Изображение выглядит как текст

Автоматически созданное описание

**КУРСОВАЯ РАБОТА**

**Алгоритмы работы со словарями**

**по дисциплине «Алгоритмы и структуры данных»**

Выполнил

cтудент гр. 5130904/40008 Лиходиевский А. М.

Руководитель Червинский А. П.

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2025

Оглавление

[1. Введение. Общая постановка задачи: 3](#__RefHeading___Toc4183_112312499)

[Основная часть работы 5](#__RefHeading___Toc4426_112312499)

[1. Описание алгоритма решения и используемых структур данных 5](#__RefHeading___Toc4428_112312499)

[2. Анализ алгоритма 7](#__RefHeading___Toc4430_112312499)

[3. Описание спецификации программы (детальные требования) 8](#__RefHeading___Toc34476_112312499)

[4. Описание программы (структура программы, формат входных и выходных данных) 9](#__RefHeading___Toc34565_112312499)

[Заключение 10](#__RefHeading___Toc34478_112312499)

# **1.** **Введение. Общая постановка задачи:**

**Тема: Алгоритмы работы со словарями**

1. Для разрабатываемого словаря реализовать основные операции:

* + INSERT (ключ, значение) — добавить запись с указанным ключом и значением
  + SEARCH (ключ) — найти запись с указанными ключом
  + DELETE (ключ) — удалить запись с указанным ключом

2. Предусмотреть обработку и инициализацию исключительных ситуаций, связанных, например, с проверкой значения полей перед инициализацией и присваиванием.

3. Программа должна быть написана в соответствии со стилем программирования: C++ Programming Style Guidelines (<http://geosoft.no/development/cppstyle.html>).

4. Тесты должны учитывать как допустимые, так и не допустимые последовательности входных данных.

**Вариант 1.3.3.**

**Частотный словарь.** ***Красно-черное дерево*.**

Разработать и реализовать алгоритм формирования частотного словаря:

* определить понятие слово,
* прочитать текст и сформировать набор слов вместе с информацией о частоте их встречаемости
* определить три чаще всего встречающихся слова

Для реализации задания использовать красно-черное дерево.

Узел дерева должен содержать:

• Ключ – слово

• Цвет узла

• Информационная часть – количество слов

# **Основная часть работы**

## Описание алгоритма решения и используемых структур данных

*Красно-чёрное дерево* - это бинарное дерево поиска, где у каждого узла есть атрибут цвета, при этом:

1. Узел должен быть либо красным, либо чёрным
2. Корень дерева чёрный
3. Все фиктивные листья (NIL – узлы) — чёрные
4. Оба потомка каждого красного узла — чёрные
5. Любой простой путь от узла-предка до листового узла потомка содержит одинаковое число чёрных узлов

*Частотный словарь —* набор слов, вместе с частотой и их встречаемостью.

Определим понятие «слово» для данного словаря. *Слово — слово естественного языка, представленное в формате данных string.* Ключом данного дерева является определённое нами понятие «слова».

Узел дерева — структура Node, содержащая следующие поля:

* key\_ (значение ключа)
* frequency\_ (количество повторений данного слова а тексте)
* right\_ (указатель на правое поддерево)
* left\_ (указатель на левое поддерево)
* color\_ (цвет узла)
* parent\_ (указатель на «родителя» узла)

Дерево строится на основе класса RedBlackTree, содержащего указатель на его вершину (root), начальное значение которого равно нулевому указателю.

Дерево поддерживает следующие методы:

1. getTopKFrequent(std::size\_t k) – поиск k самых часто встречающихся слов

Данный метод основывается на итераторе, написанном для красно-чёрного дерева, классах Vector, являющимся аналогом одноимённого класса из STL, и Pair (аналог std::pair)

Данный метод возвращает вектор из k элементов пар вида «ключ, частота». При вводе числа k большего, чем размер самой структуры, метод вернёт вектор состоящий из количества элементов равного размеру структуры

1. search(const T &k) – поиск записи с указанным значением

Данный метод основывается на методе searchNode (const T &k), который возвращает указатель на узел, если запись с указанным значением присутствует в структуре или нулевой указатель, если такая запись отсутствует

Результат работы метода search – булевское значение, отвечающее на вопрос: «Есть ли элемент с таким ключом в структуре?»

1. insert(const T &key) – добавление записи с указанным ключом

Данный метод основывается на методе searchNode (const T &k), благодаря которому частота слова (frequency\_) увеличивается, если происходит повторение

При перебалансировке дерева fixInsert(std::shared\_ptr<Node> pivot) для данного узла с помощью методов RotateRight(std::shared\_ptr<Node> pivot) и RotateLeft(std::shared\_ptr<Node> pivot), которые совершают повороты влево и вправо относительно узла для перебалансировки дерева

В итоге, метод insert добавляет в дерево новый узел, если переданное в качестве параметра значение не было найдено или увеличивает частоту встречаемости для данного значения

1. remove(const T& key) – удаление узла с заданным значением

Работает на основе методов searchNode(const T&k), который находит нужный узел для удаления (если таковой имеется), replaceSubtree(std::shared\_ptr<Node> oldNode, std::shared\_ptr<Node> newNode), который заменяет одно поддерево другим в дереве, и fixDelete(std::shared\_ptr<Node>) , который восстанавливает свойства красно-чёрного дерева после удаления узла

Метод remove удаляет узел с заданным значением, если оно было найдено. Метод возвращает булевское значение, которое зависит от того: произошло ли удаление элемента или нет.

1. getUncle(const std::shared\_ptr<Node>) - метод возвращает брата родителя, если он есть
2. getNumberOfNodes() - метод возвращает количество элементов, находящихся в структуре
3. getFrequency(T& arg) – возвращает частоту словарями

## 2. Анализ алгоритма

*Время выполнения:*

* Поиск слова в RB-дерево: O(log n)
* Вставка слова в RB-дерево: O(log n)
* Увеличение частоты слова: O (1)

Поскольку алгоритм выполняет эти операции для каждого слова в тексте, а длина текста в среднем составляет n слов, общая сложность алгоритма равна O (n log n)

*Память:*

Алгоритм хранит RB-дерево, которое содержит слова и их частоты в тексте. Размер RB-дерева зависит от количества уникальных слов в тексте. В среднем: O(n), где n – количество уникальных слов.

## 3. Описание спецификации программы (детальные требования)

* Поиск 3-х наиболее часто встречающихся слов, возможен только тогда, когда количество элементов в словаре больше 3-х
  + Если количество элементов меньше 3, возвращается вектор состоящий из того количества элементов, сколько находится в RB-дереве, вида «значение, частота»
  + Иначе поиск 3-х наиболее встречающихся ключа и возвращается вектор из 3-х элементов вида «значение, частота»

## 4. Описание программы (структура программы, формат входных и выходных данных)

Подключение библиотек и заголовочных файлов

#include <cassert>  
#include <fstream>  
#include <sstream>  
#include "RedBlackTree.hpp"  
#include "ReadFunc.hpp"

Тестирование программы происходит с помощью функции *mainTest()*, которая проверяет работу RB-дерева и поиска 3-х часто встречающихся слов в тексте.

В начале работы программы происходит тестирование с помощью функций. Затем создаётся файл с текстом.

*Входные данные:*

* Последовательности символов из сгенерированного файла (string)

*Выходные данные:*

* Информация о трёх наиболее часто встречающихся словах (значение — строка типа std::string, частота встречаемости — число типо std::size\_t)

**Тест план**

1. Файл должен быть открыт корректно, в противном случае — вывод в консоль сообщения «Unable to open file: {filePath}»
2. Функция по получению 3-х самых часто встречающихся слов должна возвращать корректные данные

# **Заключение**

В ходе работы были изучены сбалансированное дерево поиска (красно-черное дерево), основы его построение, а также понятие частотного словаря. Был разработан частный словарь, реализованный на основе RB-дерева, написаны методы вставки, поиска, удаления заданного значения в словаре, поиск 3-х наиболее часто встречающимися слов. Написаны структуры данных: вектор, очередь на основе связанного списка, стэк и пары

