

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
Дальневосточный федеральный университет

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Департамент программной инженерии и искусственного интеллекта

О Т Ч Е Т
по лабораторной работе №1.2
дисциплина «Фундаментальные структуры данных и алгоритмы»

Студент гр. Б9123-09.03.04прогин
_____ Д.И. Комаров
(подпись)

Преподаватель _____
ассистент _____
_____ А.А. Шулятьев
(подпись) (И.О. Фамилия)

г. Владивосток
2024

Оглавление

1. Неформальная постановка задачи	3
2. Описание типа + спецификация подпрограмм + тесты	4
3. Текст программы.....	12

1. Неформальная постановка задачи

Реализовать пакет подпрограмм для работы с бинарным деревом поиска – Красно-черное, ключ в котором представлен в виде структуры серии и номера паспорта: первое поле – серия (4 цифры) второе поле – номер (6 цифр).

Основные операции:

1. Инициализация (пустого дерева)
2. Добавление нового элемента
3. Удаление заданного элемента (при полном совпадении ключа)
4. Поиск заданного элемента
5. Печать
6. Обход (pre-ordered)
7. Освобождение памяти (удаление всего дерева)

2. Описание типа + спецификация подпрограмм + тесты

Подпрограмма `main` файла:

void menu();

Описание: функция `menu()` представляет собой вывод на экран меню для взаимодействия с программой. Она используется для отображения списка возможных действий, доступных пользователю.

Перечисляемый тип данных:

enum Color;

Описание: перечисляемый тип для цвета узла.

Поля: RED – красный, BLACK – черный.

Структуры:

struct Passport;

Описание: структура, описывающая паспорт.

Поля: `int series` – хранит серию паспорта, `int passport` – номер паспорта, `int line` – номер строки в файле.

struct Node;

Описание: структура, описывающая узел дерева.

Поля: `Passport passport` – паспорт, `Color color` – цвет узла, `Node* parent` – отец узла, `Node* right` – правый узел узла, `Node* left` – левый узел узла, `DoublyLinkedList DuplicateList` – список для дубликатов.

struct lNode;

Описание: структура, описывающая узел списка.

Поля: `int data` – информация в узле, `lNode* next` – следующий узел, `lNode* prev` – предыдущий узел.

Классы:

class DoublyLinkedList;

Описание: класс для двусвязного списка.

Поля: `lNode* head` – голова списка, `lNode* tail` – хвост списка.

class RBtree;

Описание: класс для красно-черного дерева.

Поля: `Node* root` – корень красно-черного дерева.

Методы класса `class DoublyLinkedList`:

DoublyLinkedList::~DoublyLinkedList();

Описание: метод – деструктор для двусвязного списка.

Входные данные: объект класса `DoublyLinkedList`.

Выходные данные: освобождение памяти, указатели `head` и `tail` устанавливаются в `nullptr`.

DoublyLinkedList::void push_back(int value):

Описание: метод для добавления в конец списка.

Входные данные: объект класса DoublyLinkedList и int value – значение.

Выходные данные: объект класса DoublyLinkedList с добавленным узлом.

DoublyLinkedList::void print():

Описание: метод для вывода списка.

Входные данные: объект класса DoublyLinkedList.

Выходные данные: объект класса DoublyLinkedList выводится на экран.

DoublyLinkedList::void delete_value(int value):

Описание: метод для удаления узла по значению.

Входные данные: объект класса DoublyLinkedList и int value – значение узла.

Выходные данные: объект класса DoublyLinkedList без указанного узла.

Методы класса class RBtree:

RBtree::void leftRotate(Node& root, Node* x):*

Описание: выполняет левый поворот вокруг узла x в красно-черном дереве.

Входные данные: Node*& root — ссылка на указатель корня дерева, Node* x — указатель на узел, вокруг которого выполняется поворот.

Выходные данные: модифицированное дерево с обновленным балансом после левого поворота.

RBtree::void rightRotate(Node& root, Node* y):*

Описание: метод выполняет правый поворот вокруг узла y в красно-черном дереве.

Входные данные: Node*& root — ссылка на указатель корня дерева, Node* y — указатель на узел, вокруг которого выполняется поворот.

Выходные данные: модифицированное дерево с обновленным балансом после правого поворота.

RBtree::void fixInsert(Node& root, Node* z):*

Описание: метод, который исправляет нарушения свойств красно-черного дерева после добавления нового узла z.

Входные данные: Node*& root — ссылка на указатель корня дерева, Node* z — указатель на узел, который был вставлен.

Выходные данные: модифицированное дерево, соответствующее свойствам красно-черного дерева.

RBtree::void transplant(Node& root, Node* u, Node* v):*

Описание: заменяет поддерево с корнем u поддеревом с корнем v.

Входные данные: Node*& root — ссылка на указатель корня дерева, Node* u — узел, который заменяется, Node* v — узел, который занимает место u.

Выходные данные: обновленное дерево, где поддерево u заменено на v.

