**Лабораторная работа**

**Красно-чёрное дерево.  Индексирования записей в БД**

# **Цель**

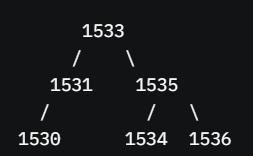
Освоить красно-чёрные деревья как эффективную структуру данных для организации индексов в базах данных, изучить их алгоритмы и реализовать на языке C++. Студенты научатся применять красно-чёрные деревья для ускорения операций поиска, вставки и удаления данных, разрабатывать схемы баз данных и создавать программы, обеспечивающие оптимальное взаимодействие с данными через индексирование.

# **Теория**

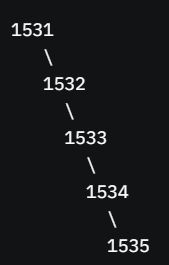
## **2.1. Повторение**

Для начала вспомним, что такое структура данных — это способ, которым компьютер организует информацию, чтобы с ней было удобно работать. Простыми словами, это как ящики в шкафу: в одном ящике вещи лежат в куче, в другом — аккуратно по полочкам. От того, как ты организуешь данные, зависит, как быстро ты сможешь что-то найти или добавить. Теперь вспомним простое дерево – это бинарного дерева поиска (BST). Это структура, которая выглядит как дерево с ветками, где:

* Каждый "узел" (точка на дереве) имеет значение, например, IdStudent.
* У каждого узла может быть до двух "детей": левый и правый.
* Все значения в левом поддереве меньше значения узла.
* Все значения в правом поддереве больше значения узла.



Ну вот мы и повторили лёгкую теорию, но зачем нам использовать красно-чёрное дерево , а не простое – бинарное? Проблема в том, что если добавлять элементы в определённом порядке, дерево может "испортиться". К примеру добавляем студентов в порядке возрастания: 1531, 1532, 1533, 1534, 1535. Тогда BST превратится в длинную цепочку:



Это уже не дерево, а список! Теперь, чтобы найти 1535, нужно пройти все узлы от начала до конца — это медленно, как если бы мы просто перебирали таблицу без индекса.

Для баз данных это катастрофа, потому что данные могут добавляться в любом порядке, и нам нужно, чтобы структура всегда оставалась быстрой. Поэтому обычное BST не подходит — нам нужна структура, которая сама себя балансирует.

## **2.2. Красно-чёрное дерево: почему такое название и как оно работает?**

Красно-чёрное дерево — это усовершенствованная версия бинарного дерева поиска, которая всегда остаётся сбалансированной. Оно не превращается в длинную цепочку, а сохраняет "широкую" и "невысокую" форму, обеспечивая логарифмическую сложность операций ( O(log n) ).

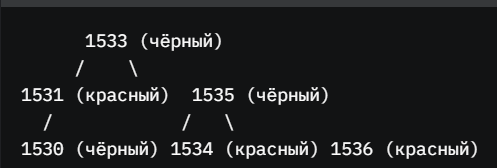
Почему такое название?  
Каждый узел красно-чёрного дерева имеет цвет — либо красный, либо чёрный. Эти цвета используются как инструмент для поддержания баланса. Название "красно-чёрное" отражает эту ключевую особенность, введённую для того, чтобы дерево не теряло свою эффективность даже при интенсивных операциях вставки и удаления.

### **2.2.1. Правила красно-чёрного дерева:**

1. Каждый узел — красный или чёрный. Это основа системы балансировки.
2. Корень всегда чёрный. Это упрощает контроль высоты дерева.
3. Все листья — чёрные NULL-узлы. В отличие от обычного BST, где лист — это узел без детей, в красно-чёрном дереве листья — это несуществующие узлы (NULL), которые считаются чёрными по умолчанию.
4. У красного узла не может быть красных детей. Два красных узла не могут стоять рядом (ни родитель, ни ребёнок).
5. Чёрная высота одинакова на всех путях. От любого узла до его листьев (NULL-узлов) количество чёрных узлов должно быть одинаковым.

Что такое чёрная высота?  
Чёрная высота — это количество чёрных узлов на любом пути от данного узла до листа (NULL), причём сам NULL считается чёрным, а исходный узел — нет. Если в какую-то сторону количество чёрных узлов отличается, баланс нарушается, и дерево перестраивается.

Пример красно-чёрного дерева:



Здесь цвета помогают держать дерево в балансе. Если правила нарушаются, дерево "перестраивается" само. Эти правила гарантируют, что самый длинный путь от корня до листа не более чем в два раза длиннее самого короткого, а операции поиска, вставки и удаления выполняются за O(log n).

### **Вставка в красно-чёрное дерево**

При добавлении нового узла:

* Новый узел вставляется как красный. Это важно, потому что вставка красного узла в среднем нарушает меньше правил, чем вставка чёрного, которая сразу меняет чёрную высоту родителя.
* Находится свободное место (слева или справа) с учётом правил BST.
* Добавляется ссылка на родителя, что позволяет обойтись без рекурсии и использовать цикл для поиска места вставки.

### **Случаи балансировки дерева**

**Вставка (5 случаев):**

* **Случай 1:** Новый узел — корень (перекрашиваем в чёрный).
* **Случай 2:** Красный родитель и красный дядя (перекрашиваем родителя и дядю в чёрный, деда в красный, проверяем выше).
* **Случай 3:** Красный родитель, чёрный дядя, дед — корень (перекрашиваем родителя и дядю в чёрный, корень остаётся чёрным).
* **Случай 4:** Красный родитель, чёрный дядя, "зигзаг" (левый поворот, переходим к случаю 5).
* **Случай 5:** Красный родитель, чёрный дядя, прямая линия (перекрашиваем родителя в чёрный, деда в красный, правый поворот).

**Удаление (5 основных случаев + 3 подслучая):**

* **Случай 1:** Брат чёрный, у брата два красных ребёнка (перекрашиваем брата в цвет родителя, детей в чёрный, левый поворот).
* **Случай 2:** Брат чёрный, у брата один красный ребёнок слева и один чёрный справа (аналогично случаю 1).
* **Случай 3:** Брат чёрный, у брата один красный ребёнок справа и один чёрный слева (перекрашиваем брата в красный, левого ребёнка в чёрный, правый поворот, переходим к случаю 1 или 2).
* **Случай 4:** Брат чёрный, у брата два чёрных ребёнка:
  + **4.1:** Родитель — чёрный корень (перекрашиваем брата в красный).
  + **4.2:** Родитель — красный, не корень (перекрашиваем брата в красный, родителя в чёрный).
  + **4.3:** Родитель — чёрный, не корень (перекрашиваем родителя в красный, левый поворот, переходим к другим случаям).
* **Случай 5:** Брат красный (перекрашиваем брата в чёрный, родителя в красный, левый поворот).

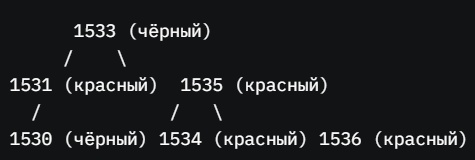
### **2.1.4. Какие случаи балансировки самые частые?**

Частота случаев балансировки в красно-чёрных деревьях зависит от структуры дерева, порядка добавления или удаления узлов и распределения данных. В сбалансированном красно-чёрном дереве примерно половина узлов красные, а половина — чёрные, что позволяет оценить вероятность возникновения различных сценариев. Давайте разберём самые частые случаи для вставки и удаления с примерами деревьев.

**Вставка: Самые частые случаи**При вставке нового узла (всегда красного) балансировка требуется, если у нового узла красный родитель, что нарушает правило "нет двух красных узлов подряд". Самые частые случаи:

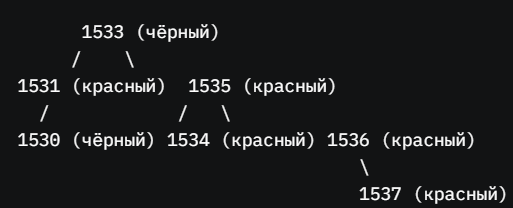
1. **Красный родитель и красный дядя** (~25% вероятность):

* **Условие:** Родитель нового узла красный, дядя (другой ребёнок деда) тоже красный.
* **Действие:** Родитель и дядя становятся чёрными, дед — красным, проверяем вышестоящие узлы.
* **Пример:**
  + Исходное дерево:

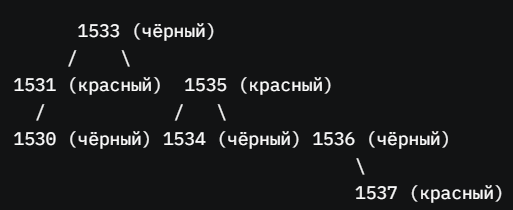


* Вставляем 1537 (правый ребёнок 1536):

Дерево становится:

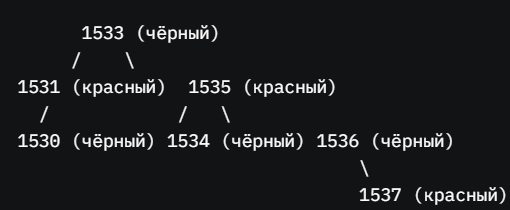


После перекрашивания:

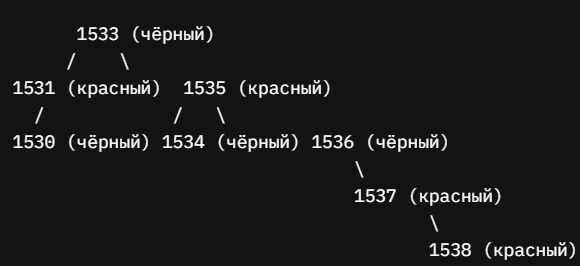


1. **Красный родитель, чёрный дядя, "прямая линия"** (~12.5%):

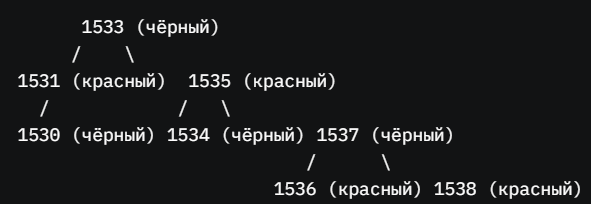
* **Условие:** Родитель красный, дядя чёрный, новый узел и родитель на одной стороне деда.
* **Действие:** Родитель становится чёрным, дед — красным, делаем правый поворот.
* **Пример:**
  + Исходное дерево (после предыдущего случая):



Вставляем 1538 (правый ребёнок 1537)

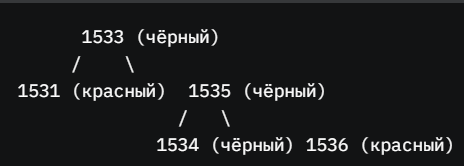


После перекрашивания и поворота:

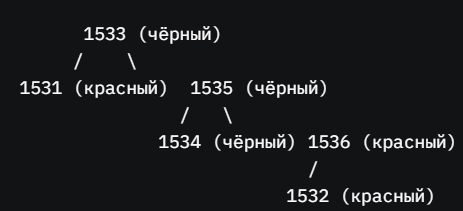


1. **Красный родитель, чёрный дядя, "зигзаг"** (~12.5%):

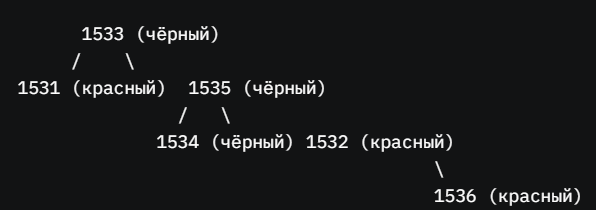
* **Условие:** Родитель красный, дядя чёрный, новый узел и родитель на разных сторонах деда.
* **Действие:** Левый поворот, затем случай "прямая линия".
* **Пример:**
  + Исходное дерево:



Вставляем 1532 (левый ребёнок 1536):



После левого поворота:

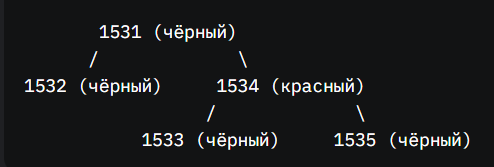


**Удаление: Самые частые случаи**

Балансировка нужна, если удаляется чёрный узел (вероятность 50%), чтобы сохранить чёрную высоту. Частые случаи:

1. **Брат красный** (~50% случаев с балансировкой):

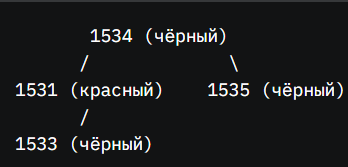
* **Условие:** Брат удаляемого узла красный.
* **Действие:** Брат становится чёрным, родитель — красным, делаем левый поворот.
* **Пример:**
  + - Исходное дерево:



* Удаляем узел 1532 (чёрный, лист). Заменяем его на null. Брат удалённого узла (1534) красный.
* **Шаги**:

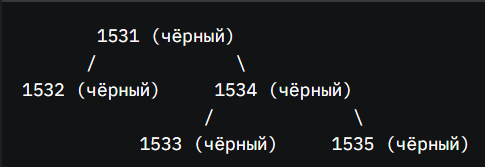
1. Брат 1534 становится чёрным, родитель 1531 — красным.
2. Выполняем левый поворот вокруг 1531.

* **Дерево после поворота**



**2. Брат чёрный, оба племянника чёрные (~25% случаев с балансировкой)**

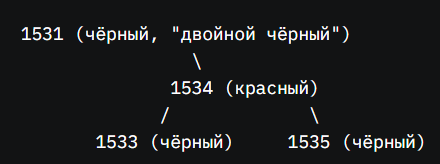
* **Условие**: Брат удаляемого узла чёрный, и оба его ребёнка (племянники) тоже чёрные (или отсутствуют).
* **Действие**: Брат становится красным, родитель требует проверки (может стать "двойным чёрным").
* **Пример**:
  + Исходное дерево:



* Удаляем узел 1532 (чёрный, лист). Заменяем его на null. Брат (1534) чёрный, его дети (1533 и 1535) тоже чёрные.
* **Шаги**:

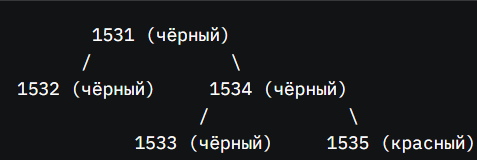
1. Брат 1534 становится красным.
2. Родитель 1531 становится "двойным чёрным".

