левого и правого поддеревьев

int leftHeight = getHeight(p->left); // вычисляем высоту левого поддерева

int rightHeight = getHeight(p->right); // вычисляем высоту правого поддерева

return max(leftHeight, rightHeight) + 1; // возвращаем максимальную высоту + 1

}

}

bool isBalanced(Node\* p) { // функция isBalanced для проверки дерева на сбалансированность

if (p == NULL) { // если дерево пусто, то оно сбалансировано

return true; // иначе вычисляем разницу высот левого и правого поддеревьев

}

else { // если разница высот больше 1, то дерево не сбалансировано

int leftHeight = getHeight(p->left); // вычисляем высоту левого поддерева

int rightHeight = getHeight(p->right); // вычисляем высоту правого поддерева

int heightDiff = abs(leftHeight - rightHeight); // вычисляем разницу высот

if (heightDiff > 1) { // если разница высот больше 1, то дерево не сбалансировано

return false;

}

else {

return isBalanced(p->left) && isBalanced(p->right); // проверяем левое и правое поддеревья на сбалансированность

}

}

}

int main()

{

setlocale(0, "Russian");

int key, choice, n, numLeft = 0;

Tree\* rc;

int n1, n2, n3;

Node\* root = NULL;

insertNode(root, 5);

insertNode(root, 3);

insertNode(root, 7);

insertNode(root, 2);

insertNode(root, 4);

insertNode(root, 6);

insertNode(root, 8);

int sum = 0;

traverseInOrder(root, sum);

cout << "Cумма всех корней: " << sum << endl;

cout << "смешанноый обход: ";

traverseeInOrderNoSum(root);

cout << endl;

cout << "нисходящего обход : ";

traversePreOrder(root);

cout << endl;

cout << "Обход после: ";

traversePostOrder(root);

cout << endl;

cout << "Высота дерева: " << getHeight(root) << endl;

if (isBalanced(root)) {

cout << "Дерево сбалансированно." << endl;

}

else {

cout << "Дерево не сбалансированно." << endl;

}

for (;;)

{

cout << "1 - создание дерева\n";

cout << "2 - добавление элемента\n";

cout << "3 - удаление элемента\n";

cout << "4 - вывод дерева\n";

cout << "5 - очистка дерева\n";

cout << "6 - количество правых дочерних вершин\n";

cout << "7 - выход\n";

cout << "ваш выбор?\n";

cin >> choice;

cout << "\n";

switch (choice)

{

case 1: Root = makeTree(Root); break;

case 2: cout << "\nВведите ключ: "; cin >> key;

cout << "Введите значение полей: "; cin >> n1 >> n2 >> n3;

insertElem(Root, key, n1, n2, n3); break;

case 3:

Root = delet(Root); break;

case 4: if (Root->key >= 0)

{

cout << "Дерево повернуто на 90 град. влево" << endl;

view(Root, 0);

}

else cout << "Дерево пустое\n"; break;

case 5: delAll(Root); break;

case 6: numLeft = check\_left(Root);

cout << "Количество правых дочерних вершин = " << numLeft << endl;

break;

case 7: exit(0);

}

}

}

Tree\* makeTree(Tree\* Root)//Создание дерева

{

int key; int n1, n2, n3;

cout << "Конец ввода - отрицательное число\n\n";

if (Root == NULL)// если дерево не создано

{

cout << "Введите ключ корня: "; cin >> key;

cout << "Введите значения полей: "; cin >> n1 >> n2 >> n3;

Root = list(key, n1, n2, n3);// установка указателя на корень

}

while (1)//добавление элементов

{

cout << "\nВведите ключ: "; cin >> key;

if (key < 0) break;//признак выхода (ключ < 0)

cout << "Введите значения полей: "; cin >> n1 >> n2 >> n3;

insertElem(Root, key, n1, n2, n3);

}

return Root;

}

Tree\* list(int i, int n1, int n2, int n3)//Создание нового элемента

{

Tree\* t = new Tree[sizeof(Tree)];// выделение памяти

t->key = i;// ключ

t->n1 = n1; t->n2 = n2; t->n3 = n3;

t->sum = n1 + n2 + n3;

t->Left = t->Right = NULL;// правая и левая ветви

return t;

}

Tree\* insertElem(Tree\* t, int key, int n1, int n2, int n3)//Добавление нового элемента

{

Tree\* Prev = NULL;// Prev - элемент перед текущим

int find = 0;// признак поиска

while (t && !find)