**Main.cpp**

#include "Test.hpp"  
  
int main() {  
 mainTest();  
}

**Test.hpp**

#pragma once  
  
#include <cassert>  
#include <fstream>  
#include <sstream>  
#include "RedBlackTree.hpp"  
#include "ReadFunc.hpp"  
  
inline void testInsertAndSearch() {  
 RedBlackTree<int> tree;  
 assert(tree.search(10) == false);  
 assert(tree.insert(10) == true);  
 assert(tree.search(10) == true);  
 assert(tree.insert(10) == false);  
 assert(tree.search(20) == false);  
}  
  
inline void testRemove() {  
 RedBlackTree<int> tree;  
 tree.insert(10);  
 tree.insert(20);  
 assert(tree.remove(10) == true);  
 assert(tree.search(10) == false);  
 assert(tree.search(20) == true);  
 assert(tree.remove(30) == false);  
}  
  
inline void testRedBlackPropertiesAfterInsert() {  
 RedBlackTree<int> tree;  
 tree.insert(10);  
 tree.insert(20);  
 tree.insert(5);  
 tree.insert(15);  
 assert(tree.begin() != tree.end());  
}  
  
  
inline void testNodeCount() {  
 RedBlackTree<int> tree;  
 assert(tree.getNumberOfNodes() == 0);  
 tree.insert(10);  
 assert(tree.getNumberOfNodes() == 1);  
 tree.insert(20);  
 assert(tree.getNumberOfNodes() == 2);  
 tree.remove(10);  
 assert(tree.getNumberOfNodes() == 1);  
}  
  
inline void testIterator() {  
 RedBlackTree<int> tree;  
 tree.insert(5);  
 tree.insert(3);  
 tree.insert(7);  
  
 auto it = tree.begin();  
 assert(\*it == 3);  
 ++it;  
 assert(\*it == 5);  
 ++it;  
 assert(\*it == 7);  
 ++it;  
 assert(it == tree.end());  
}  
  
  
inline void testRemoveWithTwoChildren() {  
 RedBlackTree<int> tree;  
 tree.insert(10);  
 tree.insert(5);  
 tree.insert(15);  
 tree.insert(12);  
 tree.insert(20);  
  
 assert(tree.remove(15) == true);  
 assert(tree.search(15) == false);  
 assert(tree.search(12) == true);  
 assert(tree.search(20) == true);  
}  
  
inline void testCustomType() {  
 struct Point {  
 int x, y;  
 bool operator<(const Point& other) const { return x < other.x; }  
 bool operator>(const Point& other) const { return x > other.x; }  
 bool operator==(const Point& other) const { return x == other.x && y == other.y; }  
 bool operator!=(const Point& other) const { return !(\*this == other); }  
 };  
  
 RedBlackTree<Point> tree;  
 tree.insert({1, 2});  
 tree.insert({3, 4});  
 assert(tree.search({1, 2}) == true);  
 assert(tree.search({3, 4}) == true);  
 assert(tree.search({5, 6}) == false);  
}  
  
inline void testMoveConstructor() {  
 RedBlackTree<int> tree1;  
 tree1.insert(10);  
 tree1.insert(5);  
  
 RedBlackTree<int> tree2(std::move(tree1));  
 assert(tree2.search(10) == true);  
 assert(tree2.search(5) == true);  
 assert(tree1.search(10) == false);  
}  
  
inline void testMoveAssignment() {  
 RedBlackTree<int> tree1, tree2;  
 tree1.insert(10);  
 tree1.insert(5);  
  
 tree2 = std::move(tree1);  
 assert(tree2.search(10) == true);  
 assert(tree2.search(5) == true);  
 assert(tree1.search(10) == false);  
}  
  
inline void testBasicTop3() {  
 const std::string test\_file = "test\_basic.txt";  
 std::ofstream out(test\_file);  
 out << "apple banana apple cherry banana apple";  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top3 = tree.getTopKFrequent(3);  
  
 assert(top3.get\_size() == 3);  
 assert(top3[0].first == "apple" && top3[0].second == 3);  
 assert(top3[1].first == "banana" && top3[1].second == 2);  
 assert(top3[2].first == "cherry" && top3[2].second == 1);  
  
}  
  
inline void testRepeatedWords() {  
 const std::string test\_file = "test\_repeated.txt";  
 std::ofstream out(test\_file);  
 out << "hello hello hello world world world world test test";  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top3 = tree.getTopKFrequent(3);  
  
 assert(top3.get\_size() == 3);  
 assert(top3[0].first == "world" && top3[0].second == 4);  
 assert(top3[1].first == "hello" && top3[1].second == 3);  
 assert(top3[2].first == "test" && top3[2].second == 2);  
  
}  
  
inline void testLessThan3Words() {  
 const std::string test\_file = "test\_less3.txt";  
 std::ofstream out(test\_file);  
 out << "single";  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top = tree.getTopKFrequent(3);  
  
 assert(top.get\_size() == 1);  
 assert(top[0].first == "single" && top[0].second == 1);  
  
}  
  
inline void testCaseInsensitive() {  
 const std::string test\_file = "test\_case.txt";  
 std::ofstream out(test\_file);  
 out << "Apple apple aPpLE Banana banana";  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top2 = tree.getTopKFrequent(2);  
  
 assert(top2.get\_size() == 2);  
 assert(top2[0].first == "apple" && top2[0].second == 3);  
 assert(top2[1].first == "banana" && top2[1].second == 2);  
  
}  
  
  
inline void testEmptyFile() {  
 const std::string test\_file = "test\_empty.txt";  
 std::ofstream out(test\_file);  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top = tree.getTopKFrequent(3);  
  
 assert(top.get\_size() == 0);  
  
}  
  
inline void testDifferentDelimiters() {  
 const std::string test\_file = "test\_delimiters.txt";  
 std::ofstream out(test\_file);  
 out << "word1,word2.word3!word4?word1\nword2\tword5";  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top3 = tree.getTopKFrequent(3);  
 assert(top3[0].first == "word" && top3[0].second == 7);  
  
}  
  
inline void testSimpleText() {  
 const std::string test\_file = "test\_simple.txt";  
 std::ofstream out(test\_file);  
 out << "the quick brown fox jumps over the lazy dog and the fox is quick";  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top3 = tree.getTopKFrequent(3);  
  
 assert(top3.get\_size() == 3);  
 assert(top3[0].first == "the" && top3[0].second == 3);  
 assert(top3[1].first == "fox" && top3[1].second == 2);  
 assert(top3[2].first == "quick" && top3[2].second == 2);  
  
}  
  
inline void testTextWithPunctuation() {  
 const std::string test\_file = "test\_punctuation.txt";  
 std::ofstream out(test\_file);  
 out << "Hello, world! World is beautiful. Hello everyone!";  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top3 = tree.getTopKFrequent(3);  
  
 assert(top3.get\_size() == 3);  
 assert(top3[0].first == "hello" && top3[0].second == 2);  
 assert(top3[1].first == "world" && top3[1].second == 2);  
 assert(top3[2].first == "is" || top3[2].first == "beautiful" || top3[2].first == "everyone");  
  
}  
  
inline void testTextWithSpecialChars() {  
 const std::string test\_file = "test\_special\_chars.txt";  
 std::ofstream out(test\_file);  
 out << "It's a well-known fact; mother-in-law and state-of-the-art are compound words.";  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top3 = tree.getTopKFrequent(3);  
  
 assert(top3.get\_size() == 3);  
 for (const auto& pair : top3) {  
 assert(pair.first == "a" || pair.first == "its" || pair.first == "and" ||  
 pair.first == "are" || pair.first == "fact" || pair.first == "words" ||  
 pair.first == "wellknown" || pair.first == "motherinlaw" || pair.first == "stateoftheart");  
 }  
  
}  
  
inline void testMultilineText() {  
 const std::string test\_file = "test\_multiline.txt";  
 std::ofstream out(test\_file);  
 out << "First line\nSecond line\nThird line\nFirst line again";  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top3 = tree.getTopKFrequent(3);  
  
 assert(top3.get\_size() == 3);  
 assert(top3[0].first == "line" && top3[0].second == 4);  
 assert((top3[1].first == "first" && top3[1].second == 2) ||  
 (top3[1].first == "second" || top3[1].first == "third" || top3[1].first == "again"));  
  
}  
  
inline void testTextWithStopWords() {  
 const std::string test\_file = "test\_stopwords.txt";  
 std::ofstream out(test\_file);  
 out << "This is a test of the system. This test should work correctly.";  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top3 = tree.getTopKFrequent(3);  
  
 assert(top3.get\_size() == 3);  
 assert(top3[0].first == "test" && top3[0].second == 2);  
 assert(top3[1].first == "this" && top3[1].second == 2);  
 assert(top3[2].first == "a" || top3[2].first == "of" || top3[2].first == "the" ||  
 top3[2].first == "system" || top3[2].first == "should" ||  
 top3[2].first == "work" || top3[2].first == "correctly" || top3[2].first == "is");  
  
}  
  
inline void testLiteratureFragment() {  
 const std::string test\_file = "test\_literature.txt";  
 std::ofstream out(test\_file);  
 out << "It was the best of times, it was the worst of times, "  
 << "it was the age of wisdom, it was the age of foolishness...";  
 out.close();  
  
 auto tree = extractWordsFromFile(test\_file);  
 auto top3 = tree.getTopKFrequent(3);  
  
 assert(top3.get\_size() == 3);  
 assert(top3[0].first == "it" && top3[0].second == 4);  
 assert((top3[1].first == "was" || top3[1].first == "of") && top3[1].second == 4);  
 assert(top3[2].first == "the" && top3[2].second == 4);  
  
}  
  
inline void mainTest() {  
 testInsertAndSearch();  
 testRemove();  
 testRedBlackPropertiesAfterInsert();  
 testNodeCount();  
 testIterator();  
 testRemoveWithTwoChildren();  
 testCustomType();  
 testMoveConstructor();  
 testMoveAssignment();  
 testBasicTop3();  
 testRepeatedWords();  
 testLessThan3Words();  
 testCaseInsensitive();  
 testEmptyFile();  
 testDifferentDelimiters();  
 testSimpleText();  
 testTextWithPunctuation();  
 testTextWithSpecialChars();  
 testMultilineText();  
 testTextWithStopWords();  
 testLiteratureFragment();  
  
 std::cout << "All tests passed successfully!" << std::endl;  
  