* **Дерево после**:



1. **Брат чёрный, дальний племянник красный (~12.5% случаев)**

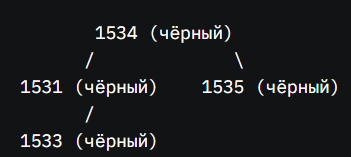
* **Условие**: Брат чёрный, а его ребёнок, находящийся дальше от удалённого узла (дальний племянник), красный.
* **Действие**: Выполняется поворот, брат принимает цвет родителя, родитель и другой племянник становятся чёрными.
* **Пример**:
  + Исходное дерево:



* Удаляем узел 1532 (чёрный, лист). Брат (1534) чёрный, дальний племянник (1535) красный.
* **Шаги**:

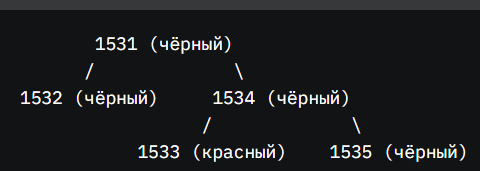
1. Выполняем левый поворот вокруг 1531.
2. Устанавливаем 1534 цветом родителя (чёрный), 1531 — чёрным, 1535 — чёрным.

* **Дерево после**:



1. **Брат чёрный, ближний племянник красный (~12.5% случаев)**

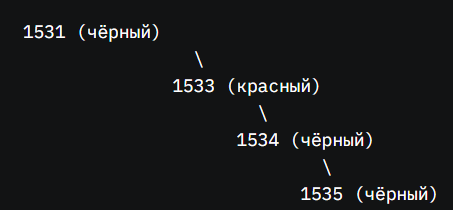
* **Условие**: Брат чёрный, а его ребёнок, находящийся ближе к удалённому узлу (ближний племянник), красный.
* **Действие**: Выполняется правый поворот на брате, затем случай сводится к "дальний племянник красный".
* **Пример**:
  + Исходное дерево:



* Удаляем узел 1532. Брат (1534) чёрный, ближний племянник (1533) красный.
* **Шаги**:

1. Выполняем правый поворот вокруг 1534, 1533 становится новым братом и красным.

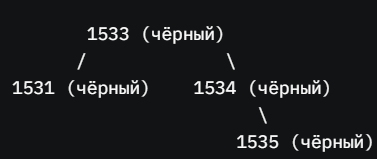
* **Дерево после правого поворота**:



* Затем применяем шаги для "дальний племянник красный":

1. Выполняем левый поворот вокруг 1531.
2. Устанавливаем 1533 цветом родителя (чёрный), 1531 и 1534 — чёрными.

* **Итоговое дерево**:



### **Шутка минутка :**

|  |
| --- |
| — Что делаешь?  — Перекрашиваю чёрных детей.  — Расист?!  — Программист. |

# **Применение красно-чёрного дерева в базах данных**

## **3.1. Зачем нужны индексы в базах данных?**

Базы данных содержат огромное количество записей, например, информацию о студентах (ID, имя, оценки и т.д.). Без правильной организации поиск нужной записи, например, студента с определённым ID, может быть очень медленным — пришлось бы перебирать все записи по очереди (сложность O(n)). Чтобы ускорить операции поиска, вставки и удаления, базы данных используют индексы.

Индекс — это структура данных, которая хранит пары "ключ-значение" (например, ID студента и ссылку на его запись в таблице) и позволяет быстро находить нужные данные. Однако индексы должны быть эффективными: они должны минимизировать время доступа и поддерживать баланс между скоростью операций и затратами памяти.

## **3.2. Почему красно-чёрное дерево подходит для индексирования?**

Красно-чёрное дерево идеально подходит для создания индексов в базах данных по нескольким причинам:

1. **Логарифмическая сложность операций**:

* Поиск, вставка и удаление в красно-чёрном дереве выполняются за O(log n), где n — количество записей. Это гораздо быстрее, чем O(n) для линейного поиска или O(n log n) для неподходящих структур.
* Например, если база данных содержит миллион записей, поиск в красно-чёрном дереве займёт примерно 20 шагов (log₂(1,000,000) ≈ 20), тогда как линейный поиск может потребовать до миллиона операций.

1. **Автоматическая балансировка**:

* В отличие от обычного бинарного дерева поиска, которое может превратиться в список, красно-чёрное дерево всегда остаётся сбалансированным благодаря своим правилам (см. раздел 2.2.1). Это гарантирует стабильную производительность даже при случайном или упорядоченном добавлении данных.

1. **Эффективное обновление**:

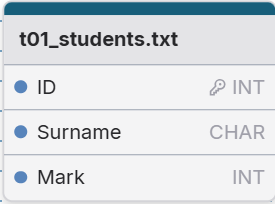
* Базы данных часто обновляются: добавляются новые записи, удаляются старые или изменяются существующие. Красно-чёрное дерево позволяет выполнять эти операции с минимальными перестройками структуры, так как балансировка требует лишь локальных изменений (поворотов и перекрашиваний).

1. **Гибкость с ключами**:

* Индексы могут строиться по любому полю таблицы (например, ID, фамилия, дата рождения). Красно-чёрное дерево поддерживает сравнение ключей любого типа, если задано отношение порядка (например, числовое или лексикографическое).

## **3.3. Пример использования в базе данных**

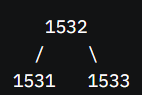
Рассмотрим таблицу студентов:



Чтобы ускорить поиск по ID, база данных создаёт индекс в виде красно-чёрного дерева, где:

* Узлы содержат ID студента (ключ) и ссылку на запись в таблице.
* Дерево организовано так, что ID слева меньше, а справа больше.

Пример структуры индекса:



* **Поиск**: Чтобы найти студента с ID 1533, база данных начинает с корня (1532), видит, что 1533 > 1532, переходит вправо и сразу находит нужный узел. Это занимает всего 2 шага.
* **Вставка**: Добавление нового студента с ID 1534 требует поиска места в дереве, вставки нового узла и, возможно, балансировки (см. раздел 2.1.2). Дерево остаётся сбалансированным.
* **Удаление**: Удаление студента с ID 1531 требует замены узла и балансировки, чтобы сохранить чёрную высоту (см. раздел 2.1.3).

## **3.4. Сравнение с другими структурами данных**

Красно-чёрные деревья не единственный выбор для индексов. Рассмотрим альтернативы:

* **Хеш-таблицы**:
* Плюсы: Поиск за O(1) в среднем.
* Минусы: Не поддерживают диапазонные запросы (например, "найти всех студентов с ID от 1531 до 1535"). Требуют больше памяти при коллизиях.
* Когда использовать: Для точечных поисков, где диапазоны не нужны.
* **B-деревья и B+-деревья**:
* Плюсы: Оптимизированы для работы с дисками, где доступ к памяти дорог. Хранят больше ключей в одном узле, уменьшая количество обращений к диску.
* Минусы: Более сложная реализация, чем красно-чёрное дерево.
* Когда использовать: В больших базах данных (например, MySQL, PostgreSQL), где данные хранятся на диске.
* **Обычные бинарные деревья поиска**:
* Плюсы: Простая реализация.
* Минусы: Могут выродиться в список, что приводит к O(n) для операций.
* Когда использовать: Только для небольших и заранее сбалансированных наборов данных.

Красно-чёрное дерево — это компромисс: оно проще, чем B-деревья, но обеспечивает надёжную производительность для большинства операций в памяти. В современных СУБД (например, SQLite или Redis) красно-чёрные деревья иногда используются для индексов в оперативной памяти или для специфичных задач, таких как кэширование или временные индексы.

## **3.5. Ограничения красно-чёрного дерева**

Несмотря на свои преимущества, красно-чёрные деревья имеют ограничения:

* **Сложность реализации**: По сравнению с хеш-таблицами или простыми списками, алгоритмы вставки и удаления сложнее из-за необходимости балансировки.
* **Дисковые операции**: Для больших баз данных, хранящихся на диске, красно-чёрные деревья менее эффективны, чем B-деревья, так как каждый узел хранит только один ключ, что увеличивает количество дисковых операций.
* **Диапазонные запросы**: Хотя красно-чёрное дерево поддерживает диапазонные запросы (например, через обход поддерева), оно менее оптимизировано для этого по сравнению с B+-деревьями, где ключи хранятся в листьях.

В данной лабораторной работе мы будем использовать обычный текстовый файл для хранения данных, имитируя работу с базой данных. Такой подход позволяет реализовать основные операции индексирования, поиска и обновления записей без использования полноценной СУБД. Каждая запись будет храниться в отдельной строке файла, а для ускорения доступа и поддержки эффективного поиска будет использоваться структура красно-чёрного дерева, построенная на основе ключей записей.

# **Типовой пример**

В рамках примера мы реализуем программу, которая имитирует базу данных студентов, хранящихся в текстовом файле students.txt.

Цель реализации — продемонстрировать использование красно-чёрного дерева для индексирования записей по полю IdStudent, обеспечивая эффективное выполнение операций поиска, вставки и удаления с логарифмической сложностью O(log n). Программа должна:

* Загружать данные из файла в память.
* Создавать индекс на основе красно-чёрного дерева.
* Поддерживать операции:
  + **Поиск**: нахождение студента по IdStudent.
  + **Вставка**: добавление нового студента в файл и индекс.
  + **Удаление**: удаление студента из индекса и обновление файла.

Вот содержимое файла students.txt :

Изображение выглядит как текст, снимок экрана, Шрифт, число

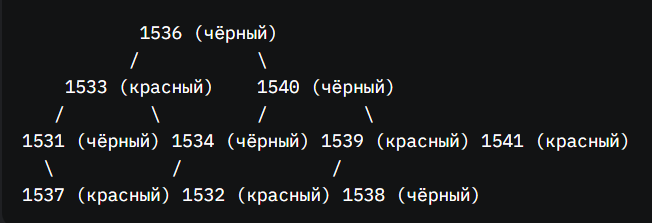
Контент, сгенерированный ИИ, может содержать ошибки.

Для простоты удаление реализовано как физическое исключение записи из файла путём его переписывания. Все операции в красно-чёрном дереве выполняются с учётом правил (корень чёрный, нет двух красных узлов подряд, одинаковая чёрная высота), что гарантирует сбалансированность и эффективность.

