|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ | | | | | | | | | | | | |
| Федеральное государственное автономное образовательное учреждение  высшего образования | | | | | | | | | | | | |
| **Дальневосточный федеральный университет** | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| **ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ** | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| **Кафедра прикладной математики, механики, управления и программного обеспечения** | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| **О Т Ч Е Т** | | | | | | | | | | | | |
| по лабораторной работе №1.2  дисциплина «Фундаментальные структуры данных и алгоритмы» | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | |  |  | Студент гр. Б9120-09.03.04прогин | | | |
|  |  |  | | О. Г. Александрович | |
|  | | | | | | |  |  | (подпись) | |  | |
|  | | | | | | | | | | | | |
|  | | | | | | |  |  | Руководитель | | |  |
|  | | | | | | |  |  | ст. преподаватель | | | |
|  | | | |  |  | |  |  |  |  | О.А. Крестникова | |
|  | | | |  |  | |  |  | (подпись) |  | (И.О. Фамилия) | |
|  |  |  |  | | |  |  |  |  | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| г. Владивосток | | | | | | | | | | | | |
| 2022 | | | | | | | | | | | | |

Оглавление

[Постановка задачи: 2](#_Toc93242413)

[Тесты: 3](#_Toc93242414)

[Общие тесты: 3](#_Toc93242415)

[1. Проверка, правильно ли программа заменяет на минимальный справа, при удалении: 3](#_Toc93242416)

[2. Проверка, правильно ли программа добавляет и удаляет одинаковые значения: 3](#_Toc93242417)

[Вставка: 3](#_Toc93242418)

[1 случай (Дядя красный): 3](#_Toc93242419)

[2 случай (Меняя направление вставки): 4](#_Toc93242420)

[3 случай (Не меняя направления вставки): 4](#_Toc93242421)

[Удаление: 4](#_Toc93242422)

[1 случай (брат красный): 4](#_Toc93242423)

[3 случай (Потомок брата красный): 5](#_Toc93242424)

[2 случай (Оба потомка брата чёрные): 6](#_Toc93242425)

[4 случай (правый потомок брата красный): 6](#_Toc93242426)

[Поиск наибольшего элемента: 7](#_Toc93242427)

[Обходы: 7](#_Toc93242428)

[Прямой, центрированный, и обратный: 7](#_Toc93242429)

[Фотографии моей тетради с интерпретацией и визуализацией (рисунками): 8](#_Toc93242430)

[Балансировки после добавления элемента: 8](#_Toc93242431)

[Балансировки после удаления элемента: 9](#_Toc93242432)

[Описание типа + Спецификация подпрограмм: заголовок (имя, список формальных параметров), назначение (что делает, входные/выходные данные) 9](#_Toc93242433)

[Файл main.cpp 9](#_Toc93242434)

[Заголовочный файл Node.h 10](#_Toc93242435)

[Файл RedBlackTree.h 11](#_Toc93242436)

# Постановка задачи:

Реализовать пакет подпрограмм для работы с бинарным деревом поиска (см. свой вариант задания), в котором элементы представлены в виде структуры (см. свой вариант задания).

Основные операции:

1. Создать

2. Очистить

3. Добавить

4. Удалить (min справа)

5. Найти

6. Найти max Выводим max.

7. Обход (прямой, обратный, симметричный (слева направо, справа налево))

Мой вариант задания:

* При удалении заменять на минимальный справа
* При повторах значений хранить count – счётчик повторов

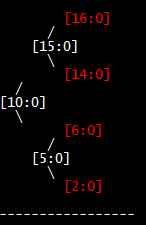
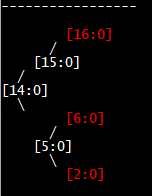
# Тесты:

## Общие тесты:

### 1. Проверка, правильно ли программа заменяет на минимальный справа, при удалении:

Добавляем: 10 5 15 2 6 14 16, Выводим

Удаляем: 10, Выводим



### 2. Проверка, правильно ли программа добавляет и удаляет одинаковые значения:

Добавляем: 10 10 10, Выводим

Удаляем: 10, Выводим

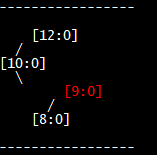
Изображение выглядит как текст, устройство, счетчик