}

**RedBlackTree.hpp**

*/\*\*  
\* @author JDH-LR-994  
\* @file RedBlackTree.hpp  
\* @brief Реализация красно-чёрного дерева  
 \* Красно-чёрное дерево - это бинарное дерево поиска, где у каждого  
 \* узла есть атрибут цвета, при этом:  
 \* 1. Узел должен быть либо красным, либо чёрным  
 \* 2. Корень - как правило чёрный  
 \* 3. Все фиктивные листья (NIL - узлы) - чёрные  
 \* 4. Оба потомка каждого красного узла - чёрные  
 \* 5. Любой простой путь от узла-предка до листового узла потомка  
 \* содержит одинаковое число чёрных узлов  
 \*  
 \* ВАЖНО! Как и любое другое бинарное дерево поиска, КЧД может работать только  
 \* с типами, которые можно сравнивать (указано вторым шаблонным параметром)  
 \*  
 \* @note Все операции (вставка, поиск) работают за O(log n) в худшем случае.  
 \*/*#ifndef REDBLACKTREE\_HPP  
#define REDBLACKTREE\_HPP  
#include <cmath>  
#include <iostream>  
#include <iterator>  
#include <memory>  
#include <type\_traits>  
#include "StackArray.hpp"  
#include "QueueList.hpp"  
#include "Vector.hpp"  
#include "Pair.hpp"  
  
template<typename...>  
using void\_t = void;  
*/\*\*  
 \* @brief SFINAE-проверка наличия операторов сравнения у типа T.  
 \* @details Проверяет, определён ли для типа T операторы <, >, ==, !=.  
 \* Если операторы отсутствуют, компиляция завершится с ошибкой.  
 \*/*template<typename T, typename = void>  
struct is\_comparable : std::false\_type {  
};  
  
template<typename T>  
struct is\_comparable<T, void\_t<  
 decltype(std::declval<T>() < std::declval<T>()),  
 decltype(std::declval<T>() > std::declval<T>()),  
 decltype(std::declval<T>() == std::declval<T>()),  
 decltype(std::declval<T>() != std::declval<T>())  
 > > : std::true\_type {  
};  
  
  
*/\*\*  
 \* @brief Класс красно-чёрного дерева.  
 \* @tparam T Тип данных, хранящихся в узлах дерева.  
 \* @tparam T0 Вспомогательный параметр для SFINAE (не используется напрямую).  
 \* @details Если операторы сравнения отсутствуют, тип T не сможет быть использован в RedBlackTree.  
\*/*template<typename T, typename T0 = std::enable\_if\_t<is\_comparable<T>::value> >  
class RedBlackTree {  
 */\*\*  
 \* @brief Перечисление цветов узлов красно-чёрного дерева.  
 \* @details В красно-чёрном дереве каждый узел должен быть либо красным, либо чёрным.  
 \* Это фундаментальное свойство, обеспечивающее балансировку дерева.  
 \* - RED: Красный узел  
 \* - BLACK: Чёрный узел  
 \*/* enum class Color { RED, BLACK };  
  
 struct Node;  
 */\*\*  
 \* @brief Умные указатели для работы с узлами дерева.  
 \* @details Используются shared\_ptr для владеющих указателей и weak\_ptr для  
 \* ссылок на родителя, чтобы избежать циклических зависимостей.  
 \*/* using NodePtr = std::shared\_ptr<Node>; *///< Умный указатель на узел.* using WeakNodePtr = std::weak\_ptr<Node>; *///< Слабый указатель на узел (для избегания циклов)  
  
 /\*\*  
 \* @brief Структура узла дерева.  
 \*/* struct Node {  
 T key\_; *///< Ключ узла.* std::size\_t frequency\_ = 1;  
 Color color\_; *///< Цвет узла.* NodePtr left\_; *///< Левый потомок.* NodePtr right\_; *///< Правый потомок.* WeakNodePtr parent\_; *///< Ссылка на родителя (слабый указатель).  
 /\*\*  
 \* @brief Конструктор узла.  
 \* @param k Ключ узла.  
 \* @param c Цвет узла.  
 \* @param p Ссылка на родителя (по умолчанию пустая).  
 \*/* Node(T k, Color c, WeakNodePtr p = WeakNodePtr{}) : key\_(k),  
 color\_(c),  
 left\_(*NIL*),  
 right\_(*NIL*),  
 parent\_(p) {  
 }  
 };  
  
public:  
 */\*\*  
 \* @brief Конструктор по умолчанию. Инициализирует дерево с пустым корнем (NIL).  
 \*/* RedBlackTree(): root(*NIL*) {  
 };  
  
 RedBlackTree(const RedBlackTree &) = delete; *///< Запрещённый конструктор копирования.* RedBlackTree &operator=(const RedBlackTree &) = delete; *///< Запрещённое копирующее присваивание.* RedBlackTree(RedBlackTree &&other) noexcept *///< Разрешенный конструктор перемещения.* ;  
  
 RedBlackTree &operator=(RedBlackTree &&other) noexcept *///< Разрешенное перемещающее присваивание.* ;  
  
 std::size\_t getFrequency(T& arg) const;  
  
 ~RedBlackTree() = default; *///< Деструктор по умолчанию.  
  
 /\*\*  
 \* @brief Итератор для обхода красно-чёрного дерева в порядке in-order (симметричный порядок).  
 \*  
 \* @details Итератор реализует forward iterator, позволяющий последовательно обходить узлы дерева  
 \* от наименьшего к наибольшему ключу. Обход выполняется нерекурсивно с использованием стека,  
 \* что гарантирует корректную работу даже для очень больших деревьев.  
 \*  
 \* Особенности реализации:  
 \* - Использует StackArray<NodePtr> для хранения узлов при обходе  
 \* - Автоматически вычисляет необходимый размер стека на основе количества узлов  
 \* - Поддерживает стандартные операции итератора: \*, ->, ++ (префиксный и постфиксный)  
 \* - Корректно обрабатывает end-итератор (пустое дерево или завершение обхода)  
 \*  
 \* Гарантии:  
 \* - Время доступа к следующему элементу: O(1) амортизированное  
 \* - Потребление памяти: O(h), где h - высота дерева  
 \* - Инвалидация итератора при модификации дерева  
 \*  
 \* Пример использования:  
 \* @code  
 \* RedBlackTree<int> tree;  
 \* // ... заполнение дерева ...  
 \*  
 \* // Вариант 1: Range-based for loop  
 \* for (const auto& value : tree) {  
 \* std::cout << value << " ";  
 \* }  
 \*  
 \* // Вариант 2: Явное использование итераторов  
 \* for (auto it = tree.begin(); it != tree.end(); ++it) {  
 \* std::cout << \*it << " ";  
 \* }  
 \* @endcode  
 \*  
 \* @tparam T Тип данных, хранящихся в узлах дерева  
 \* @tparam T0 SFINAE-параметр (не используется напрямую)  
 \*/* class Iterator {  
 public:  
 using iterator\_category = std::forward\_iterator\_tag;  
 using value\_type = T;  
 using difference\_type = std::ptrdiff\_t;  
 using pointer = T \*;  
 using reference = T &;  
  
 explicit Iterator(NodePtr root = *NIL*, size\_t max\_stack\_size = 100);  
  
 reference operator\*() const;  
  
 pointer operator->() const;  
  
 */\*\*  
 \* @brief Префиксный оператор инкремента для итератора.  
 \* @return Ссылка на текущий итератор после инкремента.  
 \* @throws std::out\_of\_range если итератор уже end().  
 \* @details Алгоритм работы:  
 \* 1. Извлекаем текущий узел из стека  
 \* 2. Если у узла есть правый потомок, заполняем стек левыми потомками правого поддерева  
 \* 3. Возвращаем обновлённый итератор  
 \*  
 \* @note Сложность: O(1) амортизированная, так как каждый узел помещается в стек  
 \* и извлекается из него ровно один раз за весь обход.  
 \*/* Iterator &operator++();  
  
 std::size\_t getFrequency() const;  
  
 */\*\*  
 \* @brief Постфиксный оператор инкремента для итератора.  
 \* @return Копия итератора до инкремента.  
 \* @details Отличается от префиксной версии тем, что возвращает  
 \* предыдущее состояние итератора до увеличения.  
 \*  
 \* @note Реализован через префиксный оператор для избежания дублирования кода.  
 \*/* Iterator operator++(int);  
  
 bool operator==(const Iterator &other) const;  
  
 bool operator!=(const Iterator &other) const;  
  
 private:  
 StackArray<NodePtr> stack{100}; // Фиксированный размер стека (можно увеличить при необходимости)  
 */\*\*  
 \* @brief Заполняет стек левыми потомками начиная с заданного узла.  
 \* @param node Узел, с которого начинается заполнение стека.  
 \* @details Рекурсивно добавляет в стек все левые потомки узла,  
 \* что позволяет в дальнейшем обходить их в порядке возрастания.  
 \*  
 \* @note Используется при инициализации итератора и переходе к правому поддереву.  
 \*/* void fillStack(NodePtr node);  
 };  
  