{

Prev = t;

if (key == t->key)

find = 1;//ключи должны быть уникальны

else

if (key < t->key) t = t->Left;

else t = t->Right;

}

if (!find)//найдено место с адресом Prev

{

t = list(key, n1, n2, n3);//создается новый узел

if (key < Prev->key)// если ключ меньше

Prev->Left = t;// то помещается на левую ветвь

else

Prev->Right = t;// иначе на правую

}

return t;

}

static void search\_min(Tree\* t, int& min\_sum, int& min\_key)

{

if (t)

{

search\_min(t->Right, min\_sum, min\_key); //вывод правого поддерева

if (t->sum < min\_sum) { min\_sum = t->sum; min\_key = t->key; }

search\_min(t->Left, min\_sum, min\_key); //вывод левого поддерева

}

}

Tree\* delet(Tree\* Root)

{

Tree\* Del, \* Prev\_Del, \* R, \* Prev\_R;

Del = Root;

int min\_sum = Del->sum;

int min\_key = Del->key;

search\_min(Root, min\_sum, min\_key);

Prev\_Del = NULL;

while (Del != NULL && Del->key != min\_key)//поиск элемента и его родителя

{

Prev\_Del = Del;

if (Del->key > min\_key)

Del = Del->Left;

else

Del = Del->Right;

}

if (Del == NULL)// элемент не найден

{

puts("\nНет такого ключа");

return Root;

}

if (Del->Right == NULL)// поиск элемента R для замены

R = Del->Left;// переход на левую ветвь

else

if (Del->Left == NULL)

R = Del->Right;// или на правую

else

{

Prev\_R = Del; //поиск самого правого элемента в левом поддереве

R = Del->Left;

while (R->Right != NULL)

{

Prev\_R = R;

R = R->Right;

}

if (Prev\_R == Del)// найден элемент для замены R и его родителя Prev\_R

R->Right = Del->Right;

else

{

R->Right = Del->Right;

Prev\_R->Right = R->Left;

R->Left = Prev\_R;

}

}

if (Del == Root) Root = R;//удаление корня и замена его на R

else

// поддерево R присоединяется к родителю удаляемого узла

if (Del->key < Prev\_Del->key)

Prev\_Del->Left = R;// на левую ветвь

else

Prev\_Del->Right = R;// на правую ветвь

int tmp = Del->key;

cout << "\nУдален элемент с ключом " << tmp << endl;

delete Del;

return Root;

}

void view(Tree\* t, int level)//Вывод дерева

{

if (t)

{

view(t->Right, level + 1);//вывод правого поддерева

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

cout << " ";

int tm = t->key;

cout << tm << ' ';

cout << t->n1 << ':' << t->n2 << ':' << t->n3;

cout << endl;

view(t->Left, level + 1);//вывод левого поддерева

}

}

void delAll(Tree\* t) //Очистка дерева

{

if (t != NULL)

{

delAll(t->Left);

delAll(t->Right);

delete t;

}

}

int check\_left(Tree\* tree)

{

if (tree->Right != NULL)

{

check\_left(tree->Right);

sum++;// подсчет количества листьев, которые являются правыми дочерними вершинами

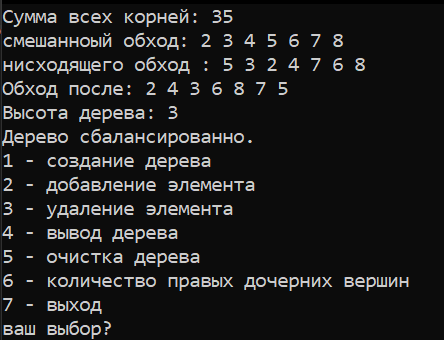
}

if (tree->Left != NULL)

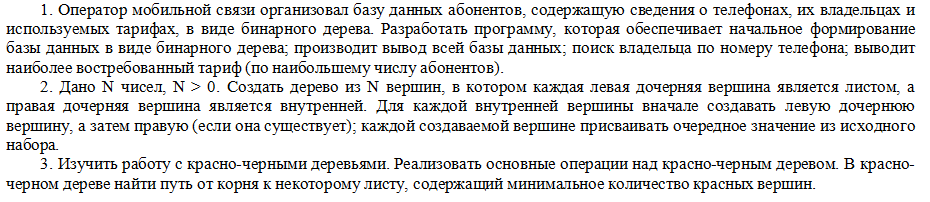
check\_left(tree->Left);

return sum;

}



Дополнительные задания



Задание 1

#include <iostream>

#include <Windows.h>

#include <vector>

#include <algorithm>

using std::endl; using std::cout; using std::cin; using std::string;