RBtree::Node maximum(Node* node):*

Описание: возвращает узел с максимальным значением в заданном поддереве.

Входные данные: Node* node — указатель на корень поддерева.

Выходные данные: указатель на узел с максимальным значением.

RBtree::void deleteNode(Node& root, Node* z):*

Описание: удаляет узел *z* из красно-черного дерева, сохраняя его свойства.

Входные данные: *Node*& root* — ссылка на указатель корня дерева, *Node* z* — указатель на узел, который нужно удалить.

Выходные данные: модифицированное дерево с восстановленным балансом после удаления.

RBtree::void fixDelete(Node& root, Node* x):*

Описание: исправляет нарушения свойств красно-черного дерева после удаления узла.

Входные данные: *Node*& root* — ссылка на указатель корня дерева, *Node* x* — узел, который может нарушать свойства дерева.

Выходные данные: модифицированное дерево с восстановленным балансом.

RBtree::void printHelper(Node root, int space):*

Описание: рекурсивно печатает дерево в древовидной форме.

Входные данные: *Node* root* — указатель на корень дерева, *int space* — отступ для корректного форматирования вывода.

Выходные данные: печать дерева в консоль.

RBtree::void print_pre_order(Node root):*

Описание: печатает дерево в порядке прямого обхода.

Входные данные: *Node* root* — указатель на корень дерева.

Выходные данные: список узлов в порядке прямого обхода, напечатанный в консоль.

RBtree::void exportToGraphviz(Node root, std::ofstream& out):*

Описание: экспортирует дерево в формате DOT для визуализации через Graphviz.

Входные данные: *Node* root* — указатель на корень дерева, *std::ofstream& out* — поток для записи в файл.

Выходные данные: файл в формате DOT для визуализации дерева.

RBtree::bool searchTreeNode(int series, int passport):

Описание: проверяет наличие узла с указанными параметрами.

Входные данные: *int series* — серия паспорта, *int passport* — номер паспорта.

Выходные данные: true, если узел найден, false, если узел отсутствует.

RBtree::void deleteTree(Node node):*

Описание: удаляет все узлы дерева, освобождая память.

Входные данные: *Node* node* — указатель на корень дерева.

Выходные данные: очищенное дерево.

RBtree::RBtree():

Описание: инициализирует пустое красно-черное дерево.

Входные данные: отсутствуют.

Выходные данные: объект дерева с *root = nullptr*.

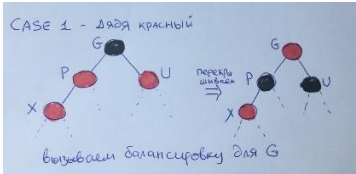
RBtree::void insert(Passport p):

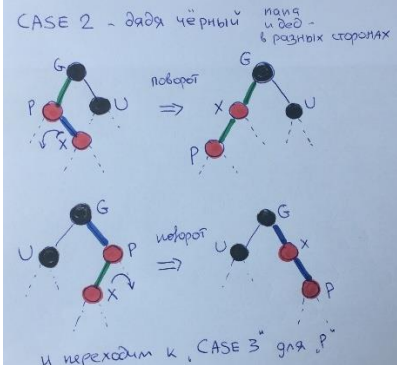
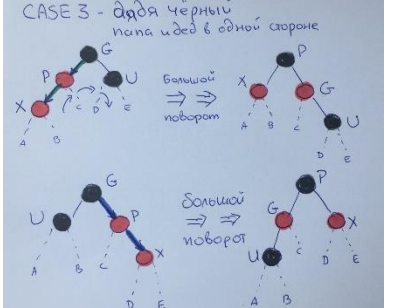
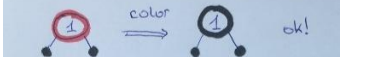
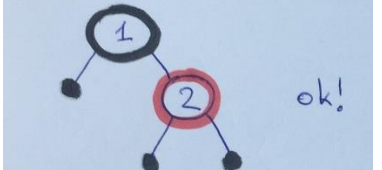
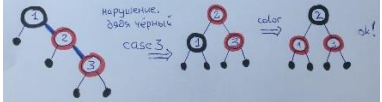
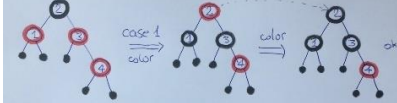
Описание: вставляет паспорт *p* в красно-черное дерево.

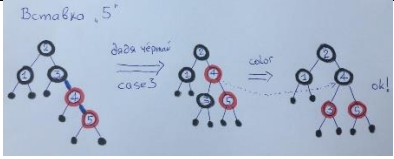
Входные данные: *Passport p* — структура, содержащая *series* — серия паспорта, *passport* — номер паспорта, *line* — номер строки из файла (если используется *insertFromFile*).

Выходные данные: модифицированное дерево с добавленным паспортом, если паспорт уже существует, обновляется список дубликатов.