 Iterator begin() const {  
 size\_t max\_depth = countNodes > 0 ? 2 \* static\_cast<size\_t>(log2(countNodes + 1)) + 5 : 10;  
 return Iterator(root, max\_depth);  
 }  
  
 Iterator end() const {  
 return Iterator();  
 }  
  
  
 Vector<Pair<T, int>> getTopKFrequent(std::size\_t k) const {  
 Vector<Pair<T, int>> elements;  
  
 for (auto it = begin(); it != end(); it++) {  
 elements.push\_back(Pair<T, int>(\*it, it.getFrequency()));  
 }  
 quickSort(elements);  
  
 if (k > elements.get\_size()) k = elements.get\_size();  
  
 Vector<Pair<T, int>> result;  
 for (size\_t i = 0; i < k; ++i) {  
 result.push\_back(elements[i]);  
 }  
 return result;  
 }  
  
 */\*\*  
 \* @brief Поиск ключа в дереве.  
 \* @param k Ключ для поиска.  
 \* @return True, если ключ найден, иначе false.  
 \* @example  
 \* RedBlackTree<int> tree;  
 \* tree.insert(10);  
 \* bool found = tree.search(10); // Вернёт true  
 \* bool notFound = tree.search(20); // Вернёт false  
 \*/* bool search(const T &k) const;  
  
 */\*\*  
 \* @brief Вставка ключа в дерево.  
 \* @param key Ключ для вставки.  
 \* @return True, если вставка успешна, false, если ключ уже существует.  
 \* @example  
 \* RedBlackTree<int> tree;  
 \* bool inserted = tree.insert(10); // Вернёт true  
 \* bool duplicate = tree.insert(10); // Вернёт false (ключ уже есть)  
 \*/* bool insert(const T &key);  
  
 */\*\*  
 \* @brief Удаляет узел с заданным ключом из дерева и восстанавливает свойства красно-чёрного дерева.  
 \*  
 \* @param key Ключ узла, который требуется удалить.  
 \* @return True, если узел был найден и удалён, false, если ключ отсутствует в дереве.  
 \*  
 \* @details Алгоритм удаления состоит из нескольких этапов:  
 \* 1. Поиск удаляемого узла с помощью searchNode.  
 \* 2. Определение узла для замены (successor) и его потомка (child):  
 \* - Если у узла нет одного из потомков, он заменяется своим единственным потомком.  
 \* - Если у узла есть оба потомка, он заменяется узлом с минимальным ключом из правого поддерева (successor).  
 \* 3. Фиксация свойств дерева после удаления, если был удалён чёрный узел (вызов fixDelete).  
 \*  
 \* @note После удаления:  
 \* - Если удалённый узел был красным, свойства дерева не нарушаются.  
 \* - Если удалённый узел был чёрным, требуется балансировка (fixDelete).  
 \*  
 \* @warning Удаление несуществующего ключа безопасно и вернёт false.  
 \*  
 \* @example  
 \* RedBlackTree<int> tree;  
 \* tree.insert(10);  
 \* tree.insert(20);  
 \* bool success = tree.remove(10); // Вернёт true  
 \* bool fail = tree.remove(30); // Вернёт false (ключ отсутствует)  
 \*  
 \* @see fixDelete() для деталей балансировки после удаления.  
 \* @see searchNode() для реализации поиска узла.  
 \*/* bool remove(const T &key);  
  
 */\*\*  
 \* @brief Метод, который позволяет получить количество узлов в дереве  
 \* @return количество узлов в дереве  
 \*/* [[nodiscard]] int getNumberOfNodes() const {  
 return countNodes;  
 }  
  
  
private:  
 */\*\*  
 \* @brief Фиктивный чёрный узел, представляющий листья дерева.  
 \* @details Инициализируется через createNIL() и является общим для всех экземпляров дерева.  
 \*/* static NodePtr *NIL*; *///< Фиктивный чёрный узел, представляющий листья дерева.* NodePtr root = *NIL*; *///< Корень дерева.* std::size\_t countNodes = 0; *///< Количество узлов в дереве  
  
 /\*\*  
 \* @brief Получение "дядюшки" узла (брата родителя).  
 \* @details Дядя узла — это брат его родителя. Например:  
 \* - Если узел является левым потомком своего родителя, то дядя — правый потомок деда.  
 \* - Если узел является правым потомком, то дядя — левый потомок деда.  
 \* @param node Узел, для которого ищется дядя.  
 \* @return Указатель на дядю или NIL, если:  
 \* - у узла нет родителя,  
 \* - у родителя нет деда,  
 \* - дядя является NIL.  
 \* @example  
 \* // Пример корректного дерева:  
 \* // G (чёрный)  
 \* // / \  
 \* // P (чёрный) U (красный) <-- U — дядя для X.  
 \* // /  
 \* // X (красный)  
 \* auto uncle = getUncle(X); // Вернёт U, так как U — брат родителя P.  
 \*  
 \* // Пример, когда дядя — NIL:  
 \* // G (чёрный)  
 \* // / \  
 \* // P (чёрный) NIL <-- Дядя для X — NIL.  
 \* // /  
 \* // X (красный)  
 \* auto uncle = getUncle(X); // Вернёт NIL.  
 \*/* NodePtr getUncle(const NodePtr &node) const;  
  
 */\*\*  
 \* @brief Выполняет правый поворот вокруг указанного узла.  
 \*  
 \* @param pivot Узел, являющийся центром поворота (ось поворота)  
 \*  
 \* @details Графическое представление операции:  
 \* @verbatim  
 \* До поворота: После поворота:  
 \*  
 \* pivot child  
 \* / \ / \  
 \* child C --> A pivot  
 \* / \ / \  
 \* A B B C  
 \* @endverbatim  
 \*  
 \* @note Особенности операции:  
 \* 1. Узел `child` становится новым корнем поддерева  
 \* 2. Поддерево `B` становится правым потомком `pivot`  
 \* 3. Сохраняется порядок элементов: A < child < B < pivot < C  
 \*  
 \* @warning Ограничения:  
 \* - Запрещено вызывать для NIL-узлов  
 \* - Узел должен иметь левого потомка (child != NIL)  
 \*  
 \* @see rotateLeft() для симметричной операции  
 \*/* void rotateRight(NodePtr pivot);  
  
 */\*\*  
 \* @brief Выполняет левый поворот вокруг указанного узла.  
 \*  
 \* @param pivot Узел, вокруг которого выполняется поворот (ось поворота)  
 \*  
 \* @details Графическое представление операции:  
 \* @verbatim  
 \* До поворота: После поворота:  
 \*  
 \* pivot child  
 \* / \ / \  
 \* A child --> pivot C  
 \* / \ / \  
 \* B C A B  
 \* @endverbatim  
 \*  
 \* @note Особенности операции:  
 \* 1. Узел `child` становится новым корнем поддерева  
 \* 2. Поддерево `B` становится левым потомком `pivot`  
 \* 3. Сохраняется порядок элементов: A < pivot < B < child < C  
 \*  
 \* @warning Ограничения:  
 \* - Запрещено вызывать для NIL-узлов  
 \* - Узел должен иметь правого потомка (child != NIL)  
 \*  
 \* @see rotateRight() для симметричной операции  
 \*/* void rotateLeft(NodePtr pivot);  
  
 */\*\*  
 \* @brief Балансировка дерева после вставки.  
 \* @param pivot Узел, с которого начинается балансировка (новый узел).  
 \* @details Восстанавливает свойства красно-чёрного дерева после вставки:  
 \* 1. Каждый узел красный или чёрный  
 \* 2. Корень чёрный  
 \* 3. Все листья (NIL) чёрные  
 \* 4. Оба потомка красного узла чёрные  
 \* 5. Пути от узла до листьев содержат одинаковое число чёрных узлов  
 \*  
 \* Алгоритм рассматривает три основных случая:  
 \* - Случай 1: Дядя узла красный -> перекрашивание  
 \* - Случай 2: Дядя чёрный, узел - правый потомок -> левый поворот  
 \* - Случай 3: Дядя чёрный, узел - левый потомок -> правый поворот  
 \*  
 \* @note Сложность: O(log n) в худшем случае, так как может потребоваться  
 \* подняться от нового узла до корня.  
 \* @example  
 \* RedBlackTree<int> tree;  
 \* tree.insert(10); // Автоматически вызывает fixInsert при необходимости.  
 \*/* void fixInsert(NodePtr pivot);  
  
 */\*\*  
 \* @brief Поиск узла по ключу.  
 \* @param k Ключ для поиска.  
 \* @return Указатель на найденный узел или NIL, если узел не найден.  
 \* @note Используется внутри методов класса (например, для удаления).  
 \*/* NodePtr searchNode(const T &k) const;  
  
 */\*\*  
 \* @brief Заменяет одно поддерево другим в дереве.  
 \*  
 \* @param oldNode Поддерево, которое нужно заменить  
 \* @param newNode Поддерево, на которое нужно заменить  
 \*  
 \* @details Функция выполняет следующие действия:  
 \* 1. Обновляет ссылку у родителя oldNode на newNode  
 \* 2. Обновляет родителя у newNode  
 \*  
 \* @note Не изменяет связи newNode с его потомками  
 \* @warning Не проверяет, является ли oldNode корнем дерева  
 \*/* void replaceSubtree(NodePtr oldNode, NodePtr newNode);  
  