Код программы:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #include <iostream>  #include <fstream>  #include <string>  #include <sstream>  #include <vector>  #include <algorithm>  #include <locale>  #include <windows.h>  using namespace std;  // Структура для хранения данных студента  struct Student {  int id;  string surname;  string name;  // Конструктор структуры Student без значений по умолчанию и списка инициализации  Student(int \_id, string \_surname, string \_name) {  id = \_id;  surname = \_surname;  name = \_name;  }  };  // RED и BLACK — два возможных цвета узлов (красный и чёрный)  enum Color { RED, BLACK };  // Структура для узла красно-чёрного дерева  struct RBNode {  int key; // Ключ узла (в нашем случае IdStudent)  Student\* data; // Указатель на данные студента (объект Student)  RBNode\* left; // Указатель на левого потомка  RBNode\* right; // Указатель на правого потомка  RBNode\* parent; // Указатель на родительский узел  Color color; // Цвет узла (RED или BLACK)  // Конструктор узла красно-чёрного дерева без списка инициализации  RBNode(int k, Student\* d) {  key = k;  data = d;  left = nullptr;  right = nullptr;  parent = nullptr;  color = RED;  }  };  struct RBTree {  RBNode\* root; // Указатель на корень дерева  RBNode\* nil; // Специальный узел "nil" (листовой узел), используется вместо nullptr  // Метод для левого поворота (используется для балансировки дерева)  void leftRotate(RBNode\* x) {  RBNode\* y = x->right; // y — правый потомок узла x  x->right = y->left; // Левый потомок y становится правым потомком x  if (y->left != nil) y->left->parent = x;  y->parent = x->parent; // Родитель x становится родителем y  if (x->parent == nullptr) root = y;  else if (x == x->parent->left) x->parent->left = y; // Если x был левым потомком, обновляем указатель  else x->parent->right = y; // Если x был правым потомком, обновляем указатель  y->left = x;  x->parent = y;  }  // Метод для правого поворота (аналогично левому, но в другую сторону)  void rightRotate(RBNode\* y) {  RBNode\* x = y->left; // x — левый потомок узла y  y->left = x->right; // Правый потомок x становится левым потомком y  if (x->right != nil) x->right->parent = y;  x->parent = y->parent;  if (y->parent == nullptr) root = x; // Если y был корнем, x становится новым корнем  else if (y == y->parent->right) y->parent->right = x; // Если y был правым потомком, обновляем указатель  else y->parent->left = x;  x->right = y;  y->parent = x;  }  // Метод для исправления свойств красно-чёрного дерева после вставки  // z — новый узел, который был вставлен  void fixInsert(RBNode\* z) {  // Пока родитель z красный (нарушение свойства: два красных узла не могут быть соседними)  while (z->parent && z->parent->color == RED) {  if (z->parent == z->parent->parent->left) { // Если родитель z — левый потомок деда  RBNode\* y = z->parent->parent->right; // y — дядя z (правый потомок деда)  if (y->color == RED) { // Случай 1: дядя красный  z->parent->color = BLACK; // Родитель становится чёрным  y->color = BLACK; // Дядя становится чёрным  z->parent->parent->color = RED; // Дед становится красным  z = z->parent->parent; // Переходим к деду  }  else { // Случай 2 или 3: дядя чёрный  if (z == z->parent->right) { // Случай 2: z — правый потомок (треугольник)  z = z->parent; // Поднимаемся к родителю  leftRotate(z); // Выполняем левый поворот  }  // Случай 3: z — левый потомок (линия)  z->parent->color = BLACK; // Родитель становится чёрным  z->parent->parent->color = RED; // Дед становится красным  rightRotate(z->parent->parent); // Выполняем правый поворот  }  }  else { // Симметричный случай: родитель z — правый потомок деда  RBNode\* y = z->parent->parent->left; // y — дядя z (левый потомок деда)  if (y->color == RED) { // Случай 1: дядя красный  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else { // Случай 2 или 3: дядя чёрный  if (z == z->parent->left) { // Случай 2: z — левый потомок (треугольник)  z = z->parent;  rightRotate(z); // Выполняем правый поворот  }  // Случай 3: z — правый потомок (линия)  z->parent->color = BLACK;  z->parent->parent->color = RED;  leftRotate(z->parent->parent); // Выполняем левый поворот  }  }  if (z == root) break; // Если z стал корнем, прекращаем  }  root->color = BLACK; // Корень всегда должен быть чёрным  }  // Метод для поиска минимального узла в поддереве (нужен для удаления)  RBNode\* minimum(RBNode\* node) {  while (node->left != nil) node = node->left; // Идём влево до конца  return node;  }  // Метод для замены одного поддерева на другое (нужен для удаления)  void transplant(RBNode\* u, RBNode\* v) {  if (u->parent == nullptr) root = v; // Если u — корень, v становится новым корнем  else if (u == u->parent->left) u->parent->left = v; // Если u — левый потомок, заменяем на v  else u->parent->right = v; // Если u — правый потомок, заменяем на v  v->parent = u->parent; // Обновляем родителя v  }  // Метод для исправления свойств красно-чёрного дерева после удаления  // x — узел, который может нарушить свойства дерева  void fixDelete(RBNode\* x) {  while (x != root && x->color == BLACK) { // Пока x не корень и x чёрный  if (x == x->parent->left) { // Если x — левый потомок  RBNode\* w = x->parent->right; // w — брат x (правый потомок родителя)  if (w->color == RED) { // Случай 1: брат красный  w->color = BLACK;  x->parent->color = RED;  leftRotate(x->parent);  w = x->parent->right;  }  if (w->left->color == BLACK && w->right->color == BLACK) { // Случай 2: оба потомка брата чёрные  w->color = RED;  x = x->parent;  }  else { // Случай 3 или 4: хотя бы один потомок брата красный  if (w->right->color == BLACK) { // Случай 3: правый потомок брата чёрный  w->left->color = BLACK;  w->color = RED;  rightRotate(w);  w = x->parent->right;  }  // Случай 4: правый потомок брата красный  w->color = x->parent->color;  x->parent->color = BLACK;  w->right->color = BLACK;  leftRotate(x->parent);  x = root;  }  }  else { // Симметричный случай: x — правый потомок  RBNode\* w = x->parent->left; // w — брат x (левый потомок родителя)  if (w->color == RED) { // Случай 1: брат красный  w->color = BLACK;  x->parent->color = RED;  rightRotate(x->parent);  w = x->parent->left;  }  if (w->right->color == BLACK && w->left->color == BLACK) { // Случай 2: оба потомка брата чёрные  w->color = RED;  x = x->parent;  }  else { // Случай 3 или 4: хотя бы один потомок брата красный  if (w->left->color == BLACK) { // Случай 3: левый потомок брата чёрный  w->right->color = BLACK;  w->color = RED;  leftRotate(w);  w = x->parent->left;  }  // Случай 4: левый потомок брата красный  w->color = x->parent->color;  x->parent->color = BLACK;  w->left->color = BLACK;  rightRotate(x->parent);  x = root;  }  }  }  x->color = BLACK; // Устанавливаем x в чёрный цвет  }  // Конструктор структуры RBTree  RBTree() {  nil = new RBNode(0, nullptr); // Создаём специальный узел nil  nil->color = BLACK; // nil всегда чёрный  nil->left = nil->right = nil; // У nil нет потомков  root = nil; // Изначально дерево пустое, корень — nil  }  // Деструктор структуры RBTree  // Освобождаем память, занятую узлом nil  // Примечание: в реальной программе нужно освобождать память всех узлов дерева рекурсивно  ~RBTree() {  delete nil;  }  // Метод для вставки нового узла в дерево  // key — ключ (IdStudent), data — указатель на данные студента  void insert(int key, Student\* data) {  RBNode\* z = new RBNode(key, data);  z->left = nil;  z->right = nil;  RBNode\* y = nullptr; // y — будущий родитель нового узла  RBNode\* x = root; // x — текущий узел для поиска места вставки  // Ищем место для вставки (как в бинарном дереве поиска)  while (x != nil) {  y = x;  if (z->key < x->key)  x = x->left; // Если ключ меньше, идём влево  else  x = x->right; // Если ключ больше или равен, идём вправо  }  z->parent = y; // Устанавливаем родителя нового узла  if (y == nullptr) root = z; // Если дерево пустое, z становится корнем  else if (z->key < y->key) y->left = z; // Если ключ меньше, z — левый потомок  else y->right = z; // Если ключ больше или равен, z — правый потомок  fixInsert(z); // Исправляем свойства красно-чёрного дерева после вставки  }  // Метод для поиска узла по ключу  // Возвращает указатель на данные студента или nullptr, если узел не найден  Student\* search(int key) {  RBNode\* current = root;  while (current != nil) {  if (key == current->key)  return current->data;  else if (key < current->key)  current = current->left;  else  current = current->right;  }  return nullptr; // Если узел не найден, возвращаем nullptr  }  // Метод для удаления узла по ключу  void remove(int key) {  RBNode\* z = root;  while (z != nil) { // Ищем узел с заданным ключом  if (key == z->key) break;  else if (key < z->key) z = z->left;  else z = z->right;  }  if (z == nil) return; // Если узел не найден, выходим  RBNode\* y = z; // y — узел, который будет удалён  Color y\_original\_color = y->color; // Сохраняем цвет удаляемого узла  RBNode\* x; // x — узел, который займёт место удаляемого  if (z->left == nil) { // Случай 1: у z нет левого потомка  x = z->right;  transplant(z, z->right); // Заменяем z на его правого потомка  }  else if (z->right == nil) { // Случай 2: у z нет правого потомка  x = z->left;  transplant(z, z->left); // Заменяем z на его левого потомка  }  else { // Случай 3: у z есть оба потомка  y = minimum(z->right); // Находим минимальный узел в правом поддереве (преемник)  y\_original\_color = y->color; // Сохраняем цвет преемника  x = y->right; // x — правый потомок преемника  if (y->parent == z) {  x->parent = y; // Если преемник — прямой потомок z  }  else {  transplant(y, y->right); // Заменяем преемника на его правого потомка  y->right = z->right; // Присоединяем правое поддерево z к преемнику  y->right->parent = y;  }  transplant(z, y); // Заменяем z на преемника  y->left = z->left; // Присоединяем левое поддерево z к преемнику  y->left->parent = y;  y->color = z->color; // Копируем цвет z в преемника  }  if (y\_original\_color == BLACK) // Если удалённый узел был чёрным, исправляем свойства дерева  fixDelete(x);  delete z;  }  };  // Структура для работы с базой данных студентов (замена класса StudentDB)  struct StudentDB {  RBTree index; // Красно-чёрное дерево для индексирования студентов по IdStudent  vector<Student\*> students; // Вектор указателей на объекты Student (хранит всех студентов)  string inputFilename; // Имя входного файла (t01\_Student.txt)  string outputFilename; // Имя выходного файла (t01\_Student\_updated.txt)  // Метод для записи обновлённого списка студентов в выходной файл  void writeToOutputFile() {  ofstream file(outputFilename);  if (!file.is\_open()) {  cout << "Ошибка записи в выходной файл!" << endl;  return;  }  file << "IdStudent\tSurname\tName\n";  for (Student\* s : students) {  file << s->id << "\t" << s->surname << "\t" << s->name << "\n";  }  file.close();  }  // Конструктор структуры StudentDB  StudentDB(string inFile, string outFile) {  inputFilename = inFile;  outputFilename = outFile;  loadFromFile();  }  // Деструктор структуры StudentDB  ~StudentDB() {  for (Student\* s : students) delete s;  }  // Метод для загрузки данных из входного файла  void loadFromFile() {  ifstream file(inputFilename);  if (!file.is\_open()) {  cout << "Ошибка: Не удалось открыть файл " << inputFilename << "!" << endl;  return;  }  string line; // Переменная для чтения строк из файла  int count = 0; // Счётчик загруженных записей  getline(file, line); // Пропускаем строку заголовка (IdStudent Surname Name)  while (getline(file, line)) { // Читаем файл построчно  if (line.empty()) continue;  // Создаём объект stringstream для парсинга строки line.  // stringstream позволяет работать со строкой как с потоком ввода (аналогично cin).  // В данном случае строка line (например, "1531 Androsik Vladimir") преобразуется в поток,  // из которого мы можем извлечь данные (id, surname, name) с помощью оператора >>.  // stringstream автоматически разделяет данные по пробелам или табуляциям,  // что упрощает извлечение отдельных полей.  stringstream ss(line);  int id;  string surname, name;  ss >> id >> surname >> name; // Считываем id, фамилию и имя из строки  Student\* student = new Student(id, surname, name); // Создаём новый объект Student  students.push\_back(student); // Добавляем указатель в вектор  index.insert(id, student); // Добавляем студента в красно-чёрное дерево  count++; // Увеличиваем счётчик  }  file.close();  cout << "Успешно загружено " << count << " записей из файла " << inputFilename << endl;  }  // Метод для поиска студента по IdStudent  void findStudent(int id) {  Student\* student = index.search(id); // Ищем студента в красно-чёрном дереве  if (student)  cout << "Найден: " << student->id << " " << student->surname << " " << student->name << endl;  else  cout << "Студент с ID " << id << " не найден" << endl;  }  // Метод для добавления нового студента  void insertStudent(int id, string surname, string name) {  if (index.search(id)) { // Проверяем, существует ли студент с таким ID  cout << "Студент с ID " << id << " уже существует!" << endl;  return;  }  Student\* student = new Student(id, surname, name); // Создаём нового студента  students.push\_back(student); // Добавляем в вектор  index.insert(id, student); // Добавляем в красно-чёрное дерево  cout << "Студент добавлен: " << id << " " << surname << " " << name << endl;  writeToOutputFile(); // Сохраняем обновлённый список в файл  }  // Метод для удаления студента по IdStudent  void deleteStudent(int id) {  Student\* student = index.search(id); // Ищем студента в дереве  if (!student) {  cout << "Студент с ID " << id << " не найден!" << endl;  return;  }  cout << "Удалён: " << student->id << " " << student->surname << " " << student->name << endl;  index.remove(id); // Удаляем студента из дерева  // Ищем студента с заданным id в векторе students.  // find\_if — это алгоритм из библиотеки <algorithm>, который перебирает элементы вектора students  // в диапазоне [begin, end) и возвращает итератор на первый элемент, для которого лямбда-выражение возвращает true.  auto it = find\_if(students.begin(), students.end(),  [id](Student\* s) { return s->id == id; });  if (it != students.end()) { // Если студент найден в векторе  delete\* it; students.erase(it); // Удаляем указатель из вектора  }  writeToOutputFile(); // Сохраняем обновлённый список в файл  }  };  // Главная функция программы  int main() {  SetConsoleOutputCP(1251);  SetConsoleCP(1251);  // Создаём объект StudentDB  StudentDB db("t01\_Student.txt", "t01\_Student\_updated.txt");  // Тестируем операции с базой данных (файлом)  cout << "1. Поиск существующих студентов:\n";  db.findStudent(1531); // Ищем студента с ID 1531 (Androsik Vladimir)  db.findStudent(1535); // Ищем студента с ID 1535 (Vilbik Ivan)  cout << "\n2. Добавление нового студента:\n";  db.insertStudent(1541, "Makarevich", "Kirill");  cout << "\n3. Поиск нового студента:\n";  db.findStudent(1541); // Ищем добавленного студента (Makarevich Kirill)  cout << "\n4. Удаление студента:\n";  db.deleteStudent(1535); // Удаляем студента с ID 1535 (Vilbik Ivan)  cout << "\n5. Повторный поиск удалённого студента:\n";  db.findStudent(1535); // Проверяем, что студент удалён  cout << "\n6. Поиск несуществующего студента:\n";  db.findStudent(9999); // Ищем студента с несуществующим ID  return 0;  }    **Cхемы БД:**   |  |  |  |  | | --- | --- | --- | --- | | Students | | | | | IdStudent | Surname | Name | | | 1531 | Androsik | Vladimir |  | | 1532 | Bogdanovich | Alexander | | | 1533 | Bogomarov | Rodion | | | 1534 | Bondarik | Evgeny | | | 1535 | Vilbik | Ivan | | | 1536 | Volchkevich | Denis | | | 1537 | Vysotskaya | Yana | | | 1538 | Demyanchik | Ilya | | | 1539 | Dyrda | Dmitry | | | 1540 | Ermakov | Ivan | | |

Схема красно-чёрного дерева:



# **Задачи по теме**

Входные данные приведены лишь в качестве примера — вы можете использовать собственные.  
Для каждой поставленной задачи необходимо самостоятельно разработать и представить **схему базы данных**, даже если для хранения данных используется обычный файл.