Автоматически созданное описание

## Вставка:

### 1 случай (Дядя красный):

Добавляем: 10 8 12, Выводим

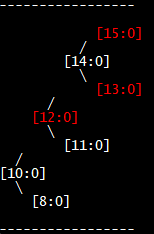
Добавляем: 9, Выводим

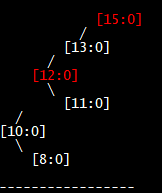
Изображение выглядит как текст, устройство, часы, счетчик

Автоматически созданное описание

### 2 случай (Меняя направление вставки):

Добавляем: 10 8 12 11 13 15, Выводим

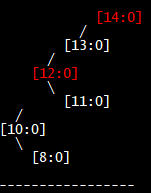
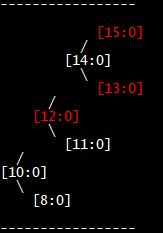
Добавляем 14, Выводим



### 3 случай (Не меняя направления вставки):

Добавляем: 10 8 12 11 13 14, Выводим

Добавляем: 15, Выводим



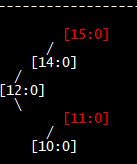
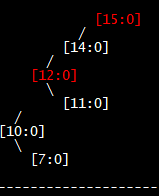
## Удаление:

### 1 случай (брат красный):

Добавляем: 10 8 12 7 11 14 15

Удаляем: 8, Выводим

Удаляем: 7, Выводим

### 3 случай (Потомок брата красный):

Добавляем: 10 20 5 15 30 40 2 3 1 25 45 41 43, Выводим

Удаляем: 40, Выводим

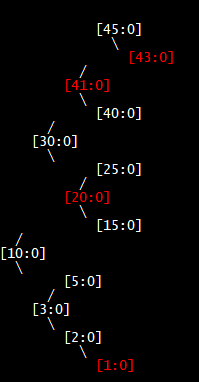
(с 3го случая идёт на 2й)

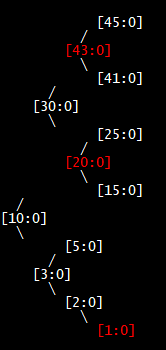
Изображение выглядит как текст, устройство

Автоматически созданное описаниеУжатая версия:

Изображение выглядит как текст

Автоматически созданное описание

Полная версия:

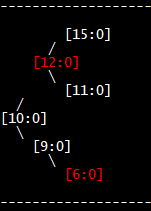


### 2 случай (Оба потомка брата чёрные):

Добавляем последовательно ключи от 15 до 6

Удаляем: 14 13 8, Выводим

Удаляем 7, Выводим

Изображение выглядит как карта

Автоматически созданное описание

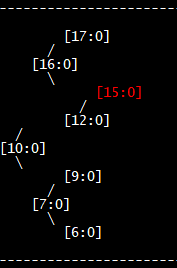
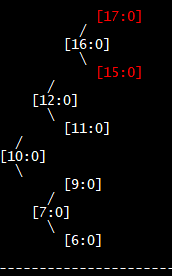
### 4 случай (правый потомок брата красный):