 */\*\*  
 \* @brief Находит узел с минимальным ключом в поддереве.  
 \*  
 \* @param subtreeRoot Корень поддерева для поиска  
 \* @return Узел с минимальным ключом в поддереве  
 \*  
 \* @note Если subtreeRoot == NIL, возвращает NIL  
 \* @complexity O(h), где h - высота поддерева  
 \*/* NodePtr minimum(NodePtr subtreeRoot);  
  
 */\*\*  
 \* @brief Восстанавливает свойства красно-чёрного дерева после удаления узла.  
 \*  
 \* @param pivot Узел, с которого начинается восстановление (обычно удалённый или заменённый узел).  
 \*  
 \* @details Этот метод вызывается после удаления узла и обрабатывает возможные нарушения свойств КЧД:  
 \* - Если удалённый узел был чёрным, это могло нарушить баланс чёрных высот.  
 \* - Метод рекурсивно поднимается от `pivot` к корню, исправляя нарушения.  
 \*  
 \* Алгоритм рассматривает 4 основных случая:  
 \*  
 \* 1. Брат красный:  
 \* - Перекрашивает брата в чёрный, родителя — в красный.  
 \* - Делает поворот вокруг родителя в сторону `pivot`.  
 \* - Преобразует ситуацию в случаи 2, 3 или 4.  
 \*  
 \* 2. Брат чёрный, оба его потомка чёрные:  
 \* - Перекрашивает брата в красный.  
 \* - Переносит проблему на родителя (теперь он становится новым `pivot`).  
 \*  
 \* 3. Брат чёрный, ближний потомок красный, дальний — чёрный:  
 \* - Перекрашивает ближний потомок в чёрный, брата — в красный.  
 \* - Делает поворот вокруг брата в сторону, противоположную `pivot`.  
 \* - Преобразует ситуацию в случай 4.  
 \*  
 \* 4. Брат чёрный, дальний потомок красный:  
 \* - Перекрашивает брата в цвет родителя.  
 \* - Перекрашивает родителя и дальний потомок в чёрный.  
 \* - Делает поворот вокруг родителя в сторону `pivot`.  
 \* - Завершает балансировку (проблема решена).  
 \*  
 \* @note После завершения цикла корень всегда перекрашивается в чёрный (свойство КЧД №2).  
 \*  
 \* @example  
 \* // После удаления узла:  
 \* fixDelete(replacedNode); // Восстанавливает свойства дерева.  
 \*/* void fixDelete(NodePtr pivot);  
  
 */\*\*  
 \* @brief Создаёт фиктивный лист (NIL) с чёрным цветом и ссылками на себя.  
 \* @details Этот метод вызывается только один раз при инициализации статического члена NIL.  
 \* @return Указатель shared\_ptr на фиктивный лист.  
 \*/* static NodePtr *createNIL*();  
  
 static void *quickSort*(Vector<Pair<std::string, int>>& vec);  
  
 static void *quickSortImpl*(Vector<Pair<std::string, int>>& vec, int low, int high);  
  
 static int *partition*(Vector<Pair<std::string, int>>& vec, int low, int high);  
};  
  
template<typename T, typename T0>  
RedBlackTree<T, T0>::RedBlackTree(RedBlackTree &&other) noexcept: root(std::move(other.root)), countNodes(other.countNodes) {  
 other.root = *NIL*;  
 other.countNodes = 0;  
}  
  
template<typename T, typename T0>  
RedBlackTree<T, T0> & RedBlackTree<T, T0>::operator=(RedBlackTree &&other) noexcept {  
 if (this != &other) {  
 root = std::move(other.root);  
 countNodes = other.countNodes;  
 other.root = *NIL*;  
 other.countNodes = 0;  
 }  
 return \*this;  
}  
  
template<typename T, typename T0>  
std::size\_t RedBlackTree<T, T0>::getFrequency(T &arg) const {  
 auto node = searchNode(arg);  
 if (node != *NIL*) {  
 return node->frequency\_;  
 }  
 return 0;  
}  
  
template<typename T, typename T0>  
RedBlackTree<T, T0>::Iterator::Iterator(NodePtr root, size\_t max\_stack\_size): stack(max\_stack\_size) {  
 if (root != *NIL*) {  
 fillStack(root);  
 }  
}  
  
template<typename T, typename T0>  
typename RedBlackTree<T, T0>::Iterator::reference RedBlackTree<T, T0>::Iterator::operator\*() const {  
 if (stack.isEmpty()) {  
 throw std::out\_of\_range("Dereferencing end iterator");  
 }  
 return stack.top()->key\_;  
}  
  
template<typename T, typename T0>  
typename RedBlackTree<T, T0>::Iterator::pointer RedBlackTree<T, T0>::Iterator::operator->() const {  
 if (stack.isEmpty()) {  
 throw std::out\_of\_range("Dereferencing end iterator");  
 }  
 return &stack.top()->key\_;  
}  
  
template<typename T, typename T0>  
typename RedBlackTree<T, T0>::Iterator &RedBlackTree<T, T0>::Iterator::operator++() {  
 if (stack.isEmpty()) {  
 throw std::out\_of\_range("Incrementing end iterator");  
 }  
 NodePtr node = stack.pop();  
 if (node->right\_ != *NIL*) {  
 fillStack(node->right\_);  
 }  
 return \*this;  
}  
  
template<typename T, typename T0>  
std::size\_t RedBlackTree<T, T0>::Iterator::getFrequency() const {  
 return stack.top().get()->frequency\_;  
}  
  
template<typename T, typename T0>  
typename RedBlackTree<T, T0>::Iterator RedBlackTree<T, T0>::Iterator::operator++(int) {  
 Iterator tmp = \*this;  
 ++(\*this);  
 return tmp;  
}  
  
template<typename T, typename T0>  
bool RedBlackTree<T, T0>::Iterator::operator==(const Iterator &other) const {  
 // Оба итератора равны, если их стеки пусты (end-итераторы)  
 // Или если их верхние элементы совпадают (для промежуточных состояний)  
 return (stack.isEmpty() && other.stack.isEmpty()) ||  
 (!stack.isEmpty() && !other.stack.isEmpty() && stack.top() == other.stack.top());  
}  
  
template<typename T, typename T0>  
bool RedBlackTree<T, T0>::Iterator::operator!=(const Iterator &other) const {  
 return !(\*this == other);  
}  
  
template<typename T, typename T0>  
void RedBlackTree<T, T0>::Iterator::fillStack(NodePtr node) {  
 while (node != *NIL*) {  
 stack.push(node);  
 node = node->left\_;  
 }  
}  
  
template<typename T, typename T0>  
bool RedBlackTree<T, T0>::search(const T &k) const {  
 return searchNode(k) != *NIL*;  
}  
  
template<typename T, typename T0>  
bool RedBlackTree<T, T0>::insert(const T &key) {  
  
 NodePtr found = searchNode(key);  
 if (found != *NIL*) {  
 ++(found->frequency\_);  
 return false;  
 }  
  
 if (root == *NIL*) {  
 root = std::make\_shared<Node>(key, Color::BLACK);  
 root->left\_ = root->right\_ = *NIL*;  
 countNodes++;  
 return true;  
 }  
  
 NodePtr currentNode = root;  
 NodePtr parent = *NIL*;  
  
 // Поиск места для вставки  
 while (currentNode != *NIL*) {  
 parent = currentNode;  
 if (key < currentNode->key\_) {  
 currentNode = currentNode->left\_;  
 } else if (key > currentNode->key\_) {  
 currentNode = currentNode->right\_;  
 } else {  
 return false; // Ключ уже существует  
 }  
 }  
  
 // Создание нового узла  
 auto newNode = std::make\_shared<Node>(key, Color::RED, parent);  
 if (key < parent->key\_) {  
 parent->left\_ = newNode;  
 } else {  
 parent->right\_ = newNode;  
 }  
  
 // Балансировка  
 fixInsert(newNode);  
 countNodes++;  
 return true;  
}  
  
template<typename T, typename T0>  
bool RedBlackTree<T, T0>::remove(const T &key) {  
 NodePtr node = searchNode(key);  
 if (node == *NIL*) return false; // Узел не найден  
  
 NodePtr successor = *NIL*;  
 NodePtr child = *NIL*;  
 Color originalColor = node->color\_;  
  
 // Определяем узел для удаления и его потомка  
 if (node->left\_ == *NIL*) {  
 child = node->right\_;  
 replaceSubtree(node, child);  
 } else if (node->right\_ == *NIL*) {  
 child = node->left\_;  
 replaceSubtree(node, child);  
 } else {  
 // У узла есть оба потомка  
 successor = minimum(node->right\_);  
 originalColor = successor->color\_;  
 child = successor->right\_;  
  
 if (successor->parent\_.lock() != node) {  
 replaceSubtree(successor, child);  
 successor->right\_ = node->right\_;  
 successor->right\_->parent\_ = successor;  
 }  
 replaceSubtree(node, successor);  
 successor->left\_ = node->left\_;  
 successor->left\_->parent\_ = successor;  
 successor->color\_ = node->color\_;  
 }  
  