|  |  |
| --- | --- |
| **№ варианта** | **Условие задачи** |
| 1 | **Задача**: Используйте красно-чёрное дерево для индексирования книг по их ID. Каждая книга имеет поля: ID, Название, Автор, Год издания. Реализуйте метод, который выводит все книги, изданные после заданного года. **Входные данные**: Файл books.txt  ID Title Author Year  1 Dune Herbert 1965  2 Sherlock Conan Doyle 1887  3 Pride Austen 1813  Найдите книги, изданные после 1900 года. **Ожидаемый результат**:  Книги после 1900 года: ID 1 Dune, Herbert, 1965 |
| 2 | **Задача**: Модифицируйте программу из лабораторной работы, чтобы хранить группу студента (ID, Фамилия, Имя, Группа). Постройте красно-чёрное дерево по ID. Реализуйте метод, который выводит всех студентов из заданной группы. **Входные данные**: Файл students.txt (пример):  ID Surname Name Group  1531 Androsik Vladimir A-101  1532 Bogdanovich Alexander A-101  1533 Bogomarov Rodion B-202  Найдите студентов из группы A-101. **Ожидаемый результат**:  Студенты из группы A-101: 1531 Androsik Vladimir, 1532 Bogdanovich Alexander |
| 3 | **Задача**: Используйте красно-чёрное дерево для индексирования заказов по их ID. Каждый заказ имеет поля: ID, Название товара, Статус ("В обработке", "Доставлен"). Реализуйте метод, который выводит все заказы с заданным статусом.  **Входные данные**: Файл orders.txt:  ID Product Status  101 Laptop V obrabotke  102 Phone Dostavlen  103 Tablet V obrabotke  **Ожидаемый результат**:  Заказы в обработке: ID 101 Laptop, ID 103 Tablet |
| 4 | **Задача**: Постройте красно-чёрное дерево для индексирования фильмов по их ID. Каждый фильм имеет поля: ID, Название, Жанр. Реализуйте метод, который подсчитывает количество фильмов каждого жанра. **Входные данные**: Файл movies.txt:  ID Title Genre  1 Inception Fantastika  2 Matrix Fantastika  3 Titanic Roman  **Ожидаемый результат**:  Фантастика: 2  Роман: 1 |
| 5 | **Задача**: Используйте красно-чёрное дерево для индексирования событий по их ID. Каждое событие имеет поля: ID, Название, Дата (например, "2025-04-15"). Реализуйте метод, который находит все события, запланированные на заданную дату.  **Входные данные**: Файл events.txt:  ID Event Date  1 Meeting 2025-04-15  2 Conference 2025-04-20  3 Workshop 2025-04-15  Найдите события на 2025-04-15.  **Ожидаемый результат**:  События на 2025-04-15: ID 1 Meeting, ID 3 Workshop |
| 6 | Постройте красно-чёрное дерево для индексирования пациентов по их ID. Каждый пациент имеет поля: ID, Фамилия, Диагноз. Реализуйте метод, который выводит всех пациентов с заданным диагнозом.  **Входные данные**: Файл patients.txt:  ID Surname Diagnosis  101 Ivanov Gripp  102 Petrov Angina  103 Sidorov Gripp  Найдите пациентов с диагнозом "Грипп". **Ожидаемый результат**:  Пациенты с диагнозом Грипп: 101 Ivanov, 103 Sidorov |
| 7 | Используйте красно-чёрное дерево для индексирования товаров по их ID. Каждый товар имеет поля: ID, Название, Категория. Реализуйте метод, который группирует товары по категориям.  **Входные данные**: Файл products.txt:  ID Product Category  1 TV Elektronika  2 Phone Elektronika  3 Table Mebel  **Ожидаемый результат**:  Электроника: ID 1 TV, ID 2 Phone  Мебель: ID 3 Table |
| 8 | Постройте красно-чёрное дерево для индексирования задач по их ID. Каждая задача имеет поля: ID, Название, Приоритет (1-5). Реализуйте метод, который выводит задачи в порядке убывания приоритета.  **Входные данные**: Файл tasks.txt:  ID Task Priority  1 Report 3  2 Presentation 5  3 Review 2  **Ожидаемый результат**:  Задачи по приоритету:  ID 2 Presentation (5)  ID 1 Report (3)  ID 3 Review (2) |
| 9 | Используйте красно-чёрное дерево для индексирования маршрутов по их ID. Каждый маршрут имеет поля: ID, Название, Тип транспорта ("Автобус", "Поезд"). Реализуйте метод, который выводит маршруты для заданного типа транспорта.  **Входные данные**: Файл routes.txt:  ID Route Transport  1 RouteA Avtobus  2 RouteB Poezd  3 RouteC Avtobus  Найдите маршруты на автобусе.  **Ожидаемый результат:**  Маршруты на автобусе: ID 1 RouteA, ID 3 RouteC |
| 10 | Постройте красно-чёрное дерево для индексирования животных по их ID. Каждое животное имеет поля: ID, Имя, Вес (в кг). Реализуйте метод, который находит животных с весом в заданном диапазоне.  **Входные данные**: Файл animals.txt:  ID Name Weight  1 Simba 150  2 Dumbo 3000  3 Leo 180  Найдите животных с весом от 100 до 200 кг. **Ожидаемый результат**: Выведите:  Животные с весом 100-200 кг: ID 1 Simba, ID 3 Leo |
| 11 | Используйте красно-чёрное дерево для индексирования билетов на концерт по их ID. Каждый билет имеет поля: ID, Покупатель, Цена. Реализуйте метод, который находит билеты в заданном ценовом диапазоне.  **Входные данные**: Файл tickets.txt:  ID Buyer Price  1 Ivanov 50  2 Petrov 100  3 Sidorov 75  Найдите билеты с ценой от 60 до 80. **Ожидаемый результат**: Выведите:  Билеты с ценой 60-80: ID 3 Sidorov |
| 12 | Постройте красно-чёрное дерево для индексирования рецептов по их ID. Каждый рецепт имеет поля: ID, Название, Время приготовления (в минутах). Реализуйте метод, который находит рецепты, которые можно приготовить за заданное время или меньше.  **Входные данные**: Файл recipes.txt:  ID Recipe Time  1 Pizza 30  2 Salad 15  3 Pasta 20  Найдите рецепты, которые готовятся не дольше 20 минут. **Ожидаемый результат**: Выведите:  Рецепты до 20 минут: ID 2 Salad, ID 3 Pasta |
| 13 | Используйте красно-чёрное дерево для индексирования видеоигр по их ID. Каждая игра имеет поля: ID, Название, Год выпуска. Реализуйте метод, который подсчитывает количество игр, выпущенных в каждом году.  **Входные данные**: Файл games.txt:  ID Game Year  1 Cyberpunk 2020  2 FIFA 2021  3 Minecraft 2011  **Ожидаемый результат**: Выведите:  2020: 1  2021: 1  2011: 1 |
| 14 | Постройте красно-чёрное дерево для индексирования сотрудников по их ID. Каждый сотрудник имеет поля: ID, Фамилия, Отдел. Реализуйте метод, который группирует сотрудников по отделам.  **Входные данные**: Файл employees.txt:  ID Surname Department  101 Ivanov IT  102 Petrov HR  103 Sidorov IT  **Ожидаемый результат**: Выведите:  IT: 101 Ivanov, 103 Sidorov  HR: 102 Petrov |
| 15 | Используйте красно-чёрное дерево для индексирования путешествий по их ID. Каждое путешествие имеет поля: ID, Направление, Длительность (в днях). Реализуйте метод, который находит путешествия с заданной длительностью.  **Входные данные**: Файл trips.txt:  ID Destination Duration  1 Paris 5  2 Tokyo 10  3 London 5  Найдите путешествия длительностью 5 дней. **Ожидаемый результат**: Выведите:  Путешествия длительностью 5 дней: ID 1 Paris, ID 3 London |

# **Дополнительные задачи**

## **6.1 Анализатор временных рядов датчиков**

Реализуйте систему индексирования временных рядов с использованием красно-чёрного дерева. Каждый временной ряд принадлежит датчику (ID, Название датчика) и содержит временные метки с измерениями (Timestamp, Value). Постройте красно-чёрное дерево по ID датчика. Реализуйте метод, который для заданного датчика прогнозирует следующее значение, используя линейную регрессию (на основе последних N измерений). Также реализуйте метод для поиска аномалий (значений, которые отклоняются от среднего более чем на 2 стандартных отклонения).

**Входные данные**  
Файл sensors.txt:

|  |  |
| --- | --- |
| ID | SensorName |
| 101 | TempSensor |
| 102 | PressureSensor |

Файл measurements.txt:

|  |  |  |
| --- | --- | --- |
| SensorID | Timestamp | Value |
| 101 | 2025-04-12 10:00 | 20.5 |
| 101 | 2025-04-12 10:05 | 21.0 |
| 101 | 2025-04-12 10:10 | 22.0 |
| 101 | 2025-04-12 10:15 | 30.0 |
| 102 | 2025-04-12 10:00 | 1013.0 |
| 102 | 2025-04-12 10:05 | 1012.5 |
| 102 | 2025-04-12 10:10 | 1014.0 |

Прогнозируйте следующее значение для датчика с ID 101 (используя последние 3 измерения) и найдите аномалии для этого датчика.

**Ожидаемый результат**

Прогноз для датчика 101: 23.0

Аномалии для датчика 101:

2025-04-12 10:15: 30.0 (отклонение от среднего: 7.5)

**Схема базы данных**  
Добавлена таблица Measurements для хранения измерений и таблица Predictions для хранения прогнозов.

## **6.2 Умный подбор товаров с ограничением по весу**

Реализуйте систему индексирования товаров с использованием красно-чёрного дерева. Каждый товар имеет поля: ID, Название, Цена, Вес, Категория. Постройте красно-чёрное дерево по ID товара. Реализуйте метод, который выполняет многомерный запрос: находит товары в заданном диапазоне цен, веса и из заданной категории. Добавьте метод, который оптимизирует выбор товаров для доставки (максимизирует суммарную стоимость при ограничении на вес).

**Входные данные**  
Файл products.txt:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | ProductName | Price | Weight | Category |
| 301 | ProductA | 100 | 2 | Electronics |
| 302 | ProductB | 150 | 3 | Electronics |
| 303 | ProductC | 80 | 1 | Clothing |

Найдите товары с ценой 90-160, весом 1-3, из категории Electronics. Оптимизируйте выбор товаров для доставки с максимальным весом 4.

**Ожидаемый результат**

Товары (цена 90-160, вес 1-3, категория Electronics):

ID 301 ProductA, Price: 100, Weight: 2

ID 302 ProductB, Price: 150, Weight: 3

Оптимальный выбор для доставки (макс. вес 4):

ID 302 ProductB, Price: 150, Weight: 3

ID 303 ProductC, Price: 80, Weight: 1

Общая стоимость: 230

## **6.3 Система мониторинга движущихся объектов с прогнозированием столкновений**

Реализуйте систему индексирования движущихся объектов с использованием красно-чёрного дерева. Каждый объект имеет поля: ID, Название, Траектория (временные метки с координатами: Timestamp, Latitude, Longitude). Постройте красно-чёрное дерево по ID объекта. Реализуйте метод, который прогнозирует будущую позицию объекта через заданное время (линейная интерполяция). Реализуйте метод, который находит потенциальные столкновения (объекты, которые окажутся ближе 1 км друг к другу через заданное время).

**Входные данные**  
Файл objects.txt:

|  |  |
| --- | --- |
| ID | ObjectName |
| 401 | ObjectA |
| 402 | ObjectB |

Файл trajectories.txt:

|  |  |  |  |
| --- | --- | --- | --- |
| ObjectID | Timestamp | Latitude | Longitude |
| 401 | 2025-04-12 10:00 | 52.0 | 13.0 |
| 401 | 2025-04-12 10:05 | 52.1 | 13.1 |
| 402 | 2025-04-12 10:00 | 52.2 | 13.2 |
| 402 | 2025-04-12 10:05 | 52.1 | 13.1 |

Прогнозируйте позицию объектов через 5 минут (на 2025-04-12 10:10). Найдите потенциальные столкновения.

**Ожидаемый результат**

Прогноз на 2025-04-12 10:10:

ObjectA: (52.2, 13.2)

ObjectB: (52.0, 13.0)

Потенциальные столкновения:

ObjectA и ObjectB (расстояние: 0 км)

## **6.4 Многопользовательская система управления транзакциями с MVCC**

Реализуйте систему индексирования транзакций с использованием красно-чёрного дерева. Каждая транзакция имеет поля: ID, Сумма, Статус (Active, Committed, Aborted), Время. Постройте красно-чёрное дерево по ID транзакции. Реализуйте систему MVCC (многоверсионное управление конкурентным доступом): каждая операция (вставка, обновление) создаёт новую версию записи. Реализуйте метод, который обеспечивает согласованность (проверяет, что сумма всех активных транзакций не превышает заданный лимит). Реализуйте метод отката до заданного времени.

**Входные данные**  
Файл transactions.txt:

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Amount | Status | Timestamp |
| 501 | 100 | Committed | 2025-04-12 10:00 |
| 502 | 200 | Committed | 2025-04-12 10:05 |

Добавьте транзакции: (503, 300, Active, 2025-04-12 10:10), (504, 400, Active, 2025-04-12 10:15). Проверьте согласованность (лимит 600). Откатите до 2025-04-12 10:05.

**Ожидаемый результат**

Сумма активных транзакций: 700

Нарушение лимита (600)!

После отката до 2025-04-12 10:05:

ID 501 Amount 100

ID 502 Amount 200

## **6.5 Интеллектуальный текстовый индекс**

Реализуйте систему индексирования слов из текстового корпуса с использованием красно-чёрного дерева. Каждое слово имеет поля: само слово (уникальный ключ), частота встречаемости, список документов (ID документа, где слово встречается), список связанных слов (слова, которые встречаются в одном предложении с данным словом). Постройте красно-чёрное дерево по строковому ключу (слову). Реализуйте следующие функции:

1. **Анализ связей слов**: Найдите компоненты связности в графе слов, где рёбра соединяют слова, которые встречаются в одном предложении.
2. **Рекомендация слов**: Для заданного слова найдите слова, которые часто встречаются с ним в одном предложении, и отсортируйте их по частоте совместного появления.
3. **Анализ редких слов**: Найдите слова, которые встречаются реже заданного порога (например, частота < 3), но имеют высокую степень связности (много связанных слов).
4. **Оптимизация корпуса**: Удалите из корпуса слова, которые встречаются только в одном документе и имеют низкую частоту (например, < 2), и обновите связи между оставшимися словами.

**Входные данные**  
Файл words.txt:

|  |  |  |
| --- | --- | --- |
| Word | Frequency | DocIDs |
| apple | 5 | 1,2,3 |
| banana | 3 | 1,2 |
| cherry | 2 | 1 |
| date | 1 | 3 |

Файл cooccurrences.txt:

|  |  |
| --- | --- |
| Word1 | Word2 |
| apple | banana |
| apple | cherry |
| banana | cherry |

**Действия**:

1. Найдите компоненты связности в графе слов.
2. Для слова "apple" найдите рекомендации (слова, с которыми оно часто встречается).
3. Найдите редкие слова с высокой степенью связности (порог частоты: 3).
4. Оптимизируйте корпус, удалив слова с частотой < 2 и встречающиеся только в одном документе, и обновите связи.

**Ожидаемый результат**

Компоненты связности:

Компонента 1: apple, banana, cherry

Компонента 2: date

Рекомендации для слова "apple":

banana (совместно: 1), cherry (совместно: 1)

Редкие слова с высокой степенью связности (порог 3):

cherry (частота: 2, связанных слов: 2)

После оптимизации корпуса:

Удалено слово: date

Оставшиеся слова: apple, banana, cherry

Обновлённые связи:

apple: banana, cherry

banana: apple, cherry

cherry: apple, banana

# **Контрольные вопросы**

**1. Что такое красно-чёрное дерево и почему оно называется именно так?**

**2. Перечислите основные правила красно-чёрного дерева.**

**3. Почему красно-чёрное дерево подходит для индексирования в базах данных?**

**4. Какой цвет имеет новый узел при вставке в красно-чёрное дерево и почему?**

**5. Опишите случай балансировки при вставке, когда у нового узла красный родитель и красный дядя.**

**6. Какой случай балансировки при удалении возникает, если брат удаляемого узла красный?**

**7. Что такое чёрная высота и почему она важна для красно-чёрного дерева?**

**8. Как красно-чёрное дерево обеспечивает логарифмическую сложность операций?**

**9. В чём преимущество красно-чёрного дерева перед обычным бинарным деревом поиска (BST) при использовании в базах данных?**