Добавляем последовательно ключи от 15 до 6

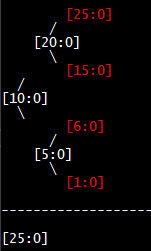
Удаляем: 14 13 8

Добавляем 16 17, Выводим

Удаляем: 11, Выводим

## Поиск наибольшего элемента:



## Обходы:

### Прямой, центрированный, и обратный:

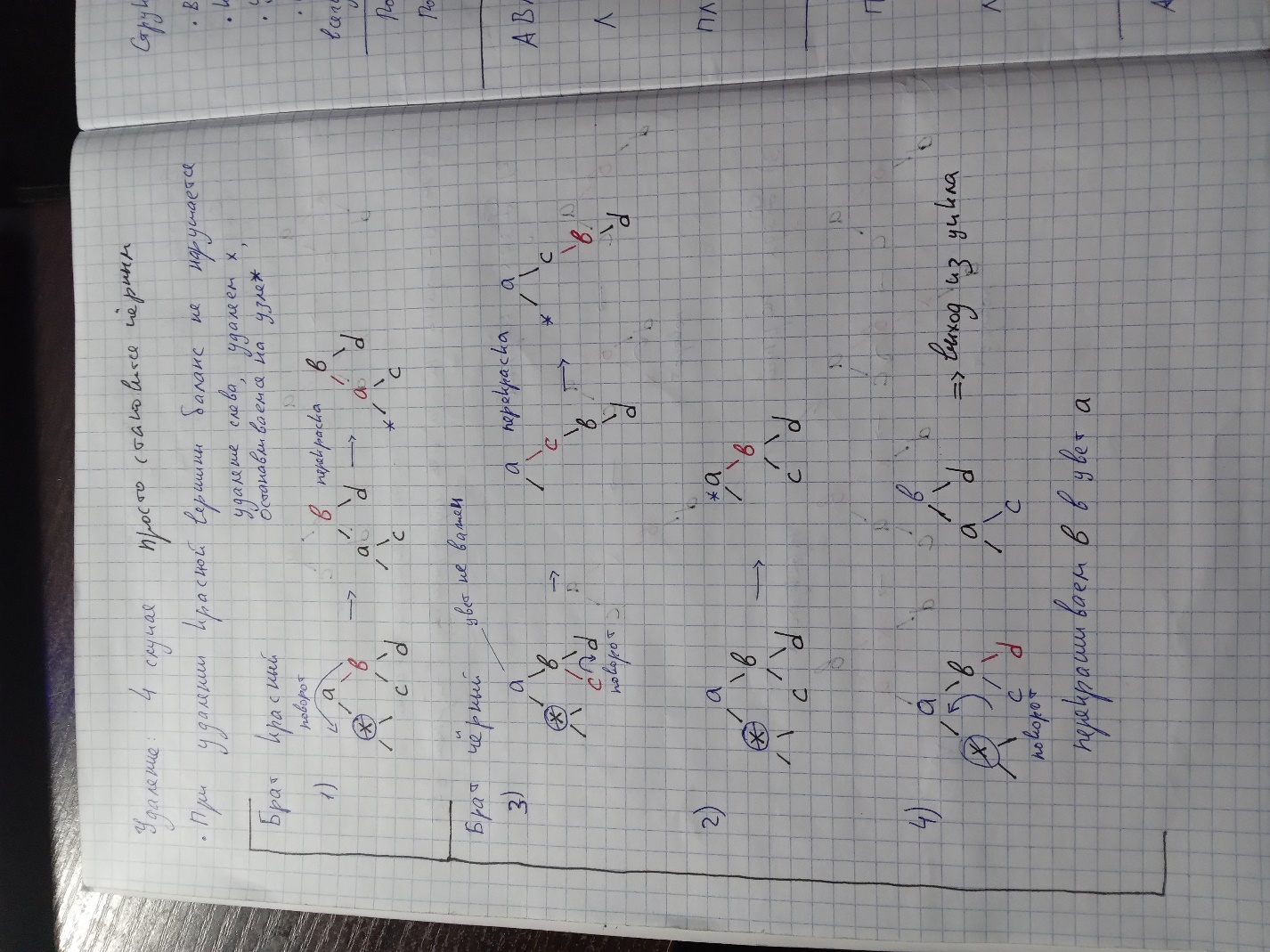
Изображение выглядит как текст

Автоматически созданное описание

## Фотографии моей тетради с интерпретацией и визуализацией (рисунками):

### Балансировки после добавления элемента:

### Балансировки после удаления элемента:



# Описание типа + Спецификация подпрограмм: заголовок (имя, список формальных параметров), назначение (что делает, входные/выходные данные)

## Файл main.cpp

#include <iostream>

#include "RedBlackTree.h"

using namespace std;

float random()

{

float t = (rand() % 26 + 0); // 13

return(t);

}

void main()

{

srand(time(0));

Time time;

time.minutes = 0;

cout << endl;