Тесты:

Описание тестовой ситуации	Входные данные (Input)	Выходные данные (Output)
Добавление в пустое дерево	Паспорт: 1234 567890, дерево: - Введите серию паспорта: 1234 Введите номер паспорта: 567890 Паспорт добавлен.	Дерево с корнем: 1234 567890 (черный) Содержимое дерева: 1234 567890(B)
Добавление меньшего элемента	Паспорт: 1233 567890, дерево: 1234 567890 Введите серию паспорта: 1233 Введите номер паспорта: 567890 Паспорт добавлен.	Дерево с узлом 1233 567890 (красный) слева Содержимое дерева: 1234 567890(B) 1233 567890(R)
Добавление большего элемента	Паспорт: 1235 567890, дерево: 1234 567890 Введите серию паспорта: 1235 Введите номер паспорта: 567890 Паспорт добавлен.	Дерево с узлом 1235 567890 (красный) справа Содержимое дерева: 1235 567890(R) 1234 567890(B)
Добавление дубликата	Паспорт: 1234 567890, дерево: 1234 567890 Passport p; Passport p1; p.series = 1234; p.passport = 567890; p.line = 1; p1.series = 1234; p1.passport = 567890; p1.line = 2; tree.insert(p); tree.insert(p1);	Узел 1234 567890: список дубликатов обновлен Duplicatelist head = 0x791dd0 data = 2 next = 0x0 prev = 0x0 tail = 0x791dd0
Балансировка дерева после вставки	Последовательность: 1234 567890, 1233 567890, 1235 567890 дерево: 1234 567890, 1233 567890, 1235 567890	Дерево сбалансировано Содержимое дерева: 1235 567890(R) 1234 567890(B) 1233 567890(R)
Случай первый — красный дядя 	Вставка: 1232 567890 Дерево: 1234 567890, 1233 567890, 1235 567890 Содержимое дерева: 1235 567890(R) 1234 567890(B) 1233 567890(R) 1232 567890(R)	Содержимое дерева: 1235 567890(B) 1234 567890(B) 1233 567890(B) 1232 567890(R)

<p>Случай второй — чёрный дядя — папа и дед в разных сторонах.</p> 	<p>Вставка: 1232 567890 Дерево: 1234 567890, 1231 567890, 1235 67890</p> <p>Содержимое дерева:</p> <p>1235 567890(B)</p> <p>1234 567890(R)</p> <p>1232 567890(B)</p> <p>1231 567890(R)</p> <p>1231 567890(R)</p> <p>Введите серию паспорта: 1232 Введите номер паспорта: 567890 Паспорт добавлен.</p>	<p>Содержимое дерева:</p> <p>1235 567890(B)</p> <p>1234 567890(R)</p> <p>1232 567890(B)</p> <p>1231 567890(R)</p>
<p>Случай третий — чёрный дядя — папа и дед в одной стороне</p> 	<p>Вставка: 1232 567890 Дерево: 1234 567890, 1231 567890, 1235 67890</p> <p>Содержимое дерева:</p> <p>1235 567890(B)</p> <p>1234 567890(R)</p> <p>1232 567890(B)</p> <p>1231 567890(R)</p> <p>1231 567890(R)</p> <p>Введите серию паспорта: 1232 Введите номер паспорта: 567890 Паспорт добавлен.</p>	<p>Содержимое дерева:</p> <p>1235 567890(B)</p> <p>1234 567890(R)</p> <p>1232 567890(B)</p> <p>1231 567890(R)</p>
<p>Вставка "1" в пустое дерево</p> 	<p>Вставка: 1231 567890 Дерево: -</p>	<p>Содержимое дерева:</p> <p>1231 567890(B)</p>
<p>Вставка "2"</p> 	<p>Вставка: 1232 567890 Дерево: 1231 567890</p>	<p>Содержимое дерева:</p> <p>1232 567890(R)</p> <p>1231 567890(B)</p>
<p>Вставка "3"</p> 	<p>Вставка: 1233 567890 Дерево: 1231 567890, 1232 567890</p>	<p>Содержимое дерева:</p> <p>1233 567890(R)</p> <p>1232 567890(B)</p> <p>1231 567890(R)</p>
<p>Вставка "4"</p> 	<p>Вставка: 1234 567890 Дерево: 1231 567890, 1232 567890, 1233 567890</p>	<p>Содержимое дерева:</p> <p>1234 567890(R)</p> <p>1233 567890(B)</p> <p>1232 567890(B)</p> <p>1231 567890(B)</p>

	Вставка: 1235 567890 Дерево: 1231 567890, 1232 567890, 1233 567890, 1234 567890	Содержимое дерева: 1235 567890(R) 1234 567890(B) 1233 567890(R) 1232 567890(B) 1231 567890(B)
---	--	--

RBtree::void remove(Passport p):

Описание: удаляет паспорт p из красно-черного дерева.

Входные данные: Passport p — структура, содержащая series и passport.

Выходные данные: модифицированное дерево с удаленным паспортом, если он существует.