// структура для хранения данных об абоненте

struct Abonent

{

string number; // номер телефона

string own; // имя владельца

int trafic; // тариф

friend std::ostream& operator<<(std::ostream& os, Abonent ab); // перегрузка оператора вывода

};

// перегрузка оператора вывода для вывода данных об абоненте в консоль

std::ostream& operator<<(std::ostream& os, Abonent ab)

{

os << ab.own << '\t' << ab.number << '\t' << ab.trafic; // вывод имени владельца, номера телефона и тарифа

return os;

}

// структура для хранения данных об узле дерева

struct Node

{

Abonent abonent; // данные об абоненте

Node\* left, \* right; // указатели на левого и правого потомков

};

Node\* tree = nullptr; // указатель на корень дерева

// функция для ввода данных об абоненте с клавиатуры

Abonent include\_aboniment(Abonent& ab)

{

cout << "Введите телефонный номер: "; cin >> ab.number;

cout << "Введите имя владельца: "; cin >> ab.own;

cout << "Введите тариф: "; cin >> ab.trafic;

return ab; // возвращаем заполненную структуру

}

// функция для вставки узла в дерево

void insert(Abonent a, Node\*\* t)

{

if ((\*t) == NULL)

{

(\*t) = new Node;

(\*t)->abonent = a;

(\*t)->left = (\*t)->right = NULL;

return;

}

if (a.own > (\*t)->abonent.own)

{

insert(a, &(\*t)->left);

}

else insert(a, &(\*t)->right);

}

// функция для вывода дерева в консоль

void print(Node\* t)

{

if (t == NULL) return;

else

{

print(t->left); // выводим левого потомка

cout << t->abonent << '\t'; // выводим данные об абоненте

}

print(t->right); // выводим правого потомка

}

// функция для поиска узла по номеру телефона

void find\_number(Node\* t, string number, std::vector<int>& trafic)

{

if (t == NULL) return;

else

{

find\_number(t->left, number, trafic); // рекурсивный вызов функции для левого потомка

trafic.push\_back(t->abonent.trafic); // сохраняем тариф в вектор

if (t->abonent.number == number) // если номер телефона совпадает с искомым номером

{

cout << t->abonent << '\t'; // выводим данные об абоненте

}

}

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int count;

int max = 0;

int trafic = 0;

Abonent abonent;

string number;

std::vector<int> trafics;

cout << "Введите количество элементов "; cin >> count;

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

{

insert(include\_aboniment(abonent), &tree);

}

cout << endl;

cout << "ваше дерево\n";

print(tree);

cout << endl << "Введите номер телефона, по которому будет осуществлён поиск: "; cin >> number;

find\_number(tree, number, trafics);

cout << endl;

for (auto it : trafics)

{

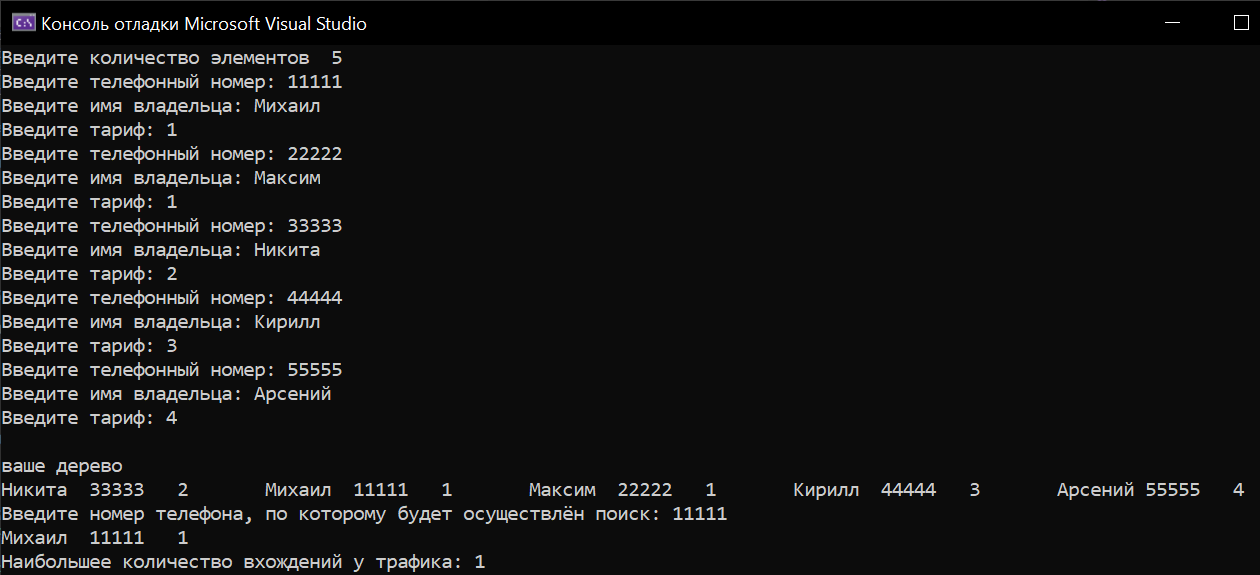
int m = std::count(trafics.begin(), trafics.end(), it);