RBTree Tree1;

setlocale(LC\_ALL, "rus");

/\* // Ввод 1

time.hour = 10; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 5; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 20; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 1; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 6; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 15; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 25; Tree1.Insert(time, Tree1.TNill); // Добавляем

\*/

/\* // Ввод 2

for (int i = 15; i > 5; i--)

{

time.hour = i; Tree1.Insert(time, Tree1.TNill); // Добавляем

}

\*/

/\* // Ввод 3

time.hour = 10; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 5; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 20; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 1; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 6; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 15; Tree1.Insert(time, Tree1.TNill); // Добавляем

time.hour = 25; Tree1.Insert(time, Tree1.TNill); // Добавляем

cout << endl; Tree1.PrintTree(Tree1.root, 0, Tree1.TNill); cout << endl; // Печатаем

cout << "------------------------------------------" << endl;

time.hour = 10; Tree1.Delete(time, &Tree1, Tree1.TNill); // Удаляем

cout << endl; Tree1.PrintTree(Tree1.root, 0, Tree1.TNill); cout << endl; // Печатаем

cout << "------------------------------------------" << endl;

time.hour = 15; Tree1.Delete(time, &Tree1, Tree1.TNill); // Удаляем

cout << endl; Tree1.PrintTree(Tree1.root, 0, Tree1.TNill); cout << endl; // Печатаем

cout << "------------------------------------------" << endl;

time.hour = 20; Tree1.Delete(time, &Tree1, Tree1.TNill); // Удаляем

cout << endl; Tree1.PrintTree(Tree1.root, 0, Tree1.TNill); cout << endl; // Печатаем

cout << "------------------------------------------" << endl;

time.hour = 5; Tree1.Delete(time, &Tree1, Tree1.TNill); // Удаляем

cout << endl; Tree1.PrintTree(Tree1.root, 0, Tree1.TNill); cout << endl; // Печатаем

cout << "------------------------------------------" << endl;

time.hour = 6; Tree1.Delete(time, &Tree1, Tree1.TNill); // Удаляем

cout << endl; Tree1.PrintTree(Tree1.root, 0, Tree1.TNill); cout << endl; // Печатаем

cout << "------------------------------------------" << endl;

Tree1.DeleteRBTree(&Tree1, Tree1.root);

cout << endl; Tree1.PrintTree(Tree1.root, 0, Tree1.TNill); cout << endl; // Печатаем

cout << "------------------------------------------" << endl;

\*/

/\* // Ввод 4

//Бешеный тестировщик v1.0

//for (int i = 0; i < 30; i++)

//{

// int r = random();

// cout << "Добавили вершину " << r << endl;

// time.hour = r; Tree1.Insert(time, Tree1.TNill); // Добавляем

// //time.hour = i; Tree1.Insert(time, Tree1.TNill); // Добавляем

// cout << endl; Tree1.PrintTree(Tree1.root, 0, Tree1.TNill); cout << endl; // Печатаем

// cout << "------------------------------------------" << endl;

//}

//for (int i = 0; i < 7; i++) cout << endl;

//for (int i = 0; i < 100; i++)

//{

// int r = random();

// cout << "Удалили вершину " << r << endl;

// time.hour = r; Tree1.Delete(time, &Tree1, Tree1.TNill); // Удаляем

// cout << endl; Tree1.PrintTree(Tree1.root, 0, Tree1.TNill); cout << endl; // Печатаем

// cout << "------------------------------------------" << endl;

//}

\*/

//time.hour = 8; Tree1.Delete(time, &Tree1, Tree1.TNill); // Удаляем

//time.hour = 16; Tree1.Insert(time, Tree1.TNill); // Добавляем

//cout << endl; Tree1.PrintTree(Tree1.root, 0); cout << endl; // Печатаем

//cout << "------------------------------------------" << endl;

}

## Заголовочный файл Node.h

#pragma once

enum class Color { Red = 4, White = 15 };

struct Time

{

public:

int hour;

int minutes;

Time(int hour = -1, int minutes = -1)

{

this->hour = hour;

this->minutes = minutes;