**10. Какие ограничения красно-чёрного дерева для индексирования в базах данных?**

**11. Какой метод используется в программе для поиска студента по ID? Опишите его работу.**

**12. Как в программе реализовано удаление студента из базы данных?**

**13. Почему в программе используется вектор students в дополнение к красно-чёрному дереву?**

**14. Как красно-чёрное дерево может быть использовано для диапазонных запросов?**

**15. Сравните красно-чёрное дерево с B-деревом для задач индексирования.**

# **8. Решения к задачам повышенной сложности**

|  |  |
| --- | --- |
| **№** | **Условие задачи** |
| 1 | Реализуйте систему индексирования временных рядов с использованием красно-чёрного дерева. Каждый временной ряд принадлежит датчику (ID, Название датчика) и содержит временные метки с измерениями (Timestamp, Value). Постройте красно-чёрное дерево по ID датчика. Реализуйте метод, который для заданного датчика прогнозирует следующее значение, используя линейную регрессию (на основе последних N измерений). Также реализуйте метод для поиска аномалий (значений, которые отклоняются от среднего более чем на 2 стандартных отклонения).  **Входные данные** Файл devices.txt:  ID SensorName  101 TempSensor  102 PressureSensor  Файл measurements.txt:  SensorID Timestamp Value  101 2025-04-12 10:00 20.5  101 2025-04-12 10:05 21.0  101 2025-04-12 10:10 22.0  101 2025-04-12 10:15 30.0  102 2025-04-12 10:00 1013.0  102 2025-04-12 10:05 1012.5  102 2025-04-12 10:10 1014.0  Прогнозируйте следующее значение для датчика с ID 101 (используя последние 3 измерения) и найдите аномалии для этого датчика.  **Ожидаемый результат**  Прогноз для устройства 101: 2025  Тренд устройства 101: Стабильность  Устройства с аномальным трендом:    **Схема базы данных** Добавлена таблица Measurements для хранения измерений и таблица Predictions для хранения прогнозов.  #include <iostream>  #include <fstream>  #include <string>  #include <sstream>  #include <vector>  #include <cmath>  using namespace *std*;  // Структура для хранения одного измерения  struct Measurement {  *string* timestamp; // Временная метка измерения  double value; // Значение измерения  };  // Структура для устройства с датчиками  struct Device {  int id; // Уникальный идентификатор устройства  *string* name; // Название устройства  *vector*<Measurement> measurements; // Вектор всех измерений устройства  // Конструктор устройства  Device(int \_id, *string* \_name) {  id = \_id;  name = \_name;  }  };  // Перечисление для цветов узлов красно-черного дерева  enum Color { RED, BLACK };  // Структура узла красно-черного дерева  struct RBNode {  int key; // Ключ узла (id устройства)  Device\* data; // Указатель на данные устройства  RBNode\* left; // Указатель на левого потомка  RBNode\* right; // Указатель на правого потомка  RBNode\* parent; // Указатель на родителя  Color color; // Цвет узла  // Конструктор узла  RBNode(int k, Device\* d) {  key = k;  data = d;  left = nullptr;  right = nullptr;  parent = nullptr;  color = RED; // Новые узлы всегда красные  }  };  // Структура красно-черного дерева  struct RBTree {  RBNode\* root; // Корень дерева  RBNode\* nil; // Специальный nil-узел (листья дерева)  // Конструктор дерева  RBTree() {  nil = new RBNode(0, nullptr);  nil->color = BLACK; // Nil-узлы всегда черные  nil->*left* = nil;  nil->*right* = nil;  root = nil;  }  // Левый поворот вокруг узла x  void leftRotate(RBNode\* x) {  RBNode\* y = x->*right*;  x->*right* = y->*left*;  if (y->*left* != nil) y->*left*->parent = x;  y->parent = x->parent;  if (x->parent == nullptr) root = y;  else if (x == x->parent->*left*) x->parent->*left* = y;  else x->parent->*right* = y;  y->*left* = x;  x->parent = y;  }  // Правый поворот вокруг узла y  void rightRotate(RBNode\* y) {  RBNode\* x = y->*left*;  y->*left* = x->*right*;  if (x->*right* != nil) x->*right*->parent = y;  x->parent = y->parent;  if (y->parent == nullptr) root = x;  else if (y == y->parent->*right*) y->parent->*right* = x;  else y->parent->*left* = x;  x->*right* = y;  y->parent = x;  }  // Восстановление свойств дерева после вставки  void fixInsert(RBNode\* z) {  while (z->parent != nullptr && z->parent->color == RED) {  if (z->parent == z->parent->parent->*left*) {  RBNode\* y = z->parent->parent->*right*;  if (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: дядя черный, z - правый потомок  z = z->parent;  leftRotate(z);  }  // Случай 3: дядя черный, z - левый потомок  z->parent->color = BLACK;  z->parent->parent->color = RED;  rightRotate(z->parent->parent);  }  }  else {  // Симметричные случаи для правого поддерева  RBNode\* y = z->parent->parent->*left*;  if (y->color == RED) {  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*left*) {  z = z->parent;  rightRotate(z);  }  z->parent->color = BLACK;  z->parent->parent->color = RED;  leftRotate(z->parent->parent);  }  }  }  root->color = BLACK; // Корень всегда черный  }  // Вставка нового узла в дерево  void insert(int key, Device\* data) {  RBNode\* z = new RBNode(key, data);  z->*left* = nil;  z->*right* = nil;  RBNode\* y = nullptr;  RBNode\* x = root;  // Поиск места для вставки  while (x != nil) {  y = x;  if (z->key < x->key) x = x->*left*;  else x = x->*right*;  }  // Установка родителя  z->parent = y;  if (y == nullptr) root = z;  else if (z->key < y->key) y->*left* = z;  else y->*right* = z;  // Балансировка дерева  fixInsert(z);  }  // Поиск устройства по id  Device\* search(int key) {  RBNode\* current = root;  while (current != nil) {  if (key == current->key) return current->*data*;  else if (key < current->key) current = current->*left*;  else current = current->*right*;  }  return nullptr; // Устройство не найдено  }  };  // Класс для работы с базой данных устройств  struct DeviceDB {  RBTree index; // Красно-черное дерево для быстрого поиска  *vector*<Device\*> devices; // Все устройства  *string* devicesFile; // Файл с данными устройств  *string* measurementsFile; // Файл с измерениями  // Конструктор  DeviceDB(*string* dFile, *string* mFile) {  devicesFile = dFile;  measurementsFile = mFile;  loadDevices(); // Загрузка устройств при создании  loadMeasurements(); // Загрузка измерений при создании  }  // Загрузка устройств из файла  void loadDevices() {  *ifstream* file(devicesFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << devicesFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int id;  *string* name;  ss >> id >> name;  // Создание нового устройства и добавление в базу  Device\* device = new Device(id, name);  devices.*push\_back*(device);  index.insert(id, device); // Вставка в индексное дерево  }  file.*close*();  }  // Загрузка измерений из файла  void loadMeasurements() {  *ifstream* file(measurementsFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << measurementsFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int deviceID;  *string* timestamp;  double value;  ss >> deviceID;  ss.*ignore*(1);  *getline*(ss, timestamp, ' ');  ss >> value;  // Поиск устройства и добавление измерения  Device\* device = index.search(deviceID);  if (device) {  Measurement m;  m.timestamp = timestamp;  m.value = value;  device->measurements.*push\_back*(m);  }  }  file.*close*();  }  // Прогнозирование следующего значения с помощью экспоненциального сглаживания  double predictNextValue(int deviceID) {  Device\* device = index.search(deviceID);  if (!device || device->measurements.*size*() < 2) return -1;  double alpha = 0.3; // Коэффициент сглаживания  double forecast = device->measurements[0].value;  // Применение формулы экспоненциального сглаживания  for (*size\_t* i = 1; i < device->measurements.*size*(); i++) {  forecast = alpha \* device->measurements[i].value + (1 - alpha) \* forecast;  }  return forecast;  }  // Определение тренда на основе скользящего среднего  *string* determineTrend(Device\* device) {  if (device->measurements.*size*() < 3) return "Недостаточно данных";  *vector*<double> movingAverages;  // Расчет скользящего среднего по 3 точкам  for (*size\_t* i = 2; i < device->measurements.*size*(); i++) {  double avg = (device->measurements[i].value +  device->measurements[i - 1].value +  device->measurements[i - 2].value) / 3.0;  movingAverages.*push\_back*(avg);  }  if (movingAverages.*size*() < 2) return "Недостаточно данных";  // Анализ изменения скользящего среднего  double diff = movingAverages.*back*() - movingAverages[movingAverages.*size*() - 2];  if (diff > 0.5) return "Рост";  else if (diff < -0.5) return "Спад";  else return "Стабильность";  }  // Поиск аномальных изменений тренда  void findAnomalousTrends() {  *cout* << "Устройства с аномальным трендом:\n";  for (Device\* device : devices) {  if (device->measurements.*size*() < 4) continue;  *vector*<*string*> trends;  // Анализ тренда на разных временных отрезках  for (*size\_t* i = 3; i < device->measurements.*size*(); i++) {  Device tempDevice(device->id, device->name);  for (*size\_t* j = 0; j <= i; j++) {  tempDevice.measurements.*push\_back*(device->measurements[j]);  }  trends.*push\_back*(determineTrend(&tempDevice));  }  // Выявление резкого роста после стабильности  if (trends.*size*() >= 2 &&  trends[trends.*size*() - 2] == "Стабильность" &&  trends.*back*() == "Рост") {  *cout* << device->name << " (резкий рост после стабильности)\n";  }  }  }  };  int *main*() {  // Создание и инициализация базы данных  DeviceDB db("devices.txt", "measurements.txt");  // Пример использования функционала  double forecast = db.predictNextValue(101);  *cout* << "Прогноз для устройства 101: " << forecast << "\n";  Device\* device = db.index.search(101);  *cout* << "Тренд устройства 101: " << db.determineTrend(device) << "\n";  // Поиск аномальных трендов  db.findAnomalousTrends();  return 0;  } |
| 2 | Реализуйте систему индексирования товаров с использованием красно-чёрного дерева. Каждый товар имеет поля: ID, Название, Цена, Вес, Категория. Постройте красно-чёрное дерево по ID товара. Реализуйте метод, который выполняет многомерный запрос: находит товары в заданном диапазоне цен, веса и из заданной категории. Добавьте метод, который оптимизирует выбор товаров для доставки (максимизирует суммарную стоимость при ограничении на вес).  **Входные данные** Файл products.txt:  ID ProductName Price Weight Category  301 ProductA 100 2 Electronics  302 ProductB 150 3 Electronics  303 ProductC 80 1 Clothing  Найдите товары с ценой 90-160, весом 1-3, из категории Electronics. Оптимизируйте выбор товаров для доставки с максимальным весом 4.  **Ожидаемый результат**  Товары (цена 90-160, вес 1-3, категория Electronics):  ID 301 ProductA, Price: 100, Weight: 2  ID 302 ProductB, Price: 150, Weight: 3  Оптимальный выбор для доставки (макс. вес 4):  ID 303 ProductC, Price: 80, Weight: 1  ID 301 ProductA, Price: 100, Weight: 2  Общая стоимость: 180    #include <iostream>  #include <fstream>  #include <string>  #include <sstream>  #include <vector>  #include <algorithm>  using namespace *std*;  // Структура для хранения информации о товаре  struct Product {  int id; // Уникальный идентификатор товара  *string* name; // Название товара  int price; // Цена товара  int weight; // Вес товара  *string* category; // Категория товара  // Конструктор товара  Product(int \_id, *string* \_name, int \_price, int \_weight, *string* \_cat) {  id = \_id;  name = \_name;  price = \_price;  weight = \_weight;  category = \_cat;  }  };  // Перечисление цветов для красно-черного дерева  enum Color { RED, BLACK };  // Структура узла красно-черного дерева  struct RBNode {  int key; // Ключ (id товара)  Product\* data; // Указатель на данные товара  RBNode\* left; // Левый потомок  RBNode\* right; // Правый потомок  RBNode\* parent; // Родительский узел  Color color; // Цвет узла  // Конструктор узла  RBNode(int k, Product\* d) {  key = k;  data = d;  left = nullptr;  right = nullptr;  parent = nullptr;  color = RED; // Новые узлы всегда красные  }  };  // Структура красно-черного дерева  struct RBTree {  RBNode\* root; // Корень дерева  RBNode\* nil; // Специальный nil-узел (листья дерева)  // Конструктор дерева  RBTree() {  nil = new RBNode(0, nullptr);  nil->color = BLACK; // Nil-узлы всегда черные  nil->*left* = nil;  nil->*right* = nil;  root = nil;  }  // Левый поворот вокруг узла x  void leftRotate(RBNode\* x) {  RBNode\* y = x->*right*; // y становится правым потомком x  x->*right* = y->*left*; // Левый потомок y становится правым потомком x  if (y->*left* != nil)  y->*left*->parent = x;  y->parent = x->parent; // Перенос родителя  if (x->parent == nullptr)  root = y;  else if (x == x->parent->*left*)  x->parent->*left* = y;  else  x->parent->*right* = y;  y->*left* = x; // x становится левым потомком y  x->parent = y;  }  // Правый поворот вокруг узла y  void rightRotate(RBNode\* y) {  RBNode\* x = y->*left*; // x становится левым потомком y  y->*left* = x->*right*; // Правый потомок x становится левым потомком y  if (x->*right* != nil)  x->*right*->parent = y;  x->parent = y->parent; // Перенос родителя  if (y->parent == nullptr)  root = x;  else if (y == y->parent->*right*)  y->parent->*right* = x;  else  y->parent->*left* = x;  x->*right* = y; // y становится правым потомком x  y->parent = x;  }  // Восстановление свойств дерева после вставки  void fixInsert(RBNode\* z) {  // Пока родитель красный (нарушение свойства красно-черного дерева)  while (z->parent != nullptr && z->parent->color == RED) {  if (z->parent == z->parent->parent->*left*) {  RBNode\* y = z->parent->parent->*right*; // Дядя узла z  if (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: дядя черный, z - правый потомок  z = z->parent;  leftRotate(z);  }  // Случай 3: дядя черный, z - левый потомок  z->parent->color = BLACK;  z->parent->parent->color = RED;  rightRotate(z->parent->parent);  }  }  else {  // Симметричные случаи для правого поддерева  RBNode\* y = z->parent->parent->*left*;  if (y->color == RED) {  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*left*) {  z = z->parent;  rightRotate(z);  }  z->parent->color = BLACK;  z->parent->parent->color = RED;  leftRotate(z->parent->parent);  }  }  }  root->color = BLACK; // Корень всегда черный  }  // Вставка нового узла в дерево  void insert(int key, Product\* data) {  RBNode\* z = new RBNode(key, data);  z->*left* = nil;  z->*right* = nil;  RBNode\* y = nullptr;  RBNode\* x = root;  // Поиск места для вставки  while (x != nil) {  y = x;  if (z->key < x->key)  x = x->*left*;  else  x = x->*right*;  }  // Установка родителя  z->parent = y;  if (y == nullptr)  root = z;  else if (z->key < y->key)  y->*left* = z;  else  y->*right* = z;  // Балансировка дерева  fixInsert(z);  }  // Рекурсивный поиск товаров в заданном диапазоне  void findInRange(RBNode\* node, int minPrice, int maxPrice,  int minWeight, int maxWeight, *string* category) {  if (node == nil) return;  // Сначала идем в левое поддерево (обход в порядке возрастания)  findInRange(node->*left*, minPrice, maxPrice, minWeight, maxWeight, category);  Product\* product = node->*data*;  // Проверка соответствия критериям поиска  if (product->price >= minPrice && product->price <= maxPrice &&  product->weight >= minWeight && product->weight <= maxWeight &&  product->category == category) {  // Вывод информации о найденном товаре  *cout* << "ID " << product->id << " " << product->name  << ", Price: " << product->price  << ", Weight: " << product->weight << "\n";  }  // Затем идем в правое поддерево  findInRange(node->*right*, minPrice, maxPrice, minWeight, maxWeight, category);  }  };  // Класс для работы с базой данных товаров  struct ProductDB {  RBTree index; // Красно-черное дерево для индексации  *vector*<Product\*> products; // Все товары  *string* inputFile; // Имя входного файла  // Конструктор базы данных  ProductDB(*string* file) {  inputFile = file;  loadFromFile(); // Загрузка данных при создании  }  // Загрузка данных из файла  void loadFromFile() {  *ifstream* file(inputFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << inputFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int id, price, weight;  *string* name, category;  // Чтение данных о товаре из строки  ss >> id >> name >> price >> weight >> category;  // Создание нового товара и добавление в базу  Product\* product = new Product(id, name, price, weight, category);  products.*push\_back*(product);  index.insert(id, product); // Вставка в индексное дерево  }  file.*close*();  }  // Поиск товаров в заданном диапазоне параметров  void findInRange(int minPrice, int maxPrice,  int minWeight, int maxWeight, *string* category) {  *cout* << "Товары (цена " << minPrice << "-" << maxPrice  << ", вес " << minWeight << "-" << maxWeight  << ", категория " << category << "):\n";  index.findInRange(index.root, minPrice, maxPrice, minWeight, maxWeight, category);  }  // Оптимизация доставки (жадный алгоритм)  void optimizeDelivery(int maxWeight) {  // Сортировка товаров по убыванию отношения цена/вес  *vector*<Product\*> sortedProducts = products;  *sort*(sortedProducts.*begin*(), sortedProducts.*end*(), [](Product\* a, Product\* b) {  return (double)a->price / a->weight > (double)b->price / b->weight;  });  *vector*<Product\*> selected; // Выбранные товары  int totalWeight = 0; // Общий вес  int totalPrice = 0; // Общая стоимость  // Выбор товаров с максимальным отношением цена/вес  for (Product\* product : sortedProducts) {  if (totalWeight + product->weight <= maxWeight) {  selected.*push\_back*(product);  totalWeight += product->weight;  totalPrice += product->price;  }  }  // Вывод результатов оптимизации  *cout* << "\nОптимальный выбор для доставки (макс. вес " << maxWeight << "):\n";  for (Product\* product : selected) {  *cout* << "ID " << product->id << " " << product->name  << ", Price: " << product->price  << ", Weight: " << product->weight << "\n";  }  *cout* << "Общая стоимость: " << totalPrice << "\n";  }  };  int *main*() {  *setlocale*(*LC\_ALL*, "ru"); // Настройка локали для вывода русских символов  // Создание и инициализация базы данных  ProductDB db("products.txt");  // Пример 1: Поиск товаров в заданном диапазоне  db.findInRange(90, 160, 1, 3, "Electronics");  // Пример 2: Оптимизация доставки  db.optimizeDelivery(4);  return 0;  } |