Тесты:

Описание тестовой ситуации	Входные данные (Input)	Выходные данные (output)
Удаление из пустого дерева	Паспорт: 1234 567890, дерево: - Введите серию паспорта для удаления: 1234 Введите номер паспорта для удаления: 567890 Паспорт удалён (если существовал).	Дерево: - Экран: -
Удаление узла без детей	Паспорт: 1233 567890, дерево: 1234 567890, 1233 567890 Содержимое дерева: 1234 567890(B) 1233 567890(R) Введите серию паспорта для удаления: 1233 Введите номер паспорта для удаления: 567890 Паспорт удалён (если существовал).	Дерево: 1234 567890 Содержимое дерева: 1234 567890(B)
Удаление узла с одним ребенком	Паспорт: 1234 567890, дерево: 1234 567890, 1235 567890 Содержимое дерева: 1235 567890(R) 1234 567890(B) Введите серию паспорта для удаления: 1234 Введите номер паспорта для удаления: 567890 Паспорт удалён (если существовал).	Дерево: 1235 567890 (черный) Содержимое дерева: 1235 567890(R)
Удаление узла с двумя детьми	Паспорт: 1234 567890, дерево: 1234 567890, 1233 567890, 1235 567890 Введите серию паспорта для удаления: 1234 Введите номер паспорта для удаления: 567890 Паспорт удалён (если существовал). Содержимое дерева: 1235 567890(R) 1234 567890(B) 1233 567890(R)	Дерево сбалансировано: Содержимое дерева: 1235 567890(R) 1233 567890(B)
Удаление корня	Паспорт: 1233 567890, дерево: 1233 567890, 1235 567890	Новый корень: 1235 567890 (черный)

	Содержимое дерева: 1235 567890(R) 1233 567890(B) Введите серию паспорта для удаления: 1233 Введите номер паспорта для удаления: 567890 Паспорт удалён (если существовал).	Содержимое дерева: 1235 567890(R)
--	--	--------------------------------------

RBtree::void printTree():

Описание: печатает дерево в древовидной форме.

Входные данные: отсутствуют.

Выходные данные: дерево, напечатанное в консоль в древовидной форме.

Тесты:

Описание тестовой ситуации	Входные данные	Выходные данные
Печать пустого дерева	Дерево: -	Содержимое дерева:
Печать дерева с одним узлом	Дерево: 1234 567890	1234 567890 (черный) Содержимое дерева 1234 567890(B)
Печать дерева с несколькими узлами	Дерево: 1234 567890, 1233 567890, 1235 567890	Отформатированная структура дерева Содержимое дерева: 1235 567890(R) 1234 567890(B) 1233 567890(R)

RBtree::bool searchTreeNode(const Passport& p):

Описание: проверяет, есть ли паспорт p в дереве.

Входные данные: Passport& p — структура, содержащая series и passport.

Выходные данные: true, если паспорт найден, false, если паспорт отсутствует.

Тесты:

Описание тестовой ситуации	Входные данные	Выходные данные
Поиск в пустом дереве	Паспорт: 1234 567890, дерево: - Введите серию паспорта для поиска: 1234 Введите номер паспорта для поиска: 56789 Содержимое дерева:	false, сообщение: "Паспорт не найден" Tree is empty! Паспорт не найден.
Поиск существующего элемента	Паспорт: 1234 567890, дерево: 1234 567890 Содержимое дерева 1234 567890(B) Введите серию паспорта для поиска: 1234 Введите номер паспорта для поиска: 567890	true, сообщение: "Паспорт найден в дереве." 1234 567890 is in the Tree !!! Паспорт найден в дереве.

Поиск отсутствующего элемента	Паспорт: 1233 567890, дерево: 1234 567890 Содержимое дерева 1234 567890(B) Введите серию паспорта для поиска: 1233 Введите номер паспорта для поиска: 567890	false, сообщение: "Паспорт не найден." 1233 567890 is not in the Tree !!! Паспорт не найден.
-------------------------------	--	---

RBtree::void printPreOrder():

Описание: печатает узлы дерева в порядке прямого обхода.

Входные данные: отсутствуют.

Выходные данные: узлы дерева, напечатанные в консоль в порядке прямого обхода.

Тесты:

Описание тестовой ситуации	Входные данные	Выходные данные
Печать пустого дерева	Пустое дерево	Прямой обход дерева:
Прямой обход дерева	Дерево: 1234 567890, 1233 567890, 1235 567890 Содержимое дерева: 1235 567890(R) 1234 567890(B) 1233 567890(R)	Узлы в порядке: 1234 567890, 1233 567890, 1235 567890 1234 567890(B): [] 1233 567890(R): [] 1235 567890(R): []
Прямой обход дерева (с дубликатами)	Ввод: 1234 567890, 1233 567890, 1235 567890, 1234 567890 Содержимое дерева: 1235 567890(R) 1234 567890(B) 1233 567890(R)	Прямой обход дерева: 1234 567890(B): [0] 1233 567890(R): [] 1235 567890(R): []

RBtree::void insertFromFile(const std::string& filename):

Описание: метод выгружает паспорта из текстового файла и добавляет их в дерево.