if (m > max) { max = m; trafic = it; }

}

cout << "Наибольшее количество вхождений у трафика: " << trafic;

}



Задание 2

#include <iostream>

#include <ctime>

using std::endl; using std::cin; using std::cout;

//Структура узла дерева

struct Node

{

int data;

Node\* left, \* right;

};

//Указатель на корень дерева

Node\* tree;

//Функция для добавления элемента в дерево

void include(Node\*\* t, int data)

{

//Если дерево пустое, создаем новый корневой узел

if (\*t == NULL)

{

\*t = new Node;

(\*t)->data = data;

(\*t)->left = (\*t)->right = nullptr;

return;

}

//Если добавляемый элемент больше текущего, добавляем вправо, иначе влево

if ((\*t)->data < data)

{

include(&(\*t)->right, data);

}

else

{

//Если есть левый потомок, добавляем влево, иначе добавляем вправо

if (!(\*t)->left)

{

(\*t)->left = nullptr;

include(&(\*t)->left, data);

return;

}

else include(&(\*t)->right, data);

}

}

//Функция для вывода дерева на экран

void print(Node\* t, int u)

{

if (t == NULL) return;

else

{

//Рекурсивно обходим левое поддерево, увеличивая значение отступа

print(t->left, ++u);

//Выводим отступы и вертикальные линии до корня

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

cout << '\t';

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

cout << '|';

cout << t->data << endl;

//Уменьшаем значение отступа и обходим правое поддерево

u--;

}

print(t->right, ++u);

}

int main()

{

//Инициализация генератора случайных чисел текущим временем

srand((unsigned int)time(NULL));

//Добавление 10 случайных чисел в дерево

for (int i = 0; i < 10; i++)

{

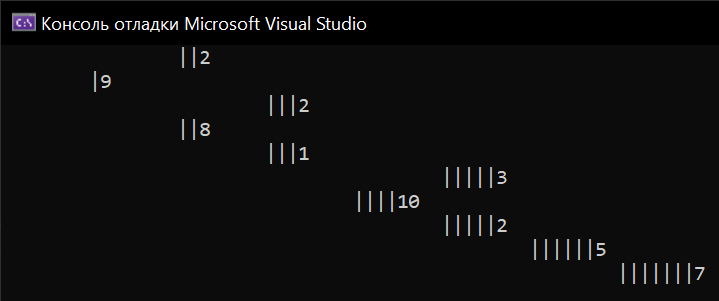
include(&tree, rand() % 10 + 1);

}

//Вывод дерева на экран, начиная с корня и отступом 0

print(tree, 0);

}



Задание 3

#include <vector>

#include <algorithm>

#include <iostream>

struct Node { // Вершина красно-черного дерева

int key; // Ключ

char color; // 'R' - red, 'B' - black

Node\* left; // Левый потомок

Node\* right; // Правый потомок

Node\* parent; // Родитель

};

enum Color { RED, BLACK }; // Цвета вершин

class RedBlackTree { // Красно-черное дерево

private:

void leftRotate(Node\* x) { // Выполняет левый поворот поддерева с корнем x

Node\* y = x->right; // Устанавливаем y

x->right = y->left; // Переносим левое поддерево y в правое поддерево x

if (y->left != nullptr) y->left->parent = x; // Устанавливаем родителя для x

y->parent = x->parent; // Устанавливаем родителя y

if (x->parent == nullptr) root = y; // Если x - корень, то y - корень

else if (x == x->parent->left) x->parent->left = y; // Если x - левый потомок, то y - левый потомок

else x->parent->right = y; // Если x - правый потомок, то y - правый потомок

y->left = x; // Устанавливаем x левым потомком y

x->parent = y; // Устанавливаем y родителем x

}

void rightRotate(Node\* x) { // Выполняет правый поворот поддерева с корнем x

Node\* y = x->left; // Устанавливаем y

x->left = y->right; // Переносим правое поддерево y в левое поддерево x

if (y->right != nullptr) y->right->parent = x; // Устанавливаем родителя для x

y->parent = x->parent; // Устанавливаем родителя y

if (x->parent == nullptr) root = y; // Если x - корень, то y - корень

else if (x == x->parent->right) x->parent->right = y; // Если x - правый потомок, то y - правый потомок

else x->parent->left = y; // Если x - левый потомок, то y - левый потомок

y->right = x; // Устанавливаем x правым потомком y

x->parent = y; // Устанавливаем y родителем x

}

public:

RedBlackTree() { // Конструктор

root = nullptr; // Инициализируем корень

}

Node\* minimum(Node\* node) { // Возвращает минимальный элемент в поддереве с корнем node

while (node->left != nullptr) node = node->left; // Двигаемся влево до конца

return node; // Возвращаем минимальный элемент

}

void insert(int key) { // Вставка элемента с ключом key

Node\* newNode = new Node{ key, RED, nullptr, nullptr, nullptr }; // Создаем новую вершину

if (root == nullptr) { // Если дерево пустое, то корнем становится новая вершина

root = newNode; // Устанавливаем корень

}

else {

Node\* cur = root; // Иначе начинаем с корня

Node\* parent; // Указатель на родителя

while (true) { // Двигаемся вниз по дереву

parent = cur; // Устанавливаем родителя

if (key < cur->key) { // Если ключ меньше ключа текущей вершины, то двигаемся влево

cur = cur->left; // Двигаемся влево

if (cur == nullptr) { // Если достигли конца, то вставляем новую вершину

parent->left = newNode; // Вставляем новую вершину

break; // Выходим из цикла

}

}

else { // Если ключ больше ключа текущей вершины, то двигаемся вправо

cur = cur->right; // Двигаемся вправо

if (cur == nullptr) { // Если достигли конца, то вставляем новую вершину

parent->right = newNode; // Вставляем новую вершину

break; // Выходим из цикла

}

}

}

newNode->parent = parent;

}

// Исправляем дерево

Node\* node = newNode;

while (node != root && node->parent->color == RED) {

Node\* parent = node->parent;

Node\* grandpa = parent->parent;

if (parent == grandpa->left) {

Node\* uncle = grandpa->right;

if (uncle != nullptr && uncle->color == RED) {

parent->color = BLACK;

uncle->color = BLACK;

grandpa->color = RED;

node = grandpa;

}

else {

if (node == parent->right) {

leftRotate(parent);

parent = node;

node = parent->left;

}

parent->color = BLACK;

grandpa->color = RED;

rightRotate(grandpa);

}

}

else {

// Симметричный случай

Node\* uncle = grandpa->left;

if (uncle != nullptr && uncle->color == RED) {

parent->color = BLACK;

uncle->color = BLACK;

grandpa->color = RED;

node = grandpa;

}

else {

if (node == parent->left) {

rightRotate(parent);

parent = node;

node = parent->right;

}

parent->color = BLACK;

grandpa->color = RED;

leftRotate(grandpa);

}

}

}

root->color = BLACK;

}

// Поиск пути с минимальным количеством красных вершин

std::vector<Node\*> findPath(Node\* node) {

if (node == root) return { root };

std::vector<Node\*> path;

Node\* cur = node;

while (cur != root) {

path.push\_back(cur);

Node\* parent = cur->parent;

if (parent->color == RED) cur = parent; // Двигаемся вверх по красной вершине

else if (cur == parent->left) { // Двигаемся вправо по черной вершине

cur = parent;

path.pop\_back();

}

else { // Двигаемся влево по черной вершине

Node\* child = cur;

cur = parent;

while (cur != root && cur->color == BLACK && cur == cur->parent->left) {

path.push\_back(cur);

child = cur;

cur = cur->parent;

}

if (cur->color == RED) path.push\_back(cur); // Встретили красную вершину

path.push\_back(child);

}

}

path.push\_back(root);

std::reverse(path.begin(), path.end());

return path;

}

Node\* root;

};

int main() {

RedBlackTree tree;

tree.insert(7);

tree.insert(3);

tree.insert(18);

tree.insert(2);

tree.insert(5);

tree.insert(4);

tree.insert(1);

tree.insert(8);

Node\* node = tree.minimum(tree.root->right); // Выберем любой лист

std::vector<Node\*> path = tree.findPath(node);

for (Node\* n : path) {

std::cout << n->key << (n->color == RED ? "(R) " : "(B) ");

}

std::cout << std::endl;

}