 // Балансировка, если удалён чёрный узел  
 if (originalColor == Color::BLACK) {  
 fixDelete(child);  
 }  
 countNodes--;  
 return true;  
}  
  
  
template<typename T, typename T0>  
typename RedBlackTree<T, T0>::NodePtr RedBlackTree<T, T0>::getUncle(const NodePtr &node) const {  
 auto parent = node->parent\_.lock();  
 if (!parent) return *NIL*; // Если родитель не существует, дядя тоже не существует.  
 auto grandparent = parent->parent\_.lock();  
 if (!grandparent) return *NIL*;  
 return (parent == grandparent->left\_) ? grandparent->right\_ : grandparent->left\_;  
}  
  
template<typename T, typename T0>  
void RedBlackTree<T, T0>::rotateLeft(NodePtr pivot) {  
 if (pivot == *NIL* || pivot->right\_ == *NIL*) return; // Проверка на NIL  
 auto child = pivot->right\_;  
 pivot->right\_ = child->left\_;  
  
 if (pivot->right\_)  
 pivot->right\_->parent\_ = pivot;  
  
 child->parent\_ = pivot->parent\_;  
  
 if (auto parent = pivot->parent\_.lock()) {  
 if (parent->left\_ == pivot)  
 parent->left\_ = child;  
 else  
 parent->right\_ = child;  
 } else {  
 root = child;  
 }  
  
 child->left\_ = pivot;  
 pivot->parent\_ = child;  
}  
  
template<typename T, typename T0>  
void RedBlackTree<T, T0>::rotateRight(NodePtr pivot) {  
 if (pivot == *NIL* || pivot->left\_ == *NIL*) return; // Проверка на NIL  
 auto child = pivot->left\_;  
 pivot->left\_ = child->right\_;  
  
 if (pivot->left\_)  
 pivot->left\_->parent\_ = pivot;  
  
 child->parent\_ = pivot->parent\_;  
  
 if (auto parent = pivot->parent\_.lock()) {  
 if (parent->left\_ == pivot)  
 parent->left\_ = child;  
 else  
 parent->right\_ = child;  
 } else {  
 root = child;  
 }  
  
 child->right\_ = pivot;  
 pivot->parent\_ = child;  
}  
  
  
template<typename T, typename T0>  
void RedBlackTree<T, T0>::fixInsert(NodePtr pivot) {  
 while (pivot != root && pivot->parent\_.lock()->color\_ == Color::RED) {  
 auto parent = pivot->parent\_.lock();  
 auto grandparent = parent->parent\_.lock();  
 NodePtr uncle = (parent == grandparent->left\_) ? grandparent->right\_ : grandparent->left\_;  
  
 if (uncle->color\_ == Color::RED) {  
 // Случай 1: дядя красный.  
 parent->color\_ = Color::BLACK;  
 uncle->color\_ = Color::BLACK;  
 grandparent->color\_ = Color::RED;  
 pivot = grandparent;  
 } else {  
 if (parent == grandparent->left\_) {  
 // Случай 2: pivot — правый потомок.  
 if (pivot == parent->right\_) {  
 pivot = parent;  
 rotateLeft(pivot);  
 parent = pivot->parent\_.lock(); // Обновляем parent после поворота  
 }  
 // Случай 3: pivot — левый потомок.  
 parent->color\_ = Color::BLACK;  
 grandparent->color\_ = Color::RED;  
 rotateRight(grandparent);  
 } else {  
 // Зеркальные случаи для правого поддерева.  
 if (pivot == parent->left\_) {  
 pivot = parent;  
 rotateRight(pivot);  
 parent = pivot->parent\_.lock(); // Обновляем parent  
 }  
 parent->color\_ = Color::BLACK;  
 grandparent->color\_ = Color::RED;  
 rotateLeft(grandparent);  
 }  
 }  
 }  
 root->color\_ = Color::BLACK;  
}  
  
  
template<typename T, typename T0>  
typename RedBlackTree<T, T0>::NodePtr RedBlackTree<T, T0>::searchNode(const T &k) const {  
 if (root == *NIL*) return *NIL*;  
 NodePtr current = root;  
 while (current != *NIL*) {  
 if (current->key\_ == k) return current;  
 current = (current->key\_ > k) ? current->left\_ : current->right\_;  
 }  
 return *NIL*;  
}  
  
  
template<typename T, typename T0>  
void RedBlackTree<T, T0>::replaceSubtree(NodePtr oldNode, NodePtr newNode) {  
 if (oldNode == *NIL*) return;  
  
 if (auto parent = oldNode->parent\_.lock()) {  
 if (oldNode == parent->left\_) {  
 parent->left\_ = newNode;  
 } else {  
 parent->right\_ = newNode;  
 }  
 } else {  
 root = newNode;  
 }  
  
 if (newNode != *NIL*) {  
 newNode->parent\_ = oldNode->parent\_;  
 }  
}  
  
  
template<typename T, typename T0>  
typename RedBlackTree<T, T0>::NodePtr RedBlackTree<T, T0>::minimum(NodePtr subtreeRoot) {  
 if (subtreeRoot == *NIL*) return *NIL*;  
 while (subtreeRoot->left\_ != *NIL*) {  
 subtreeRoot = subtreeRoot->left\_;  
 }  
 return subtreeRoot;  
}  
  
template<typename T, typename T0>  
void RedBlackTree<T, T0>::fixDelete(NodePtr pivot) {  
 while (pivot != root && pivot->color\_ == Color::BLACK) {  
 auto parent = pivot->parent\_.lock();  
 if (!parent) break; // Защита от nullptr  
  
 bool isLeftChild = (pivot == parent->left\_);  
 NodePtr sibling = isLeftChild ? parent->right\_ : parent->left\_;  
  
 // Случай 1: Брат красный -> преобразуем в случаи 2-4  
 if (sibling != *NIL* && sibling->color\_ == Color::RED) {  
 sibling->color\_ = Color::BLACK;  
 parent->color\_ = Color::RED;  
 if (isLeftChild) {  
 rotateLeft(parent);  
 sibling = parent->right\_; // Обновляем брата после поворота  
 } else {  
 rotateRight(parent);  
 sibling = parent->left\_;  
 }  
 }  
  
 // Случай 2: Брат чёрный, оба потомка чёрные  
 if ((sibling->left\_ == *NIL* || sibling->left\_->color\_ == Color::BLACK) &&  
 (sibling->right\_ == *NIL* || sibling->right\_->color\_ == Color::BLACK)) {  
 if (sibling != *NIL*) {  
 sibling->color\_ = Color::RED;  
 }  
 pivot = parent; // Поднимаем проблему вверх  
 } else {  
 // Случай 3: Брат чёрный, ближний потомок красный  
 if (isLeftChild && (sibling->right\_ == *NIL* || sibling->right\_->color\_ == Color::BLACK)) {  
 if (sibling->left\_ != *NIL*) {  
 sibling->left\_->color\_ = Color::BLACK;  
 }  
 sibling->color\_ = Color::RED;  
 rotateRight(sibling);  
 sibling = parent->right\_;  
 } else if (!isLeftChild && (sibling->left\_ == *NIL* || sibling->left\_->color\_ == Color::BLACK)) {  
 if (sibling->right\_ != *NIL*) {  
 sibling->right\_->color\_ = Color::BLACK;  
 }  
 sibling->color\_ = Color::RED;  
 rotateLeft(sibling);  
 sibling = parent->left\_;  
 }  
  
 // Случай 4: Брат чёрный, дальний потомок красный  
 sibling->color\_ = parent->color\_;  
 parent->color\_ = Color::BLACK;  
 if (isLeftChild) {  
 if (sibling->right\_ != *NIL*) {  
 sibling->right\_->color\_ = Color::BLACK;  
 }  
 rotateLeft(parent);  
 } else {  
 if (sibling->left\_ != *NIL*) {  
 sibling->left\_->color\_ = Color::BLACK;  
 }  
 rotateRight(parent);  
 }  
 pivot = root; // Выход из цикла  
 }  
 }  
 pivot->color\_ = Color::BLACK; // Корень всегда чёрный  
}  
  
template<typename T, typename T0>  
typename RedBlackTree<T, T0>::NodePtr RedBlackTree<T, T0>::*createNIL*() {  
 auto nil = std::make\_shared<Node>(T{}, Color::BLACK);  
 nil->left\_ = nil->right\_ = nil; // Ссылки на себя  
 nil->parent\_ = WeakNodePtr{}; // Пустой родитель  
 return nil;  
}  
  
template<typename T, typename T0>  
void RedBlackTree<T, T0>::*quickSort*(Vector<Pair<std::string, int>> &vec) {  
 if (vec.get\_size() <= 1) return;  
 *quickSortImpl*(vec, 0, vec.get\_size() - 1);  
}  
  
template<typename T, typename T0>  
void RedBlackTree<T, T0>::*quickSortImpl*(Vector<Pair<std::string, int>> &vec, int low, int high) {  
 if (low < high) {  
 int pi = *partition*(vec, low, high);  
 *quickSortImpl*(vec, low, pi - 1);  
 *quickSortImpl*(vec, pi + 1, high);  
 }  
}  
  
template<typename T, typename T0>  
int RedBlackTree<T, T0>::*partition*(Vector<Pair<std::string, int>> &vec, int low, int high) {  
 int pivot = vec[high].second;  
 int i = low - 1;  
 for (int j = low; j < high; ++j) {  
 if (vec[j].second >= pivot) {  
 ++i;  
 std::swap(vec[i], vec[j]);  
 }  
 }  
 std::swap(vec[i + 1], vec[high]);  
 return i + 1;  
}  
  