Входные данные: std::string filename — имя файла с данными, формат строки: <series> <passport>.

Выходные данные: модифицированное дерево с добавленными паспортами, сообщения об ошибках, если данные некорректны или файл недоступен.

Тесты:

Описание тестовой ситуации	Входные данные	Выходные данные
Вставка из корректного файла	Файл: data.txt (содержит 8537 576580 3840 957894 7395 353768 7395 353768 9060 751071 4925 949330	Дерево с узлами из файла:

	1723 606604 3668 422693 1667 536116 423 97252 sosi americanec 3840 957894 3840 957894)	Содержимое дерева: 9060 751071(R) 8537 576580(B) 7395 353768(R) 4925 949330(B) 3840 957894(B) 3668 422693(B) 1723 606604(R) 1667 536116(B) 423 97252(R)
Файл отсутствует	passportd.txt	Сообщение "Unable to open file: src/passport.txt " Unable to open file: src/passportd.txt

RBtree::void exportToGraphviz(const std::string& filename):

Описание: экспортирует дерево в формате DOT для визуализации через Graphviz.

Входные данные: std::string filename — имя выходного файла (например, tree.dot).

Выходные данные: файл в формате DOT, представляющий структуру дерева.

Тесты:

Описание тестовой ситуации	Входные данные	Выходные данные
Экспорт дерева с узлами	tree.dot	Файл содержит корректное описание дерева в формате DOT Дерево экспортировано в файл tree.dot. Введите: dot -Tpng tree.dot -o tree.png

RBtree::~~RBtree():

Описание: удаляет все узлы дерева, освобождая память.

Входные данные: отсутствуют.

Выходные данные: полностью очищенное дерево.

3. Текст программы

- **Файл "RBtree.h"**

```
#ifndef RBTREE_H
#define RBTREE_H

#include<iostream>
#include"DoublyLinkedList.h"
#include<fstream>
#include<sstream>

// Паспорт
struct Passport {
int series;    // серия
int passport; // номер
```

```

int line;
Passport(int s, int p) : series(s), passport(p), line(0) {}
};

enum Color { RED, BLACK };

// Узел дерева
struct Node {
    Passport passport;
    Color color;
    Node* parent;
    Node* right;
    Node* left;
    DoublyLinkedList DuplicateList;

    Node(Passport k) : passport(k), color(BLACK), parent(nullptr), right(nullptr),
    left(nullptr), DuplicateList() {};
};

class RBtree {
private:
    // Корень дерева
    Node* root;
    // Поворот налево
    void leftRotate(Node*& root, Node* x) {
        // Указатель на правого потомка x
        Node* y = x->right;

        // Перемещаем левое поддерево y на место правого поддерева x
        x->right = y->left;
        if (y->left != nullptr) {
            y->left->parent = x;
        }

        // Устанавливаем родителя y в качестве родителя x
        y->parent = x->parent;
        if (x->parent == nullptr) {
            // x был корнем, теперь y становится корнем
            root = y;
        } else if (x == x->parent->left) {
            x->parent->left = y;
        } else {
            x->parent->right = y;
        }

        // Делаем x левым потомком y
        y->left = x;
        x->parent = y;
    }
    // Поворот направо
    void rightRotate(Node*& root, Node* y) {

```

```

// Указатель на левого потомка y
Node* x = y->left;

// Перемещаем правое поддерево x на место левого поддерева y
y->left = x->right;
if (x->right != nullptr) {
    x->right->parent = y;
}

// Устанавливаем родителя x в качестве родителя y
x->parent = y->parent;
if (y->parent == nullptr) {
    // y был корнем, теперь x становится корнем
    root = x;
} else if (y == y->parent->right) {
    y->parent->right = x;
} else {
    y->parent->left = x;
}

// Делаем y правым потомком x
x->right = y;
y->parent = x;
}

// Исправить нарушения после вставки узла
void fixInsert(Node*& root, Node* z) {
    // Исправляет нарушения красно-черного дерева после вставки узла z
    while (z != root && z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            Node* y = z->parent->parent->right; // Дядя узла z
            if (y != nullptr && y->color == RED) {
                // Случай 1: Дядя красный
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    // Случай 2: z — правый потомок
                    z = z->parent;
                    leftRotate(root, z);
                }
                // Случай 3: z — левый потомок
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rightRotate(root, z->parent->parent);
            }
        } else {
            Node* y = z->parent->parent->left; // Дядя узла z
            if (y != nullptr && y->color == RED) {
                // Случай 1: Дядя красный