| 3 | Реализуйте систему индексирования движущихся объектов с использованием красно-чёрного дерева. Каждый объект имеет поля: ID, Название, Траектория (временные метки с координатами: Timestamp, Latitude, Longitude). Постройте красно-чёрное дерево по ID объекта. Реализуйте метод, который прогнозирует будущую позицию объекта через заданное время (линейная интерполяция). Реализуйте метод, который находит потенциальные столкновения (объекты, которые окажутся ближе 1 км друг к другу через заданное время).  **Входные данные** Файл objects.txt:  ID ObjectName  401 ObjectA  402 ObjectB  Файл trajectories.txt:  ObjectID Timestamp Latitude Longitude  401 2025-04-12 10:00 52.0 13.0  401 2025-04-12 10:05 52.1 13.1  402 2025-04-12 10:00 52.2 13.2  402 2025-04-12 10:05 52.1 13.1  Прогнозируйте позицию объектов через 5 минут (на 2025-04-12 10:10). Найдите потенциальные столкновения.  **Ожидаемый результат**  Прогноз на 2025-04-12 10:10:  ObjectA: (52.2, 13.2)  ObjectB: (52.0, 13.0)  Потенциальные столкновения:  ObjectA и ObjectB (расстояние: 0 км)     |  | | --- | | #include <iostream>  #include <fstream>  #include <string>  #include <sstream>  #include <vector>  #include <cmath>  using namespace *std*;  // Структура для хранения точки траектории с временной меткой  struct Point {  *string* timestamp; // Временная метка (формат ГГГГ-ММ-ДД ЧЧ:ММ:СС)  double latitude; // Широта в градусах  double longitude; // Долгота в градусах  };  // Структура для объекта с траекторией движения  struct Object {  int id; // Уникальный идентификатор объекта  *string* name; // Название объекта  *vector*<Point> trajectory; // Вектор точек траектории  // Конструктор объекта  Object(int \_id, *string* \_name) {  id = \_id;  name = \_name;  }  };  // Цвета узлов для красно-черного дерева  enum Color { RED, BLACK };  // Узел красно-черного дерева  struct RBNode {  int key; // Ключ (id объекта)  Object\* data; // Указатель на данные объекта  RBNode\* left; // Левый потомок  RBNode\* right; // Правый потомок  RBNode\* parent; // Родительский узел  Color color; // Цвет узла  // Конструктор узла  RBNode(int k, Object\* d) {  key = k;  data = d;  left = nullptr;  right = nullptr;  parent = nullptr;  color = RED; // Новые узлы всегда красные  }  };  // Красно-черное дерево для хранения и быстрого поиска объектов  struct RBTree {  RBNode\* root; // Корень дерева  RBNode\* nil; // Специальный nil-узел (листья дерева)  // Конструктор дерева  RBTree() {  nil = new RBNode(0, nullptr);  nil->color = BLACK; // Nil-узлы всегда черные  nil->*left* = nil;  nil->*right* = nil;  root = nil;  }  // Левый поворот вокруг узла x  void leftRotate(RBNode\* x) {  RBNode\* y = x->*right*; // y становится правым потомком x  x->*right* = y->*left*; // Левый потомок y становится правым потомком x  if (y->*left* != nil)  y->*left*->parent = x;  y->parent = x->parent; // Перенос родителя  if (x->parent == nullptr)  root = y;  else if (x == x->parent->*left*)  x->parent->*left* = y;  else  x->parent->*right* = y;  y->*left* = x; // x становится левым потомком y  x->parent = y;  }  // Правый поворот вокруг узла y  void rightRotate(RBNode\* y) {  RBNode\* x = y->*left*; // x становится левым потомком y  y->*left* = x->*right*; // Правый потомок x становится левым потомком y  if (x->*right* != nil)  x->*right*->parent = y;  x->parent = y->parent; // Перенос родителя  if (y->parent == nullptr)  root = x;  else if (y == y->parent->*right*)  y->parent->*right* = x;  else  y->parent->*left* = x;  x->*right* = y; // y становится правым потомком x  y->parent = x;  }  // Восстановление свойств дерева после вставки  void fixInsert(RBNode\* z) {  // Пока родитель красный (нарушение свойства красно-черного дерева)  while (z->parent != nullptr && z->parent->color == RED) {  if (z->parent == z->parent->parent->*left*) {  RBNode\* y = z->parent->parent->*right*; // Дядя узла z  if (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: дядя черный, z - правый потомок  z = z->parent;  leftRotate(z);  }  // Случай 3: дядя черный, z - левый потомок  z->parent->color = BLACK;  z->parent->parent->color = RED;  rightRotate(z->parent->parent);  }  }  else {  // Симметричные случаи для правого поддерева  RBNode\* y = z->parent->parent->*left*;  if (y->color == RED) {  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*left*) {  z = z->parent;  rightRotate(z);  }  z->parent->color = BLACK;  z->parent->parent->color = RED;  leftRotate(z->parent->parent);  }  }  }  root->color = BLACK; // Корень всегда черный  }  // Вставка нового узла в дерево  void insert(int key, Object\* data) {  RBNode\* z = new RBNode(key, data);  z->*left* = nil;  z->*right* = nil;  RBNode\* y = nullptr;  RBNode\* x = root;  // Поиск места для вставки  while (x != nil) {  y = x;  if (z->key < x->key)  x = x->*left*;  else  x = x->*right*;  }  // Установка родителя  z->parent = y;  if (y == nullptr)  root = z;  else if (z->key < y->key)  y->*left* = z;  else  y->*right* = z;  // Балансировка дерева  fixInsert(z);  }  // Поиск объекта по id  Object\* search(int key) {  RBNode\* current = root;  while (current != nil) {  if (key == current->key)  return current->*data*;  else if (key < current->key)  current = current->*left*;  else  current = current->*right*;  }  return nullptr; // Объект не найден  }  };  // Класс для работы с базой данных объектов  struct ObjectDB {  RBTree index; // Красно-черное дерево для быстрого поиска  *vector*<Object\*> objects; // Все объекты  *string* objectsFile; // Файл с данными объектов  *string* trajectoriesFile; // Файл с траекториями  // Конструктор базы данных  ObjectDB(*string* oFile, *string* tFile) {  objectsFile = oFile;  trajectoriesFile = tFile;  loadObjects(); // Загрузка объектов при создании  loadTrajectories(); // Загрузка траекторий при создании  }  // Загрузка объектов из файла  void loadObjects() {  *ifstream* file(objectsFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << objectsFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int id;  *string* name;  // Чтение данных об объекте из строки  ss >> id >> name;  // Создание нового объекта и добавление в базу  Object\* object = new Object(id, name);  objects.*push\_back*(object);  index.insert(id, object); // Вставка в индексное дерево  }  file.*close*();  }  // Загрузка траекторий из файла  void loadTrajectories() {  *ifstream* file(trajectoriesFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << trajectoriesFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int objectID;  *string* timestamp;  double lat, lon;  ss >> objectID;  ss.*ignore*(1);  *getline*(ss, timestamp, ' ');  ss >> lat >> lon;  // Поиск объекта и добавление точки траектории  Object\* object = index.*search*(objectID);  if (object) {  Point p;  p.timestamp = timestamp;  p.latitude = lat;  p.longitude = lon;  object->trajectory.*push\_back*(p);  }  }  file.*close*();  }  // Прогнозирование позиции объекта через указанное количество минут  *pair*<double, double> predictPosition(Object\* object, int minutesAhead) {  // Необходимо минимум 2 точки для расчета скорости  if (object->trajectory.*size*() < 2)  return { 0, 0 };  // Берем две последние точки траектории  Point p1 = object->trajectory[object->trajectory.*size*() - 2];  Point p2 = object->trajectory[object->trajectory.*size*() - 1];  // Расчет скорости изменения координат (предполагаем интервал 5 минут между точками)  double speedLat = (p2.latitude - p1.latitude) / 5.0; // Скорость по широте (градусов/мин)  double speedLon = (p2.longitude - p1.longitude) / 5.0; // Скорость по долготе (градусов/мин)  // Расчет прогнозируемой позиции  double newLat = p2.latitude + speedLat \* minutesAhead;  double newLon = p2.longitude + speedLon \* minutesAhead;  return { newLat, newLon };  }  // Проверка потенциальных столкновений между объектами  void findCollisions(int minutesAhead) {  *vector*<*pair*<Object\*, *pair*<double, double>>> predictions;  // Прогнозирование позиций для всех объектов  for (Object\* object : objects) {  auto pos = predictPosition(object, minutesAhead);  predictions.*push\_back*({ object, pos });  }  // Вывод прогнозируемых позиций  *cout* << "Прогноз на 2025-04-12 10:10:\n";  for (auto& pred : predictions) {  *cout* << pred.*first*->name << ": (" << pred.*second*.*first* << ", " << pred.*second*.*second* << ")\n";  }  // Поиск потенциальных столкновений  *cout* << "\nПотенциальные столкновения:\n";  for (*size\_t* i = 0; i < predictions.*size*(); i++) {  for (*size\_t* j = i + 1; j < predictions.*size*(); j++) {  // Расчет расстояния между объектами (1° ≈ 111 км)  double dist = *sqrt*(*pow*(predictions[i].*second*.*first* - predictions[j].*second*.*first*, 2) +  *pow*(predictions[i].*second*.*second* - predictions[j].*second*.*second*, 2)) \* 111;  // Если расстояние меньше 1 км - потенциальное столкновение  if (dist < 1) {  *cout* << predictions[i].*first*->name << " и " << predictions[j].*first*->name  << " (расстояние: " << dist << " км)\n";  }  }  }  }  };  int *main*() {  *setlocale*(*LC\_ALL*, "ru"); // Настройка локали для вывода русских символов  // Создание и инициализация базы данных  ObjectDB db("objects.txt", "trajectories.txt");  // Поиск потенциальных столкновений через 5 минут  db.findCollisions(5);  return 0;  } | |
| 4 | Реализуйте систему индексирования транзакций с использованием красно-чёрного дерева. Каждая транзакция имеет поля: ID, Сумма, Статус (Active, Committed, Aborted), Время. Постройте красно-чёрное дерево по ID транзакции. Реализуйте систему MVCC (многоверсионное управление конкурентным доступом): каждая операция (вставка, обновление) создаёт новую версию записи. Реализуйте метод, который обеспечивает согласованность (проверяет, что сумма всех активных транзакций не превышает заданный лимит). Реализуйте метод отката до заданного времени.  **Входные данные** Файл transactions.txt:  ID Amount Status Timestamp  501 100 Committed 2025-04-12 10:00  502 200 Committed 2025-04-12 10:05  Добавьте транзакции: (503, 300, Active, 2025-04-12 10:10), (504, 400, Active, 2025-04-12 10:15). Проверьте согласованность (лимит 600). Откатите до 2025-04-12 10:05.  **Ожидаемый результат**  Сумма активных транзакций: 700  Нарушение лимита (600)!  После отката до 2025-04-12 10:05:  ID 501 Amount 100  ID 502 Amount 200     |  | | --- | | #include <iostream>  #include <fstream>  #include <string>  #include <sstream>  #include <vector>  using namespace *std*;  // Структура для хранения версии транзакции  struct Version {  int amount; // Сумма транзакции  *string* status; // Статус ("Active", "Cancelled" и т.д.)  *string* timestamp; // Временная метка (формат ГГГГ-ММ-ДД ЧЧ:ММ)  };  // Структура для транзакции с историей версий  struct Transaction {  int id; // Уникальный идентификатор транзакции  *vector*<Version> versions; // История изменений транзакции  // Конструктор транзакции (создает первую версию)  Transaction(int \_id, int \_amount, *string* \_status, *string* \_ts) {  id = \_id;  Version v;  v.amount = \_amount;  v.status = \_status;  v.timestamp = \_ts;  versions.*push\_back*(v); // Добавляем начальную версию  }  };  // Цвета узлов для красно-черного дерева  enum Color { RED, BLACK };  // Узел красно-черного дерева  struct RBNode {  int key; // Ключ (id транзакции)  Transaction\* data; // Указатель на данные транзакции  RBNode\* left; // Левый потомок  RBNode\* right; // Правый потомок  RBNode\* parent; // Родительский узел  Color color; // Цвет узла  // Конструктор узла  RBNode(int k, Transaction\* d) {  key = k;  data = d;  left = nullptr;  right = nullptr;  parent = nullptr;  color = RED; // Новые узлы всегда красные  }  };  // Красно-черное дерево для хранения транзакций  struct RBTree {  RBNode\* root; // Корень дерева  RBNode\* nil; // Специальный nil-узел (листья дерева)  // Конструктор дерева  RBTree() {  nil = new RBNode(0, nullptr);  nil->color = BLACK; // Nil-узлы всегда черные  nil->*left* = nil;  nil->*right* = nil;  root = nil;  }  // Левый поворот вокруг узла x  void leftRotate(RBNode\* x) {  RBNode\* y = x->*right*; // y становится правым потомком x  x->*right* = y->*left*; // Левый потомок y становится правым потомком x  if (y->*left* != nil)  y->*left*->parent = x;  y->parent = x->parent; // Перенос родителя  if (x->parent == nullptr)  root = y;  else if (x == x->parent->*left*)  x->parent->*left* = y;  else  x->parent->*right* = y;  y->*left* = x; // x становится левым потомком y  x->parent = y;  }  // Правый поворот вокруг узла y  void rightRotate(RBNode\* y) {  RBNode\* x = y->*left*; // x становится левым потомком y  y->*left* = x->*right*; // Правый потомок x становится левым потомком y  if (x->*right* != nil)  x->*right*->parent = y;  x->parent = y->parent; // Перенос родителя  if (y->parent == nullptr)  root = x;  else if (y == y->parent->*right*)  y->parent->*right* = x;  else  y->parent->*left* = x;  x->*right* = y; // y становится правым потомком x  y->parent = x;  }  // Восстановление свойств дерева после вставки  void fixInsert(RBNode\* z) {  // Пока родитель красный (нарушение свойства красно-черного дерева)  while (z->parent != nullptr && z->parent->color == RED) {  if (z->parent == z->parent->parent->*left*) {  RBNode\* y = z->parent->parent->*right*; // Дядя узла z  if (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: дядя черный, z - правый потомок  z = z->parent;  leftRotate(z);  }  // Случай 3: дядя черный, z - левый потомок  z->parent->color = BLACK;  z->parent->parent->color = RED;  rightRotate(z->parent->parent);  }  }  else {  // Симметричные случаи для правого поддерева  RBNode\* y = z->parent->parent->*left*;  if (y->color == RED) {  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->*left*) {  z = z->parent;  rightRotate(z);  }  z->parent->color = BLACK;  z->parent->parent->color = RED;  leftRotate(z->parent->parent);  }  }  }  root->color = BLACK; // Корень всегда черный  }  // Вставка новой транзакции в дерево  void insert(int key, Transaction\* data) {  RBNode\* z = new RBNode(key, data);  z->*left* = nil;  z->*right* = nil;  RBNode\* y = nullptr;  RBNode\* x = root;  // Поиск места для вставки  while (x != nil) {  y = x;  if (z->key < x->key)  x = x->*left*;  else  x = x->*right*;  }  // Установка родителя  z->parent = y;  if (y == nullptr)  root = z;  else if (z->key < y->key)  y->*left* = z;  else  y->*right* = z;  // Балансировка дерева  fixInsert(z);  }  // Рекурсивный вывод транзакций на момент указанного времени  void printInOrder(RBNode\* node, *string* maxTimestamp) {  if (node == nil) return;  printInOrder(node->*left*, maxTimestamp);  Transaction\* trans = node->*data*;  // Ищем последнюю версию на момент maxTimestamp  for (int i = trans->versions.*size*() - 1; i >= 0; i--) {  if (trans->versions[i].timestamp <= maxTimestamp) {  *cout* << "ID " << trans->id << " Amount " << trans->versions[i].amount << "\n";  break;  }  }  printInOrder(node->*right*, maxTimestamp);  }  // Рекурсивный подсчет суммы активных транзакций  int sumActive(RBNode\* node) {  if (node == nil) return 0;  int sum = sumActive(node->*left*);  Transaction\* trans = node->*data*;  // Проверяем последнюю версию транзакции  if (trans->versions.*back*().status == "Active") {  sum += trans->versions.*back*().amount;  }  sum += sumActive(node->*right*);  return sum;  }  };  // Класс для работы с базой данных транзакций  struct TransactionDB {  RBTree index; // Красно-черное дерево для индексации  *vector*<Transaction\*> transactions; // Все транзакции  *string* inputFile; // Имя входного файла  // Конструктор базы данных  TransactionDB(*string* file) {  inputFile = file;  loadFromFile(); // Загрузка данных при создании  }  // Загрузка транзакций из файла  void loadFromFile() {  *ifstream* file(inputFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << inputFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  int id, amount;  *string* status, timestamp;  ss >> id >> amount >> status;  ss.*ignore*(1);  *getline*(ss, timestamp);  // Создание новой транзакции и добавление в базу  Transaction\* trans = new Transaction(id, amount, status, timestamp);  transactions.*push\_back*(trans);  index.insert(id, trans); // Вставка в индексное дерево  }  file.*close*();  }  // Добавление новой транзакции  void insert(int id, int amount, *string* status, *string* timestamp) {  Transaction\* trans = new Transaction(id, amount, status, timestamp);  transactions.*push\_back*(trans);  index.insert(id, trans);  }  // Проверка соблюдения лимита активных транзакций  void checkConsistency(int limit) {  int total = index.sumActive(index.root);  *cout* << "Сумма активных транзакций: " << total << "\n";  if (total > limit) {  *cout* << "Нарушение лимита (" << limit << ")!\n";  }  else {  *cout* << "Лимит соблюдён.\n";  }  }  // Откат системы до указанного времени  void rollbackTo(*string* timestamp) {  *cout* << "После отката до " << timestamp << ":\n";  index.printInOrder(index.root, timestamp);  }  };  int *main*() {  *setlocale*(*LC\_ALL*, "ru"); // Настройка локали для вывода русских символов  // Создание и инициализация базы данных  TransactionDB db("transactions.txt");  // Добавление тестовых транзакций  db.insert(503, 300, "Active", "2025-04-12 10:10");  db.insert(504, 400, "Active", "2025-04-12 10:15");  // Проверка лимита (макс. 600)  db.checkConsistency(600);  // Откат системы до указанного времени  db.rollbackTo("2025-04-12 10:05");  return 0;  } | |