// Инициализация статического члена NIL  
template<typename T, typename T0>  
typename RedBlackTree<T, T0>::NodePtr RedBlackTree<T, T0>::*NIL* = *createNIL*();  
  
#endif //REDBLACKTREE\_HPP

**Pair.hpp**

#ifndef PAIR\_HPP  
#define PAIR\_HPP  
  
template<typename T1, typename T2>  
struct Pair {  
 T1 first;  
 T2 second;  
 Pair(T1 first, T2 second): first(first), second(second) {}  
 Pair() = default;  
};  
  
#endif //PAIR\_HPP

**QueueList.hpp**

//  
// Created by jdh99 on 3/26/2025.  
//  
  
#ifndef QUEUELIST\_HPP  
#pragma once  
#define QUEUELIST\_HPP  
#include <stdexcept>  
#include <iterator>  
  
template<class T>  
struct Node {  
 T data;  
 Node \*next;  
  
 explicit Node(const T &data, Node \*next = nullptr) : data(data), next(next) {  
 }  
};  
  
class QueueOverflow : public std::runtime\_error {  
public:  
 QueueOverflow() : std::runtime\_error("Queue overflow") {  
 }  
};  
  
class QueueUnderflow : public std::runtime\_error {  
public:  
 QueueUnderflow() : std::runtime\_error("Queue underflow") {  
 }  
};  
  
template<class T>  
class Queue {  
public:  
 virtual ~Queue() = default;  
  
 virtual void enQueue(const T &e) = 0; // Добавление элемента в очередь  
 virtual T deQueue() = 0; // Удаление элемента из очереди  
  
 [[nodiscard]] virtual bool isEmpty() const noexcept = 0; // Проверка очереди на пустоту  
};  
  
template<typename T>  
class QueueListIterator {  
public:  
 using iterator\_category = std::forward\_iterator\_tag;  
 using value\_type = T;  
 using difference\_type = std::ptrdiff\_t;  
 using pointer = T \*;  
 using reference = T &;  
  
 explicit QueueListIterator(Node<T> \*node) : current\_(node) {  
 }  
  
 reference operator\*() const;  
  
 pointer operator->() const;  
  
 QueueListIterator &operator++();  
  
 QueueListIterator operator++(int);  
  
 bool operator==(const QueueListIterator &other) const { return current\_ == other.current\_; }  
 bool operator!=(const QueueListIterator &other) const { return !(\*this == other); }  
  
private:  
 Node<T> \*current\_;  
};  
  
  
template<class T>  
class QueueList final : public Queue<T> {  
public:  
 using iterator = QueueListIterator<T>;  
 using const\_iterator = QueueListIterator<const T>;  
  
 void enQueue(const T &e) override;  
  
 T deQueue() override;  
  
 [[nodiscard]] int size() const;  
  
 T front() const;  
  
 [[nodiscard]] bool isEmpty() const noexcept override;  
  
 ~QueueList() override;  
  
 iterator begin() noexcept { return iterator(head); }  
 iterator end() { return iterator(nullptr); }  
  
 const\_iterator begin() const noexcept { return const\_iterator(head); }  
 const\_iterator end() const { return const\_iterator(nullptr); }  
  
private:  
 Node<T> \*head = nullptr;  
 Node<T> \*tail = nullptr;  
 size\_t size\_ = 0;  
};  
  
  
template<typename T>  
typename QueueListIterator<T>::reference QueueListIterator<T>::operator\*() const {  
 if (!current\_) throw std::out\_of\_range("Dereferencing end iterator");  
 return current\_->data;  
}  
  
template<typename T>  
typename QueueListIterator<T>::pointer QueueListIterator<T>::operator->() const {  
 if (!current\_) throw std::out\_of\_range("Dereferencing end iterator");  
 return &current\_->data;  
}  
  
template<typename T>  
QueueListIterator<T> &QueueListIterator<T>::operator++() {  
 if (current\_) current\_ = current\_->next;  
 return \*this;  
}  
  
template<typename T>  
QueueListIterator<T> QueueListIterator<T>::operator++(int) {  
 QueueListIterator tmp = \*this;  
 ++(\*this);  
 return tmp;  
}  
  
template<class T>  
void QueueList<T>::enQueue(const T &e) {  
 try {  
 auto \*newNode = new Node<T>(e);  
 if (isEmpty()) {  
 head = tail = newNode;  
 } else {  
 tail->next = newNode;  
 tail = newNode;  
 }  
 size\_++;  
 } catch (std::bad\_alloc &) {  
 throw QueueOverflow();  
 }  
}  
  
template<class T>  
T QueueList<T>::deQueue() {  
 if (isEmpty()) {  
 throw QueueUnderflow();  
 }  
 T result = head->data;  
 Node<T> \*temp = head;  
 head = head->next;  
 if (head == nullptr) {  
 tail = nullptr;  
 }  
 delete temp;  
 size\_--;  
 return result;  
}  
  
template<class T>  
int QueueList<T>::size() const {  
 return size\_;  
}  
  
template<class T>  
T QueueList<T>::front() const {  
 return head->data;  
}  
  
  
template<class T>  
bool QueueList<T>::isEmpty() const noexcept { return head == nullptr; }  
  
template<class T>  
QueueList<T>::~QueueList() {  
 while (head) {  
 Node<T> \*temp = head;  
 head = head->next;  
 delete temp;  
 }  
}  
  
#endif //QUEUELIST\_HPP

**ReadFunc.hpp**

#include <cctype>  
#include <iostream>  
#include <fstream>  
#include "RedBlackTree.hpp"  
  
RedBlackTree<std::string> extractWordsFromFile(const std::string &filePath) {  
 RedBlackTree<std::string> words;  
 std::ifstream file(filePath);  
  
 if (!file.is\_open()) {  
 std::cerr << "Unable to open file: " << filePath << std::endl;  
 return words;  
 }  
  
 std::string word;  
 char ch;  
  
 while (file.get(ch)) {  
 if (isalpha(ch)) {  
 word += tolower(ch);  
 } else if (!word.empty()) {  
 words.insert(word);  
 word.clear();  
 }  
 }  
  
 if (!word.empty()) {  
 words.insert(word);  
 }  
  
 file.close();  
 return words;  
}

**StackArray.hpp**

#pragma once  
  
template<class T>  
class Stack {  
public:  
 virtual ~Stack() = default;  
  
 virtual void push(const T &e) = 0; // Добавление элемента в стек  
 virtual T pop() = 0; // Удаление верхнего элемента  
 [[nodiscard]] virtual bool isEmpty() const = 0; // Добавлен const  
 [[nodiscard]] virtual bool isFull() const = 0; // Новый метод  
 virtual const T& top() const = 0; // Новый метод  
};  
  
  
class WrongStackSize final : public std::exception {  
public:  
 [[nodiscard]] const char \*what() const noexcept override {  
 return "Wrong stack size";  
 }  
};  
  
class StackOverflow final : public std::exception {  
public:  
 [[nodiscard]] const char \*what() const noexcept override {  
 return "Stack overflow";  
 }  
};  
  
class StackUnderflow final : public std::exception {  
public:  
 [[nodiscard]] const char \*what() const noexcept override {  
 return "Stack underflow";  
 }  
};  
  
template<typename T>  
class StackArray final : public Stack<T> {  
 T \*data;  
 std::size\_t size\_;  
 std::size\_t capacity\_;  
  
public:  
 explicit StackArray(size\_t size);  
  
 StackArray(const StackArray &other)  
 : data(new T[other.capacity\_]),  
 size\_(other.size\_),  
 capacity\_(other.capacity\_) {  
 std::copy(other.data, other.data + size\_, data);  
 }  
  
 StackArray(StackArray &&other) noexcept:  
 data(other.data),  
 size\_(other.size\_),  
 capacity\_(other.capacity\_) {  
 other.data = nullptr;  
 other.size\_ = 0;  
 other.capacity\_ = 0;  
 }  
 [[nodiscard]] bool isFull() const override;  
  
 StackArray &operator=(const StackArray &other);  
  
 StackArray &operator=(StackArray &&other) noexcept;  
  
 void push(const T &e) override;  
  
 const T& top() const override;  
  
 T pop() override;  
  