```

```

z->parent->color = BLACK;
y->color = BLACK;
z->parent->parent->color = RED;
z = z->parent->parent;
} else {
if (z == z->parent->left) {
// Случай 2: z — левый потомок
z = z->parent;
rightRotate(root, z);
}
// Случай 3: z — правый потомок
z->parent->color = BLACK;
z->parent->parent->color = RED;
leftRotate(root, z->parent->parent);
}
}
}
root->color = BLACK; // Корень всегда черный
}

// Функция transplant, используемая при удалении
void transplant(Node*& root, Node* u, Node* v) {
// Заменяет поддерево с корнем в u поддеревом с корнем в v
if (u->parent == nullptr) {
root = v; // Если u — корень, то v становится новым корнем
} else if (u == u->parent->left) {
u->parent->left = v; // u был левым потомком
} else {
u->parent->right = v; // u был правым потомком
}
if (v != nullptr) {
v->parent = u->parent; // Устанавливаем родителя v
}
}

// Найти максимальный слева узел в поддереве
Node* maximum(Node* node) {
while (node->right != nullptr) node = node->right;
return node;
}

// Удаление node
void deleteNode(Node*& root, Node* z) {
// Удаляет узел z из красно-черного дерева
Node* y = z;
Node* x;
Node* k = z;
Color originalColor = y->color;
if (z->left == nullptr) {
x = z->right;
transplant(root, z, z->right);

```

```

} else if (z->right == nullptr) {
x = z->left;
transplant(root, z, z->left);
} else {
y = maximum(z->left); // Предшественник z
originalColor = y->color;
x = y;
if (y == z) {
if (x != nullptr) {
x->parent = y;
}
} else {
transplant(root, y, y->left);
y->left = z->left;
y->left->parent = y;
}
transplant(root, z, y);
y->right = z->right;
y->right->parent = y;
y->color = z->color;
}

if (originalColor == BLACK && x != nullptr && x != root) {
fixDelete(root, x);
}
}

// Функция исправления нарушений после удаления узла
void fixDelete(Node*& root, Node* x) {
// Исправляет нарушения красно-черного дерева после удаления узла x
while (x != root && (x == nullptr || x->color == BLACK)) {
if (x == x->parent->left) {
Node* w = x->parent->right;
if (w->color == RED) {
w->color = BLACK;
x->parent->color = RED;
leftRotate(root, x->parent);
w = x->parent->right;
}
if ((w->left == nullptr || w->left->color == BLACK) &&
(w->right == nullptr || w->right->color == BLACK)) {
w->color = RED;
x = x->parent;
} else {
if (w->right == nullptr || w->right->color == BLACK) {
if (w->left != nullptr) {
w->left->color = BLACK;
}
}
w->color = RED;
rightRotate(root, w);
w = x->parent->right;
}
}
}

```



```

}
w->color = x->parent->color;
x->parent->color = BLACK;
if (w->right != nullptr) {
w->right->color = BLACK;
}
leftRotate(root, x->parent);
x = root;
}
} else {
Node* w = x->parent->left;
if (w->color == RED) {
w->color = BLACK;
x->parent->color = RED;
rightRotate(root, x->parent);
w = x->parent->left;
}
if ((w->right == nullptr || w->right->color == BLACK) &&
(w->left == nullptr || w->left->color == BLACK)) {
w->color = RED;
x = x->parent;
} else {
if (w->left == nullptr || w->left->color == BLACK) {
if (w->right != nullptr) {
w->right->color = BLACK;
}
w->color = RED;
leftRotate(root, w);
w = x->parent->left;
}
w->color = x->parent->color;
x->parent->color = BLACK;
if (w->left != nullptr) {
w->left->color = BLACK;
}
rightRotate(root, x->parent);
x = root;
}
}
}
if (x != nullptr) {
x->color = BLACK;
}
}
// Распечатать древовидную структуру (прямой обход)
void printHelper(Node* root, int space) {
// Печатает дерево в древовидной форме (прямой обход)
if (root == nullptr) {
return;
}
const int COUNT = 10;

```

```

space += COUNT;
printHelper(root->right, space);
std::cout << std::endl;
for (int i = COUNT; i < space; i++) {
std::cout << " ";
}
std::cout << root->passport.series << " " << root->passport.passport << (root-
>color == RED ? "(R)" : "(B)") << "\n";
printHelper(root->left, space);
}
// Распечатать прямой обход
void print_pre_order(Node* root) {

if (root == nullptr) return;
Node* current = root;
std::cout << root->passport.series << " " << root->passport.passport << (root-
>color == RED ? "(R)" : "(B)") << ": ";
root->DuplicateList.print();
std::cout << std::endl;
print_pre_order(root->left);
print_pre_order(root->right);
}
// Дополнительно
void exportToGraphviz(Node* root, std::ofstream& out) {
if (root == nullptr) {
return;
}

// Записываем текущий узел
out << "\t\"" << root->passport.series << "_" << root->passport.passport << "\"
[label=\""
<< root->passport.series << " " << root->passport.passport << "\n";
out << (root->color == RED ? "RED" : "BLACK") << "\"]";
out << ";\n";

// Если есть левый потомок, соединяем с ним
if (root->left != nullptr) {
out << "\t\"" << root->passport.series << "_" << root->passport.passport << "\" -
> \""
<< root->left->passport.series << "_" << root->left->passport.passport <<
"\";\n";
}

// Если есть правый потомок, соединяем с ним
if (root->right != nullptr) {
out << "\t\"" << root->passport.series << "_" << root->passport.passport << "\" -
> \""
<< root->right->passport.series << "_" << root->right->passport.passport <<
"\";\n";
}
}