| 5 | Реализуйте систему индексирования слов из текстового корпуса с использованием красно-чёрного дерева. Каждое слово имеет поля: само слово (уникальный ключ), частота встречаемости, список документов (ID документа, где слово встречается), список связанных слов (слова, которые встречаются в одном предложении с данным словом). Постройте красно-чёрное дерево по строковому ключу (слову). Реализуйте следующие функции:   1. **Анализ связей слов**: Найдите компоненты связности в графе слов, где рёбра соединяют слова, которые встречаются в одном предложении. 2. **Рекомендация слов**: Для заданного слова найдите слова, которые часто встречаются с ним в одном предложении, и отсортируйте их по частоте совместного появления. 3. **Анализ редких слов**: Найдите слова, которые встречаются реже заданного порога (например, частота < 3), но имеют высокую степень связности (много связанных слов). 4. **Оптимизация корпуса**: Удалите из корпуса слова, которые встречаются только в одном документе и имеют низкую частоту (например, < 2), и обновите связи между оставшимися словами.   **Входные данные** Файл words.txt:  Word Frequency DocIDs  apple 5 1,2,3  banana 3 1,2  cherry 2 1  date 1 3  Файл cooccurrences.txt:  Word1 Word2  apple banana  apple cherry  banana cherry  **Действия**:   1. Найдите компоненты связности в графе слов. 2. Для слова "apple" найдите рекомендации (слова, с которыми оно часто встречается). 3. Найдите редкие слова с высокой степенью связности (порог частоты: 3). 4. Оптимизируйте корпус, удалив слова с частотой < 2 и встречающиеся только в одном документе, и обновите связи.   **Ожидаемый результат**  Компоненты связности:  Компонента 1: apple, banana, cherry  Компонента 2: date  Рекомендации для слова "apple":  banana (совместно: 1), cherry (совместно: 1)  Редкие слова с высокой степенью связности (порог 3):  cherry (частота: 2, связанных слов: 2)  После оптимизации корпуса:  Удалено слово: date  Оставшиеся слова: apple, banana, cherry  Обновлённые связи:  apple: banana, cherry  banana: apple, cherry  cherry: apple, banana     |  | | --- | | #include <iostream>  #include <fstream>  #include <string>  #include <sstream>  #include <vector>  #include <map>  #include <set>  using namespace *std*;  // Структура для хранения информации о слове  struct Word {  *string* word; // Само слово  int frequency; // Частота встречаемости  *vector*<int> docIDs; // ID документов, где встречается слово  *vector*<*string*> relatedWords; // Связанные слова (встречаются в одном контексте)  *map*<*string*, int> cooccurrenceCount; // Частота совместного появления с другими словами  // Конструктор слова  Word(*string* \_word, int \_freq) {  word = \_word;  frequency = \_freq;  }  };  // Цвета узлов для красно-черного дерева  enum Color { RED, BLACK };  // Узел красно-черного дерева  struct RBNode {  *string* key; // Ключ (слово)  Word\* data; // Указатель на данные слова  RBNode\* left; // Левый потомок  RBNode\* right; // Правый потомок  RBNode\* parent; // Родительский узел  Color color; // Цвет узла  // Конструктор узла  RBNode(*string* k, Word\* d) {  key = k;  data = d;  left = nullptr;  right = nullptr;  parent = nullptr;  color = RED; // Новые узлы всегда красные  }  };  // Красно-черное дерево для хранения слов  struct RBTree {  RBNode\* root; // Корень дерева  RBNode\* nil; // Специальный nil-узел (листья дерева)  // Конструктор дерева  RBTree() {  nil = new RBNode("", nullptr);  nil->color = BLACK; // Nil-узлы всегда черные  nil->*left* = nil;  nil->*right* = nil;  root = nil;  }  // Деструктор дерева (рекурсивное удаление узлов)  ~RBTree() {  deleteTree(root);  delete nil;  }  // Рекурсивное удаление узлов дерева  void deleteTree(RBNode\* node) {  if (node == nil) return;  deleteTree(node->*left*);  deleteTree(node->*right*);  delete node->*data*; // Удаляем данные слова  delete node; // Удаляем сам узел  }  // Левый поворот вокруг узла x  void leftRotate(RBNode\* x) {  RBNode\* y = x->right; // y становится правым потомком x  x->right = y->left; // Левый потомок y становится правым потомком x  if (y->left != nil)  y->left->parent = x;  y->parent = x->parent; // Перенос родителя  if (x->parent == nullptr)  root = y;  else if (x == x->parent->left)  x->parent->left = y;  else  x->parent->right = y;  y->left = x; // x становится левым потомком y  x->parent = y;  }  // Правый поворот вокруг узла y  void rightRotate(RBNode\* y) {  RBNode\* x = y->left; // x становится левым потомком y  y->left = x->right; // Правый потомок x становится левым потомком y  if (x->right != nil)  x->right->parent = y;  x->parent = y->parent; // Перенос родителя  if (y->parent == nullptr)  root = x;  else if (y == y->parent->right)  y->parent->right = x;  else  y->parent->left = x;  x->right = y; // y становится правым потомком x  y->parent = x;  }  // Восстановление свойств дерева после вставки  void fixInsert(RBNode\* z) {  // Пока родитель красный (нарушение свойства красно-черного дерева)  while (z->parent != nullptr && z->parent->color == RED) {  if (z->parent == z->parent->parent->left) {  RBNode\* y = z->parent->parent->right; // Дядя узла z  if (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: дядя черный, z - правый потомок  z = z->parent;  leftRotate(z);  }  // Случай 3: дядя черный, z - левый потомок  z->parent->color = BLACK;  z->parent->parent->color = RED;  rightRotate(z->parent->parent);  }  }  else {  // Симметричные случаи для правого поддерева  RBNode\* y = z->parent->parent->left;  if (y->color == RED) {  z->parent->color = BLACK;  y->color = BLACK;  z->parent->parent->color = RED;  z = z->parent->parent;  }  else {  if (z == z->parent->left) {  z = z->parent;  rightRotate(z);  }  z->parent->color = BLACK;  z->parent->parent->color = RED;  leftRotate(z->parent->parent);  }  }  }  root->color = BLACK; // Корень всегда черный  }  // Вставка нового слова в дерево  void insert(*string* key, Word\* data) {  RBNode\* z = new RBNode(key, data);  z->left = nil;  z->right = nil;  RBNode\* y = nullptr;  RBNode\* x = root;  // Поиск места для вставки  while (x != nil) {  y = x;  if (z->key < x->key)  x = x->left;  else  x = x->right;  }  // Установка родителя  z->parent = y;  if (y == nullptr)  root = z;  else if (z->key < y->key)  y->left = z;  else  y->right = z;  // Балансировка дерева  fixInsert(z);  }  // Поиск слова в дереве  Word\* search(*string* key) {  RBNode\* current = root;  while (current != nil) {  if (key == current->key)  return current->data;  else if (key < current->key)  current = current->left;  else  current = current->right;  }  return nullptr; // Слово не найдено  }  // Удаление слова из дерева  void remove(*string* key) {  RBNode\* z = root;  while (z != nil) {  if (key == z->key) break;  else if (key < z->key) z = z->left;  else z = z->right;  }  if (z == nil) return;  RBNode\* y = z;  Color yOriginalColor = y->color;  RBNode\* x;  if (z->left == nil) {  x = z->right;  transplant(z, z->right);  }  else if (z->right == nil) {  x = z->left;  transplant(z, z->left);  }  else {  y = minimum(z->right);  yOriginalColor = y->color;  x = y->right;  if (y->parent == z) {  x->parent = y;  }  else {  transplant(y, y->right);  y->right = z->right;  y->right->parent = y;  }  transplant(z, y);  y->left = z->left;  y->left->parent = y;  y->color = z->color;  }  if (yOriginalColor == BLACK) {  fixDelete(x);  }  delete z->data; // Удаляем данные слова  delete z; // Удаляем узел  }  // Замена поддерева  void transplant(RBNode\* u, RBNode\* v) {  if (u->parent == nullptr)  root = v;  else if (u == u->parent->left)  u->parent->left = v;  else  u->parent->right = v;  v->parent = u->parent;  }  // Поиск минимального узла в поддереве  RBNode\* minimum(RBNode\* node) {  while (node->*left* != nil) {  node = node->*left*;  }  return node;  }  // Восстановление свойств дерева после удаления  void fixDelete(RBNode\* x) {  while (x != root && x->color == BLACK) {  if (x == x->parent->left) {  RBNode\* w = x->parent->right;  if (w->color == RED) {  w->color = BLACK;  x->parent->color = RED;  leftRotate(x->parent);  w = x->parent->right;  }  if (w->left->color == BLACK && w->right->color == BLACK) {  w->color = RED;  x = x->parent;  }  else {  if (w->right->color == BLACK) {  w->left->color = BLACK;  w->color = RED;  rightRotate(w);  w = x->parent->right;  }  w->color = x->parent->color;  x->parent->color = BLACK;  w->right->color = BLACK;  leftRotate(x->parent);  x = root;  }  }  else {  RBNode\* w = x->parent->left;  if (w->color == RED) {  w->color = BLACK;  x->parent->color = RED;  rightRotate(x->parent);  w = x->parent->left;  }  if (w->right->color == BLACK && w->left->color == BLACK) {  w->color = RED;  x = x->parent;  }  else {  if (w->left->color == BLACK) {  w->right->color = BLACK;  w->color = RED;  leftRotate(w);  w = x->parent->left;  }  w->color = x->parent->color;  x->parent->color = BLACK;  w->left->color = BLACK;  rightRotate(x->parent);  x = root;  }  }  }  x->color = BLACK;  }  // Обход дерева в порядке возрастания (для получения всех слов)  void inorderTraversal(RBNode\* node, *vector*<*string*>& words) {  if (node == nil) return;  inorderTraversal(node->*left*, words);  words.*push\_back*(node->key);  inorderTraversal(node->*right*, words);  }  // Получение списка всех слов в дереве  *vector*<*string*> getAllWords() {  *vector*<*string*> words;  inorderTraversal(root, words);  return words;  }  };  // Класс для анализа текстовых данных  struct TextAnalyzer {  RBTree index; // Красно-черное дерево для индексации слов  *vector*<Word\*> words; // Все слова  *string* wordsFile; // Файл со словами  *string* cooccurrencesFile; // Файл с совместными появлениями  // Конструктор анализатора  TextAnalyzer(*string* wFile, *string* cFile) {  wordsFile = wFile;  cooccurrencesFile = cFile;  loadWords(); // Загрузка слов при создании  loadCooccurrences(); // Загрузка совместных появлений  }  // Деструктор (освобождение памяти)  ~TextAnalyzer() {  for (Word\* word : words) {  delete word;  }  }  // Загрузка слов из файла  void loadWords() {  *ifstream* file(wordsFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << wordsFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  *string* word;  int freq;  *string* docIDsStr;  if (!(ss >> word >> freq >> docIDsStr)) {  *cout* << "Ошибка: Некорректный формат строки в файле " << wordsFile << ": " << line << "\n";  continue;  }  // Создание нового слова и добавление в базу  Word\* w = new Word(word, freq);  *stringstream* docSS(docIDsStr);  *string* docID;  while (*getline*(docSS, docID, ',')) {  w->docIDs.*push\_back*(*stoi*(docID));  }  words.*push\_back*(w);  index.insert(word, w); // Вставка в индексное дерево  }  file.*close*();  }  // Загрузка информации о совместных появлениях слов  void loadCooccurrences() {  *ifstream* file(cooccurrencesFile);  if (!file.*is\_open*()) {  *cout* << "Ошибка: Не удалось открыть файл " << cooccurrencesFile << "!\n";  return;  }  *string* line;  *getline*(file, line); // Пропуск заголовка  while (*getline*(file, line)) {  if (line.*empty*()) continue;  *stringstream* ss(line);  *string* word1, word2;  if (!(ss >> word1 >> word2)) {  *cout* << "Ошибка: Некорректный формат строки в файле " << cooccurrencesFile << ": " << line << "\n";  continue;  }  // Поиск слов в индексе  Word\* w1 = index.search(word1);  Word\* w2 = index.search(word2);  if (w1 && w2) {  // Добавление информации о совместном появлении  w1->relatedWords.*push\_back*(word2);  w1->cooccurrenceCount[word2]++;  w2->relatedWords.*push\_back*(word1);  w2->cooccurrenceCount[word1]++;  }  else {  *cout* << "Ошибка: Слово " << word1 << " или " << word2 << " не найдено!\n";  }  }  file.*close*();  }  // Поиск компонент связности графа слов (алгоритм DFS)  void findConnectedComponents() {  if (words.*empty*()) {  *cout* << "Компоненты связности:\nНет слов.\n";  return;  }  *set*<*string*> visited; // Множество посещенных слов  *vector*<*vector*<*string*>> components; // Найденные компоненты связности  for (Word\* word : words) {  if (!visited.*count*(word->word)) {  *vector*<*string*> component;  dfs(word->word, visited, component); // Обход в глубину  components.*push\_back*(component);  }  }  // Вывод результатов  *cout* << "Компоненты связности:\n";  for (*size\_t* i = 0; i < components.*size*(); i++) {  *cout* << "Компонента " << (i + 1) << ": ";  for (*size\_t* j = 0; j < components[i].*size*(); j++) {  *cout* << components[i][j];  if (j < components[i].*size*() - 1) *cout* << ", ";  }  *cout* << "\n";  }  }  // Обход в глубину для поиска компонент связности  void dfs(*string* word, *set*<*string*>& visited, *vector*<*string*>& component) {  Word\* w = index.search(word);  if (!w) return;  visited.*insert*(word);  component.*push\_back*(word);  // Рекурсивный обход всех связанных слов  for (const *string*& related : w->relatedWords) {  if (!visited.*count*(related)) {  dfs(related, visited, component);  }  }  }  // Рекомендация слов на основе совместных появлений  void recommendWords(*string* targetWord) {  Word\* w = index.search(targetWord);  if (!w) {  *cout* << "Слово " << targetWord << " не найдено.\n";  return;  }  *cout* << "Рекомендации для слова \"" << targetWord << "\":\n";  *vector*<pair<*string*, int>> recommendations;  // Сбор рекомендаций  for (const auto& pair : w->cooccurrenceCount) {  recommendations.*push\_back*(pair);  }  // Сортировка по частоте совместного появления (по убыванию)  for (*size\_t* i = 0; i < recommendations.*size*(); i++) {  for (*size\_t* j = i + 1; j < recommendations.*size*(); j++) {  if (recommendations[i].*second* < recommendations[j].*second*) {  *swap*(recommendations[i], recommendations[j]);  }  }  }  // Вывод результатов  if (recommendations.*empty*()) {  *cout* << "Нет рекомендаций.\n";  }  else {  for (const auto& rec : recommendations) {  *cout* << rec.*first* << " (совместно: " << rec.*second* << ")";  if (&rec != &recommendations.*back*()) *cout* << ", ";  }  *cout* << "\n";  }  }  // Поиск редких слов с высокой степенью связности  void findRareConnectedWords(int freqThreshold) {  *cout* << "Редкие слова с высокой степенью связности (порог " << freqThreshold << "):\n";  bool found = false;  for (Word\* w : words) {  if (w->frequency < freqThreshold && w->relatedWords.*size*() >= 2) {  *cout* << w->word << " (частота: " << w->frequency << ", связанных слов: " << w->relatedWords.*size*() << ")\n";  found = true;  }  }  if (!found) *cout* << "Нет подходящих слов.\n";  }  // Оптимизация корпуса (удаление редких и малоиспользуемых слов)  void optimizeCorpus(int freqThreshold) {  *vector*<*string*> wordsToRemove;  // Поиск слов для удаления  for (Word\* w : words) {  if (w->frequency < freqThreshold && w->docIDs.*size*() <= 1) {  wordsToRemove.*push\_back*(w->word);  }  }  // Удаление слов из индекса  for (const *string*& word : wordsToRemove) {  *cout* << "Удалено слово: " << word << "\n";  index.remove(word);  }  // Обновление списка слов  *vector*<Word\*> newWords;  for (Word\* w : words) {  if (*find*(wordsToRemove.*begin*(), wordsToRemove.*end*(), w->word) == wordsToRemove.*end*()) {  newWords.*push\_back*(w);  }  else {  delete w; // Освобождение памяти  }  }  words = newWords;  // Обновление связей между словами  for (Word\* w : words) {  *vector*<*string*> newRelatedWords;  *map*<*string*, int> newCooccurrenceCount;  // Фильтрация связанных слов  for (const *string*& related : w->relatedWords) {  if (index.search(related)) {  newRelatedWords.*push\_back*(related);  newCooccurrenceCount[related] = w->cooccurrenceCount[related];  }  }  w->relatedWords = newRelatedWords;  w->cooccurrenceCount = newCooccurrenceCount;  }  // Вывод результатов оптимизации  *cout* << "Оставшиеся слова: ";  for (*size\_t* i = 0; i < words.*size*(); i++) {  *cout* << words[i]->word;  if (i < words.*size*() - 1) *cout* << ", ";  }  *cout* << "\nОбновлённые связи:\n";  for (Word\* w : words) {  *cout* << w->word << ": ";  for (*size\_t* i = 0; i < w->relatedWords.*size*(); i++) {  *cout* << w->relatedWords[i];  if (i < w->relatedWords.*size*() - 1) *cout* << ", ";  }  *cout* << "\n";  }  }  };  int *main*() {  *setlocale*(*LC\_ALL*, "ru"); // Настройка локали для русских символов  // Создание и инициализация анализатора  TextAnalyzer analyzer("words.txt", "cooccurrences.txt");  // Анализ компонент связности  analyzer.findConnectedComponents();  // Получение рекомендаций для слова "apple"  analyzer.recommendWords("apple");  // Поиск редких, но связанных слов  analyzer.findRareConnectedWords(3);  // Оптимизация корпуса (удаление редких слов)  analyzer.optimizeCorpus(2);  return 0;  } | |