}

};

class Node

{

public:

Color color;

Node\* parent;

Node\* leftchild;

Node\* rightchild;

Time key;

int count;

Node(Time key = Time(),

Node\* parent = nullptr,

Node\* leftchild = nullptr,

Node\* rightchild = nullptr,

Color color = Color::Red,

int count = 1)

{

this->key = key;

this->parent = parent;

this->leftchild = leftchild;

this->rightchild = rightchild;

this->color = color;

this->count = count;

}

};

## Файл RedBlackTree.h

#pragma once

#include "Node.h"

#include "RedBlackTree.h"

#include <iostream>

#include <Windows.h>

#include <string>

#include <sstream>

#include <iostream>

#include <algorithm>

using namespace std;

using namespace std;

class RBTree

{

public:

Node\* root;

Node\* TNill;

RBTree();

~RBTree();

void DeleteRBTree(Node\* node);

void PrintTree(Node\* r, int space = 0);

void PrintPreOrder(Node\* r);

void PrintInOrder(Node\* r);

void PrintPostOrder(Node\* r);

void Insert(Time key, Node\* Tnill);

void FixupInsert(Node\* node, Node\* Tnill);

Node\* FindMinOnRight(Node\* node, Node\* Tnill);

Node\* FindNode(Time key, Node\* Tnill);

void Transplant(RBTree\* T, Node\* u, Node\* v);

void Delete(Time key, RBTree\* T, Node\* Tnill);

void FixupDelete(Node\* node, Node\* Tnill);

void RightRotate(Node\* node, Node\* Tnill);

void LeftRotate(Node\* node, Node\* Tnill);

void SetColor(int text, int background);

Node\* FindMax(Node\* Tnill);

};

// Описание: Конструктор дерева. Создаётся указатель на корень и Tnill

RBTree::RBTree()

{

root = new Node();

TNill = new Node();

TNill->color = Color::White;

root = TNill;

}

// Описание: Также пустой конструктор дерева

RBTree::~RBTree()

{

}

// Описание: Полностью удаляет всё дерево

// Принимает: Указатель на вершину (корень)

void RBTree::DeleteRBTree(Node\* node)

{

if (node != TNill)

{

DeleteRBTree(node->leftchild);

DeleteRBTree(node->rightchild);

if (node == root)

{

delete TNill;

delete node;

}

else

delete node;

}

}

#pragma warning(disable : 4996)

// Описание: Функция, для преобразования int в string

// Принимает: число

// Возвращает: строку

template <class T>

string NumberToString(T param)

{

string str = "";

stringstream ss;

ss << param;

getline(ss, str);

return str;

}

// Описание: Функция для печати элемента с типом время

// Принимает: указатель на элемент типа Time

// Возвращает: строку (с часами и минутами)

string printTime(Time value) // Функция печати времени. Принимает указатель на элемент, и возвращет строку.

{

//std::string s = std::to\_string(123);

string c1 = "[";

string c2 = ":";

string c3 = "]";

string str2 = c1 + NumberToString(value.hour) + c2 + NumberToString(value.minutes) + c3;

return(str2);

}

// Описание: Эта процедура устанавливает нужный цвет текста и фона. Так я выделяю красные и чёрные элементы дерева

// Принимает: Нужный цвет текста, и фона (число от 0, до 15)

void RBTree::SetColor(int text, int background) //

{

HANDLE hStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

SetConsoleTextAttribute(hStdOut, (WORD)((background << 4) | text));

}

// Описание: Поцедура вывода дерева слева направо, с ветками и цветными элементами

// Принимает: Указатель на корень, текущий уровень вывода (по умолчанию 0)

void RBTree::PrintTree(Node\* p, int indent)

{

if ((p != NULL) && ((p->key.hour != -1) || (p->key.minutes != -1)))

// ((p->value.hour != 0) || (p->value.min != 0)))

{

if (p->rightchild)

{

PrintTree(p->rightchild, indent + 4);

}

if (indent)

{

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

{

cout << ' ';

}

//std::cout << std::setw(indent) << ' ';