 [[nodiscard]] bool isEmpty() const override { return size\_ == 0; }  
 ~StackArray() override { delete[] data; }  
};  
  
template<typename T>  
StackArray<T>::StackArray(const size\_t size) {  
 //Почему long long у size?  
 //std::size\_t не давал бы смысл для WrongStackSize (нет отрицательных чисел)  
 //int - не даёт возможности в bad\_alloc уйти  
  
 if (size <= 0) {  
 throw WrongStackSize();  
 }  
 try {  
 data = new T[size];  
 } catch (const std::bad\_alloc &) {  
 throw std::bad\_alloc();  
 }  
 capacity\_ = size;  
 size\_ = 0;  
}  
  
template<typename T>  
bool StackArray<T>::isFull() const {  
 return size\_ == capacity\_;  
}  
  
template<typename T>  
StackArray<T> & StackArray<T>::operator=(const StackArray &other) {  
 if (this != &other) {  
 delete[] data;  
 data = new T[other.capacity\_];  
 std::copy(other.data, other.data + other.size\_, data);  
 size\_ = other.size\_;  
 capacity\_ = other.capacity\_;  
 }  
 return \*this;  
}  
  
template<typename T>  
StackArray<T> & StackArray<T>::operator=(StackArray &&other) noexcept {  
 if (this != &other) {  
 delete[] data;  
 data = other.data;  
 size\_ = other.size\_;  
 capacity\_ = other.capacity\_;  
 other.data = nullptr;  
 other.size\_ = 0;  
 other.capacity\_ = 0;  
 }  
 return \*this;  
}  
  
  
template<typename T>  
void StackArray<T>::push(const T &e) {  
 if (capacity\_ == size\_) {  
 throw StackOverflow();  
 }  
 data[size\_++] = e;  
}  
  
template<typename T>  
const T& StackArray<T>::top() const {  
 if (isEmpty()) {  
 throw StackUnderflow();  
 }  
 return data[size\_ - 1];  
}  
  
template<typename T>  
T StackArray<T>::pop() {  
 if (this->isEmpty()) {  
 throw StackUnderflow();  
 }  
 return data[--size\_];  
}

**Vector.hpp**

#ifndef VECTOR\_HPP  
#define VECTOR\_HPP  
#include <stdexcept>  
#include <initializer\_list>  
  
template<typename T>  
class Vector {  
 std::size\_t size\_;  
 std::size\_t capacity\_;  
 T \*data\_;  
  
 void resize(std::size\_t new\_capacity);  
  
public:  
 class Iterator {  
 T \*ptr;  
  
 public:  
 explicit Iterator(T \*p);  
  
 T &operator\*() const;  
  
 T \*operator->() const;  
  
 Iterator &operator++();  
 bool operator!=(const Iterator& other) const;  
  
 Iterator operator++(int);  
  
 Iterator operator+(int i) const;  
 };  
  
 class ConstIterator {  
 const T\* ptr;  
  
 public:  
 explicit ConstIterator(const T\* p);  
  
 const T& operator\*() const;  
  
 const T\* operator->() const;  
  
 ConstIterator& operator++();  
  
 ConstIterator operator++(int);  
  
 bool operator==(const ConstIterator& other) const;  
  
 bool operator!=(const ConstIterator& other) const;  
  
 using difference\_type = std::ptrdiff\_t;  
 using value\_type = T;  
 using pointer = const T\*;  
 using reference = const T&;  
 using iterator\_category = std::forward\_iterator\_tag;  
 };  
  
 Vector();  
  
 explicit Vector(std::size\_t initial\_size);  
  
 Vector(std::initializer\_list<T> init)  
 : size\_(init.size()),  
 capacity\_(init.size()),  
 data\_(new T[capacity\_])  
 {  
 const T\* src = init.begin();  
 T\* dest = data\_;  
  
 for (size\_t i = 0; i < size\_; ++i) {  
 dest[i] = src[i];  
 }  
 }  
  
 ~Vector();  
  
 Vector(const Vector &other);  
  
 Vector &operator=(const Vector &other);  
  
 void push\_back(const T &value);  
  
 void pop\_back();  
  
 T &operator[](std::size\_t index);  
  
 const T &operator[](std::size\_t index) const;  
  
 std::size\_t get\_size() const;  
  
 std::size\_t get\_capacity() const;  
  
 bool empty() const;  
  
 Iterator begin() { return Iterator(data\_); }  
 Iterator end() { return Iterator(data\_ + size\_); }  
  
 ConstIterator begin() const { return ConstIterator(data\_); }  
 ConstIterator end() const { return ConstIterator(data\_ + size\_); }  
  
 ConstIterator cbegin() const { return ConstIterator(data\_); }  
 ConstIterator cend() const { return ConstIterator(data\_ + size\_); }  
  
 void clear();  
};  
  
template<typename T>  
void Vector<T>::resize(std::size\_t new\_capacity) {  
 T \*new\_data = new T[new\_capacity];  
 for (std::size\_t i = 0; i < size\_; ++i) {  
 new\_data[i] = data\_[i];  
 }  
 delete[] data\_;  
 data\_ = new\_data;  
 capacity\_ = new\_capacity;  
}  
  
template<typename T>  
Vector<T>::Iterator::Iterator(T \*p): ptr(p) {  
}  
  
template<typename T>  
T & Vector<T>::Iterator::operator\*() const { return \*ptr; }  
  
template<typename T>  
T \* Vector<T>::Iterator::operator->() const { return ptr; }  
  
template<typename T>  
typename Vector<T>::Iterator & Vector<T>::Iterator::operator++() {  
 ++ptr;  
 return \*this;  
}  
  
template<typename T>  
bool Vector<T>::Iterator::operator!=(const Iterator &other) const { return ptr != other.ptr; }  
  
template<typename T>  
typename Vector<T>::Iterator Vector<T>::Iterator::operator++(int) {  
 Iterator tmp = \*this;  
 ++ptr;  
 return tmp;  
}  
  
template<typename T>  
typename Vector<T>::Iterator Vector<T>::Iterator::operator+(int i) const {  
 return Iterator(ptr + i);  
}  
  
template<typename T>  
Vector<T>::ConstIterator::ConstIterator(const T \*p): ptr(p) {}  
  
template<typename T>  
const T & Vector<T>::ConstIterator::operator\*() const { return \*ptr; }  
  
template<typename T>  
const T \* Vector<T>::ConstIterator::operator->() const { return ptr; }  
  
template<typename T>  
typename Vector<T>::ConstIterator & Vector<T>::ConstIterator::operator++() {  
 ++ptr;  
 return \*this;  
}  
  
template<typename T>  
typename Vector<T>::ConstIterator Vector<T>::ConstIterator::operator++(int) {  
 ConstIterator tmp = \*this;  
 ++ptr;  
 return tmp;  
}  
  
template<typename T>  
bool Vector<T>::ConstIterator::operator==(const ConstIterator &other) const {  
 return ptr == other.ptr;  
}  
  
template<typename T>  
bool Vector<T>::ConstIterator::operator!=(const ConstIterator &other) const {  
 return ptr != other.ptr;  
}  
  
template<typename T>  
Vector<T>::Vector(): data\_(nullptr), capacity\_(0), size\_(0) {  
}  
  
template<typename T>  
Vector<T>::Vector(std::size\_t initial\_size): capacity\_(initial\_size), size\_(initial\_size) {  
 data\_ = new T[capacity\_];  
}  
  
template<typename T>  
Vector<T>::~Vector() {  
 delete[] data\_;  
}  
  
template<typename T>  
Vector<T>::Vector(const Vector &other): capacity\_(other.capacity\_), size\_(other.size\_) {  
 data\_ = new T[capacity\_];  
 for (std::size\_t i = 0; i < size\_; ++i) {  
 data\_[i] = other.data\_[i];  
 }  
}  
  
template<typename T>  
Vector<T> &Vector<T>::operator=(const Vector &other) {  
 if (this != &other) {  
 delete[] data\_;  
 capacity\_ = other.capacity\_;  
 size\_ = other.size\_;  
 data\_ = new T[capacity\_];  
 for (size\_t i = 0; i < size\_; ++i) {  
 data\_[i] = other.data\_[i];  
 }  
 }  
 return \*this;  
}  
  
template<typename T>  
void Vector<T>::push\_back(const T &value) {  
 if (size\_ >= capacity\_) {  
 resize(capacity\_ == 0 ? 1 : capacity\_ \* 2);  
 }  
 data\_[size\_++] = value;  
}  
  
template<typename T>  
void Vector<T>::pop\_back() {  
 if (size\_ == 0) {  
 throw std::out\_of\_range("Vector is empty");  
 }  
 --size\_;  
}  
  
template<typename T>  
T &Vector<T>::operator[](std::size\_t index) {  
 if (index >= size\_) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return data\_[index];  
}  
  
template<typename T>  
const T &Vector<T>::operator[](std::size\_t index) const {  
 if (index >= size\_) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return data\_[index];  
}  
  
template<typename T>  
std::size\_t Vector<T>::get\_size() const {  
 return size\_;  
}  
  
template<typename T>  
std::size\_t Vector<T>::get\_capacity() const {  
 return capacity\_;  
}  
  
template<typename T>  
bool Vector<T>::empty() const {  
 return size\_ == 0;  
}  
  
template<typename T>  
void Vector<T>::clear() {  
 size\_ = 0;  
}  
  
#endif //VECTOR\_HPP