```

```

// Рекурсивно обрабатываем потомков
exportToGraphviz(root->left, out);
exportToGraphviz(root->right, out);
}
// Поиск заданного элемента (if node in tree = true/false)
bool searchTreeNode(int series, int passport) {
Node* current = root; // Начинаем поиск с корня
if (current == nullptr) {
std::cout << "Tree is empty!" << std::endl;
return false; // Explicit return when the tree is empty
}

while (current != nullptr) {
if (current->passport.series == series && current->passport.passport == passport)
{
std::cout << series << " " << passport << " is in the Tree !!!" << std::endl;
return true; // Элемент найден
} else if (series < current->passport.series ||
(series == current->passport.series && passport < current->passport.passport)) {
current = current->left; // Идем в левое поддерево
} else {
current = current->right; // Идем в правое поддерево
}
}

std::cout << series << " " << passport << " is not in the Tree !!!" << std::endl;
// Элемент не найден
return false; // Explicit return when the element is not found
}

// Рекурсивная функция для удаления узлов
void deleteTree(Node* node) {
if (node == nullptr) {
return;
}
deleteTree(node->left); // Удаляем левое поддерево
deleteTree(node->right); // Удаляем правое поддерево
delete node;           // Удаляем текущий узел
}

public:
RBtree() : root(nullptr) {}
// Вставить паспорт
void insert(Passport p) {
Passport newPassport = {p.series, p.passport};
newPassport.line = p.line;

// Создать новый узел для вставки
Node* newNode = new Node(newPassport);
newNode->color = RED; // Новый узел всегда красный
newNode->left = newNode->right = nullptr;

```

```

// Если дерево пустое, делаем новый узел корнем
if (root == nullptr) {
newNode->color = BLACK; // Корень всегда черный
root = newNode;
return;
}

// Найти подходящее место для вставки
Node* current = root;
Node* parent = nullptr;
while (current != nullptr) {
parent = current;
if (newPassport.series == current->passport.series && newPassport.passport ==
current->passport.passport) {
// Если нашли дубликат, добавляем в список дубликатов и выходим
current->DuplicateList.push_back(newPassport.line);
delete newNode; // Удаляем новый узел, так как он не нужен
return;
} else if (newPassport.series < current->passport.series ||
(newPassport.series == current->passport.series && newPassport.passport <
current->passport.passport)) {
current = current->left;
} else {
current = current->right;
}
}

// Устанавливаем родителя нового узла
newNode->parent = parent;
if (newPassport.series < parent->passport.series ||
(newPassport.series == parent->passport.series && newPassport.passport < parent-
>passport.passport)) {
parent->left = newNode;
} else {
parent->right = newNode;
}

// Исправляем нарушения красно-черного дерева
fixInsert(root, newNode);
}

// Удалить паспорт
void remove(Passport p) {
int series = p.series;
int passport = p.passport;
Node* temp = root;
while (temp != nullptr) {
if (temp->passport.series == series && temp->passport.passport == passport) {
deleteNode(root, temp);
return;
}
}
}

```

```

} else if (series < temp->passport.series || (series == temp->passport.series &&
passport < temp->passport.passport)) {
temp = temp->left;
} else {
temp = temp->right;
}
}
}
// Напечатать в древовидном виде
void printTree() {
printHelper(root, 0);
}
// Есть ли?
bool searchTreeNode(const Passport& p) {
return searchTreeNode(p.series, p.passport);
}
// Вывод в прямом обходе
void printPreOrder() {
print_pre_order(root);
std::cout << std::endl;
}
// Вставка из файла
void insertFromFile(const std::string& filename) {
std::ifstream file(filename);
if (!file.is_open()) {
std::cerr << "Unable to open file: " << filename << std::endl;
return;
}

std::string line;
int lineNumber = 0;
while (std::getline(file, line)) {
lineNumber++;
std::istringstream iss(line);
int series, passport;
if (!(iss >> series >> passport)) {
std::cerr << "Invalid data on line " << lineNumber << ": " << line << std::endl;
continue;
}
Passport p = {series, passport};
p.line = lineNumber;
insert(p);
}

file.close();
}
// Дополнительно : dot -Tpng tree.dot -o tree.png
void exportToGraphviz(const std::string& filename) {
std::ofstream out(filename);
if (!out.is_open()) {
std::cerr << "Unable to open file: " << filename << std::endl;

```

```

return;
}

out << "digraph RBTREE {\n";
out << "\tnode [fontname=\"Arial\", shape=circle];\n";

if (root == nullptr) {
out << "\tEmptyTree;\n";
} else {
exportToGraphviz(root, out);
}

out << "}";
out.close();
}

// Деструктор
~RBtree() {
deleteTree(root); // Рекурсивно удаляем все узлы
}
};

#endif // RBTREE_H