}

if ((p->rightchild) && ((p->rightchild->key.hour != -1) || (p->rightchild->key.minutes != -1)))

{

cout << " /\n";

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

{

cout << ' ';

}

// std::cout << " /\n" << std::setw(indent) << ' ';

}

//if

{

if (p->color == Color::Red)

SetColor(12, 0);

else

SetColor(15, 0);

if (p->count < 2)

{

cout << printTime(p->key) << "\n ";

}

else

{

cout << printTime(p->key);

SetColor(2, 0);

cout << " count = " << p->count << "\n ";

SetColor(15, 0);

}

SetColor(15, 0);

}

if ((p->leftchild) && ((p->leftchild->key.hour != -1) || (p->leftchild->key.minutes != -1)))

{

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

{

cout << ' ';

}

cout << " \\\n";

//std::cout << std::setw(indent) << ' ' << " \\\n";

PrintTree(p->leftchild, indent + 4);

}

}

}

// Описание: Рекурсивный центрированный обход и вывод элементов дерева

// Принимает: Указатель на корень

void RBTree::PrintPreOrder(Node\* r)

{

if (r == TNill)

return;

cout << "[" << r->key.hour << ":" << r->key.minutes << "]" << " ";

PrintPreOrder(r->leftchild);

PrintPreOrder(r->rightchild);

}

// Описание: Рекурсивный прямой обход и вывод элементов дерева (от наименьшего к наибольшему)

// Принимает: Указатель на корень

void RBTree::PrintInOrder(Node\* r)

{

if (r == TNill)

return;

PrintInOrder(r->leftchild);

cout << "[" << r->key.hour << ":" << r->key.minutes << "]" << " ";

PrintInOrder(r->rightchild);

}

// Описание: Рекурсивный обратный центрированный обход и вывод элементов дерева

// Принимает: Указатель на корень

void RBTree::PrintPostOrder(Node\* r)

{

if (r == TNill)

return;

PrintPostOrder(r->leftchild);

PrintPostOrder(r->rightchild);

cout << "[" << r->key.hour << ":" << r->key.minutes << "]" << " ";

}

// Описание: Рекурсивная вставка

// Принимает: Минуты, часы и указатель на корень

void RBTree::Insert(Time key, Node\* Tnill)

{

Node\* x = root;

Node\* prnt = TNill;

//if (true) // (key.hour < 24 && key.hour >= 0 && key.minutes >= 0 && key.minutes < 60) // Нам не нужны эти условия, при исполнении тестов

{

Time t;

t.hour = key.hour; t.minutes = key.minutes;

Node\* newx = new Node(t);

while (x != TNill)

{

prnt = x;

if (key.hour < x->key.hour)

x = x->leftchild;

else if (key.hour > x->key.hour)

x = x->rightchild;

else

{

if (key.minutes < x->key.minutes)

x = x->leftchild;

else if (key.minutes > x->key.minutes)

x = x->rightchild;

else

{

x->count++;

return;

}

}

}

newx->parent = prnt;

if (prnt == TNill)

root = newx;

else if (key.hour == prnt->key.hour)

{

if (key.minutes < prnt->key.minutes)

prnt->leftchild = newx;

else

prnt->rightchild = newx;

}

else if (key.hour < prnt->key.hour)

prnt->leftchild = newx;

else

prnt->rightchild = newx;

newx->leftchild = TNill;

newx->rightchild = TNill;

newx->color = Color::Red;

FixupInsert(newx, TNill);

}

}

// Описание: Балансировка после вставки

// Принимает: Указатель на балансируемую вершину, Tnill

void RBTree::FixupInsert(Node\* node, Node\* Tnill)

{

while (node->parent->color == Color::Red)

{

if (node->parent == node->parent->parent->leftchild)

{

if (node->parent->parent->rightchild->color == Color::Red)

{

node->parent->color = Color::White;

node->parent->parent->rightchild->color = Color::White;

node->parent->parent->color = Color::Red;

node = node->parent->parent;

}

else

{

if (node == node->parent->rightchild)

{

node = node->parent;

LeftRotate(node, TNill);

}

node->parent->color = Color::White;

node->parent->parent->color = Color::Red;

RightRotate(node->parent->parent, TNill);

}

}

else

{

if (node->parent->parent->leftchild->color == Color::Red)

{

node->parent->color = Color::White;

node->parent->parent->leftchild->color = Color::White;

node->parent->parent->color = Color::Red;

node = node->parent->parent;

}

else

{

if (node == node->parent->leftchild)

{

node = node->parent;

RightRotate(node, TNill);

}

node->parent->color = Color::White;

node->parent->parent->color = Color::Red;

LeftRotate(node->parent->parent, TNill);

}

}

}

root->color = Color::White;

}

// Описание: Поиск минимального справа элемента, для замены, при удалении

// Принимает: Указатель на вершину, Tnill

// Возвращает: Минимальный справа

Node\* RBTree::FindMinOnRight(Node\* node, Node\* Tnill)

{

while (node->leftchild != Tnill)

{

node = node->leftchild;

}

return node;

}

// Описание: Поиск вершины по значению

// Принимает: Значение (время), Tnill

// Возвращает: Найденную вершину, или указатель на Tnill

Node\* RBTree::FindNode(Time key, Node\* Tnill)

{

Node\* curnode = root;

while (curnode != Tnill)

{

if (key.hour < curnode->key.hour)

curnode = curnode->leftchild;

else if (key.hour > curnode->key.hour)

curnode = curnode->rightchild;

else if (key.minutes < curnode->key.minutes)

curnode = curnode->leftchild;

else if (key.minutes > curnode->key.minutes)

curnode = curnode->rightchild;

else

return curnode;

}

}

// Описание: Заменяет поддерево корнем в поддереве узла node1 на корень в узле node2,

// родительский узел узла node1 становится родительским узлом узла node2, который

// становится соответствующим дочерним узлом родительского узла по отношению к узлу узла 1

void RBTree::Transplant(RBTree\* T, Node\* u, Node\* v)

// Указатель на экземпляр дерева, u и v

{

if (u->parent == T->TNill)

{

T->root = u;

}

else if (u == u->parent->leftchild)

{

u->parent->leftchild = v;

}

else

{

u->parent->rightchild = v;

}

v->parent = u->parent;

}

// Описание: Удаляет узел с указанным ключом в дереве и вызывает функцию Fixup Delete(),

// если узел удален, то возвращает true иначе возвращает false

void RBTree::Delete(Time key, RBTree\* T, Node\* Tnill) // T - указатель на корень

{

Node\* z = FindNode(key, Tnill); // z - указатель на удаляемую вершину

Node\* x = z; ///

if ((z != Tnill) && (z != nullptr))

{

if (z->count > 1)

{

z->count = z->count - 1;

}

else

{

Node\* y = z;

Color originalColor = y->color;

//cout << endl; cout << endl; PrintTree(z, 0); cout << endl; cout << endl; // Печатаем

if ((z->leftchild == Tnill) && (z->rightchild == Tnill) && (z->parent == Tnill))

{

//if(z->parent != nullptr) DeleteRBTree(z->parent);

//z->parent = Tnill;

//T->root = Tnill;

DeleteRBTree(T, z);

//exit;

}

else if (z->leftchild == Tnill)

{

x = z->rightchild;

Transplant(T, z, z->rightchild);

}

else if (z->rightchild == Tnill)

{

x = z->leftchild;

Transplant(T, z, z->leftchild);

}

else

{

y = FindMinOnRight(z->rightchild, Tnill);

originalColor = y->color;

x = y->rightchild;

if (y->parent == z)

x->parent = y;

else

{

Transplant(T, y, y->rightchild);

y->rightchild = z->rightchild;

y->rightchild->parent = y;

}

Transplant(T, z, y);

y->leftchild = z->leftchild;

y->leftchild->parent = y;

y->color = z->color;

}

if (x->parent == TNill)

{

T->root = x;

}

else if (y->parent == TNill)

{

T->root = y;

}

if (originalColor == Color::White)

{

if (T->root != T->TNill)

{

//if(y->parent == TNill)

// //T->root = y;