```

- Файл *"DoublyLinkedList.h"*

```

#ifndef DOUBLY_LINKED_LIST_H
#define DOUBLY_LINKED_LIST_H

#include <iostream>

// lNode structure for the doubly linked list
struct lNode {
    int data;
    lNode* next;
    lNode* prev;

    lNode(int value) : data(value), next(nullptr), prev(nullptr) {}
};

class DoublyLinkedList {
private:
    lNode* head;
    lNode* tail;

public:
    // Constructor
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}

    // Destructor
    ~DoublyLinkedList() {

```

```

        lNode* current = head;
        while (current) {
            lNode* nextlNode = current->next;
            delete current;
            current = nextlNode;
        }
    }

    // Push an element to the back of the list
    void push_back(int value) {
        lNode* newlNode = new lNode(value);
        if (!head) {
            head = tail = newlNode;
        } else {
            tail->next = newlNode;
            newlNode->prev = tail;
            tail = newlNode;
        }
    }

    // Print the list elements
    void print() const {
        lNode* current = head;
        std::cout << "[";
        while (current) {
            std::cout << current->data;
            if (current->next) {
                std::cout << ", "; // Добавить запятую между элементами
            }
            current = current->next;
        }
        std::cout << "]; // Закрыть скобку и добавить перенос строки
    }

    // Delete an element with the given value
    void delete_value(int value) {
        lNode* current = head;
        while (current) {
            if (current->data == value) {
                if (current->prev) {
                    current->prev->next = current->next;
                } else {
                    head = current->next;
                }

                if (current->next) {
                    current->next->prev = current->prev;
                } else {
                    tail = current->prev;
                }
            }
        }
    }

```

```

        delete current;
        return;
    }
    current = current->next;
}
};

#endif // DOUBLY_LINKED_LIST_H

```

- **Файл "main.cpp"**

```

#include "src/RBtree.h"
#include <iostream>
#include <string>
#include <iomanip>

using namespace std;

void menu() {
    cout << "\n===== \n";
    cout << "                МЕНЮ                \n";
    cout << "===== \n";
    cout << left << setw(30) << " 1. Вставить паспорт" << endl;
    cout << left << setw(30) << " 2. Удалить паспорт" << endl;
    cout << left << setw(30) << " 3. Поиск паспорта" << endl;
    cout << left << setw(30) << " 4. Вывести дерево" << endl;
    cout << left << setw(30) << " 5. Прямой обход" << endl;
    cout << left << setw(30) << " 6. Нарисовать дерево" << endl;
    cout << left << setw(30) << " 7. Выход" << endl;
    cout << "===== \n";
    cout << "Выберите пункт: ";
}

int main() {
    RBtree tree;
    string filename = "src/passports.txt";
    tree.insertFromFile(filename); // Вставляем данные из файла
    // Passport p;
    // Passport p1;
    // p.series = 1234;
    // p.passport = 567890;
    // p.line = 1;
    // p1.series = 1234;
    // p1.passport = 567890;
    // p1.line = 2;
    // tree.insert(p);
    // tree.insert(p1);
    while (true) {
        menu();
        int choice;
    }
}

```



```

cin >> choice;
switch (choice) {
    case 1: {
        int series, number;
        cout << "Введите серию паспорта: ";
        cin >> series;
        cout << "Введите номер паспорта: ";
        cin >> number;
        Passport p(series, number);
        tree.insert(p);
        cout << "Паспорт добавлен.\n";
        break;
    }
    case 2: {
        int series, number;
        cout << "Введите серию паспорта для удаления: ";
        cin >> series;
        cout << "Введите номер паспорта для удаления: ";
        cin >> number;
        Passport p(series, number);
        tree.remove(p);
        cout << "Паспорт удалён (если существовал).\n";
        break;
    }
    case 3: {
        int series, number;
        cout << "Введите серию паспорта для поиска: ";
        cin >> series;
        cout << "Введите номер паспорта для поиска: ";
        cin >> number;
        Passport p(series, number);
        if (tree.searchTreeNode(p)) {
            cout << "Паспорт найден в дереве.\n";
        } else {
            cout << "Паспорт не найден.\n";
        }
        break;
    }
    case 4: {
        cout << "Содержимое дерева:\n";
        tree.printTree();
        break;
    }
    case 5: {
        cout << "Прямой обход дерева:\n";
        tree.printPreOrder();
        break;
    }
    case 6: {
        string graphvizFile = "tree.dot";
        tree.exportToGraphviz(graphvizFile);
    }
}

```

```

        cout << "Дерево экспортировано в файл " << graphvizFile << ".\n";
        cout << "Введите: dot -Tpng tree.dot -o tree.png" << endl;
        break;
    }
    case 7: {
        cout << "Выход из программы.\n";
        return 0;
    }
    default: {
        cout << "Неверный выбор. Попробуйте ещё раз.\n";
        break;
    }
}

return 0;
}

```