//if (!((y->leftchild == Tnill) && (y->rightchild == Tnill)))

{

FixupDelete(x, T->TNill);

}

//else if(x->parent != TNill) FixupDelete(x->parent, T->TNill);

}

//if((x != Tnill) && (x != nullptr))

}

}

}

}

// Описание: Балансирует дерево и исправляет цвета узлов дерева, после удаления узла

// Принимает: Указатель на балансируемую вершину, Tnill

void RBTree::FixupDelete(Node\* node, Node\* Tnill)

{

while (node != root && node->color == Color::White)

{

if (node == node->parent->leftchild)

{

Node\* auxnode = node->parent->rightchild;

if (auxnode->color == Color::Red)

{

auxnode->color = Color::White;

node->parent->color = Color::Red;

LeftRotate(node->parent, Tnill);

auxnode = node->parent->rightchild;

}

if (auxnode->leftchild->color == Color::White && auxnode->rightchild->color == Color::White)

{

auxnode->color = Color::Red;

node = node->parent;

}

else

{

if (auxnode->rightchild->color == Color::White)

{

auxnode->leftchild->color = Color::White;

auxnode->color = Color::Red;

RightRotate(auxnode, Tnill);

auxnode = node->parent->rightchild;

}

auxnode->color = node->parent->color;

node->parent->color = Color::White;

auxnode->rightchild->color = Color::White;

LeftRotate(node->parent, Tnill);

node = root;

}

}

else

{

Node\* auxnode = node->parent->leftchild;

if (auxnode->color == Color::Red)

{

auxnode->color = Color::White;

node->parent->color = Color::Red;

RightRotate(node->parent, Tnill);

auxnode = node->parent->leftchild;

}

if (auxnode->rightchild->color == Color::White && auxnode->leftchild->color == Color::White)

{

auxnode->color = Color::Red;

node = node->parent;

}

else

{

if (auxnode->leftchild->color == Color::White)

{

auxnode->rightchild->color = Color::White;

auxnode->color = Color::Red;

LeftRotate(auxnode, Tnill);

auxnode = node->parent->leftchild;

}

auxnode->color = node->parent->color;

node->parent->color = Color::White;

auxnode->leftchild->color = Color::White;

RightRotate(node->parent, Tnill);

node = root;

}

}

}

node->color = Color::White;

}

// Описание: Вращение вправо

// Принимает: Вершину, находящуюся посередине и Tnill

// Возвращает: Вершину, находящуюся сверху

void RBTree::RightRotate(Node\* node, Node\* Tnill)

{

Node\* auxnode = node->leftchild;

node->leftchild = auxnode->rightchild;

if (auxnode->rightchild != Tnill)

auxnode->rightchild->parent = node;

auxnode->parent = node->parent;

if (node->parent == Tnill)

root = auxnode;

else if (node == node->parent->rightchild)

node->parent->rightchild = auxnode;

else

node->parent->leftchild = auxnode;

auxnode->rightchild = node;

node->parent = auxnode;

}

// Описание: Вращение влево

// Принимает: Вершину, находящуюся посередине и Tnill

// Возвращает: Вершину, находящуюся сверху

void RBTree::LeftRotate(Node\* node, Node\* Tnill)

{

Node\* auxnode = node->rightchild;

node->rightchild = auxnode->leftchild;

if (auxnode->leftchild != Tnill)

auxnode->leftchild->parent = node;

auxnode->parent = node->parent;

if (node->parent == Tnill)

root = auxnode;

else if (node == node->parent->leftchild)

node->parent->leftchild = auxnode;

else

node->parent->rightchild = auxnode;

auxnode->leftchild = node;

node->parent = auxnode;

}

// Описание: Поиск максимального элемента

// Принимает: Tnill

// Возвращает: Максимальный элемент

Node\* RBTree::FindMax(Node\* Tnill)

{

Node\* curnode = root;

if (curnode != Tnill)

{

while (curnode->rightchild != Tnill)

curnode = curnode->rightchild;

return curnode;

}

else

return curnode; // Это возращение TNill, в случае не найденного элемента

}