1. **Ответы к контрольным вопросам**

**1. Что такое красно-чёрное дерево и почему оно называется именно так?**

**Ответ:**  
Красно-чёрное дерево — это сбалансированное бинарное дерево поиска, в котором каждый узел имеет цвет (красный или чёрный). Название происходит от этой особенности: цвета узлов используются для поддержания баланса, чтобы дерево не вырождалось в список, обеспечивая сложность операций O(log n).

**2. Перечислите основные правила красно-чёрного дерева.**

**Ответ:**

1. Каждый узел — либо красный, либо чёрный.
2. Корень всегда чёрный.
3. Все листья (NULL-узлы) — чёрные.
4. У красного узла не может быть красных детей (нет двух красных узлов подряд).
5. Чёрная высота (количество чёрных узлов на пути от узла до листа) одинакова для всех путей.

**3. Почему красно-чёрное дерево подходит для индексирования в базах данных?**

**Ответ:**  
Красно-чёрное дерево подходит для индексирования, потому что:

* Обеспечивает логарифмическую сложность операций (поиск, вставка, удаление — O(log n)).
* Автоматически балансируется, предотвращая вырождение в список.
* Эффективно обновляется при добавлении или удалении данных.
* Поддерживает сравнение ключей любого типа (например, ID, строки).

**4. Какой цвет имеет новый узел при вставке в красно-чёрное дерево и почему?**

**Ответ:**  
Новый узел при вставке всегда красный. Это связано с тем, что добавление красного узла меньше нарушает правила красно-чёрного дерева (например, чёрную высоту), чем добавление чёрного, и требует меньше операций для восстановления баланса.

**5. Опишите случай балансировки при вставке, когда у нового узла красный родитель и красный дядя.**

**Ответ:**  
Если у нового узла красный родитель и красный дядя:

* Родитель и дядя перекрашиваются в чёрный.
* Дед становится красным.
* Проверяется дед: если его родитель красный, процесс балансировки продолжается выше по дереву.  
  Этот случай встречается с вероятностью около 25%.

**6. Какой случай балансировки при удалении возникает, если брат удаляемого узла красный?**

**Ответ:**  
Если брат удаляемого узла красный:

* Брат становится чёрным.
* Родитель становится красным.
* Выполняется левый поворот вокруг родителя.
* После этого случай сводится к другим сценариям (например, брат становится чёрным).  
  Этот случай встречается с вероятностью около 50% при удалении.

**7. Что такое чёрная высота и почему она важна для красно-чёрного дерева?**

**Ответ:**  
Чёрная высота — это количество чёрных узлов на пути от данного узла до листа (NULL), не включая сам узел, но включая NULL. Она важна, потому что правило одинаковой чёрной высоты на всех путях гарантирует сбалансированность дерева, обеспечивая, что самый длинный путь не более чем в 2 раза длиннее самого короткого.

**8. Как красно-чёрное дерево обеспечивает логарифмическую сложность операций?**

**Ответ:**  
Красно-чёрное дерево поддерживает баланс благодаря своим правилам (например, нет двух красных узлов подряд, одинаковая чёрная высота). Это гарантирует, что высота дерева не превышает 2 \* log(n + 1), где n — число узлов, что обеспечивает сложность операций (поиск, вставка, удаление) O(log n).

**9. В чём преимущество красно-чёрного дерева перед обычным бинарным деревом поиска (BST) при использовании в базах данных?**

**Ответ:**  
Преимущество красно-чёрного дерева перед BST:

* Оно автоматически балансируется, не вырождаясь в список, в отличие от BST, где в худшем случае операции могут иметь сложность O(n).
* Гарантирует O(log n) для операций, что критично для баз данных с большим количеством записей.

**10. Какие ограничения красно-чёрного дерева для индексирования в базах данных?**

**Ответ:**  
Ограничения красно-чёрного дерева:

* Сложность реализации: алгоритмы вставки и удаления сложнее из-за балансировки.
* Менее эффективно для дисковых операций по сравнению с B-деревьями, так как хранит только один ключ в узле.
* Диапазонные запросы менее оптимизированы, чем в B+-деревьях, где ключи хранятся в листьях.

**11. Какой метод используется в программе для поиска студента по ID? Опишите его работу.**

**Ответ:**  
Метод search в структуре RBTree используется для поиска студента по ID:

* Начинается с корня дерева.
* Сравнивает ключ (ID) с текущим узлом: если ключ равен, возвращает данные узла; если меньше — идёт влево, если больше — вправо.
* Повторяет, пока не найдёт узел или не дойдёт до листа (nil).  
  Сложность — O(log n) благодаря сбалансированности дерева.

**12. Как в программе реализовано удаление студента из базы данных?**

**Ответ:**  
Удаление студента в программе:

1. Метод deleteStudent вызывает index.remove(id) для удаления узла из красно-чёрного дерева.
2. В дереве:
   * Находит узел с заданным ID.
   * Если у узла один ребёнок или нет детей, заменяет его на ребёнка.
   * Если два ребёнка, находит преемника (минимальный узел в правом поддереве), заменяет удаляемый узел на преемника и корректирует дерево.
   * Выполняет балансировку (fixDelete), если удалённый узел был чёрным.
3. Удаляет студента из вектора students и переписывает файл.

**13. Почему в программе используется вектор students в дополнение к красно-чёрному дереву?**

**Ответ:**  
Вектор students используется для:

* Хранения всех записей о студентах в порядке загрузки из файла.
* Упрощения операций, таких как переписывание файла после удаления или вставки.
* Обеспечения доступа к данным без необходимости обхода дерева, если порядок важен.  
  Красно-чёрное дерево же используется для быстрого поиска по ID.

**14. Как красно-чёрное дерево может быть использовано для диапазонных запросов?**

**Ответ:**  
Для диапазонных запросов (например, найти студентов с ID от 1531 до 1535):

* Выполняется обход дерева (например, инфиксный).
* Проверяется каждый узел: если его ключ попадает в диапазон, он включается в результат.
* Поддеревья, которые заведомо не содержат нужных ключей (например, левое поддерево, если ключ меньше минимального значения диапазона), пропускаются.  
  Это менее эффективно, чем в B+-деревьях, но всё же возможно.

**15. Сравните красно-чёрное дерево с B-деревом для задач индексирования.**

**Ответ:**

* **Красно-чёрное дерево:**
  + Плюсы: проще реализация, эффективно для операций в памяти, O(log n) сложность.
  + Минусы: не оптимизировано для дисковых операций (много узлов — много обращений к диску).
* **B-дерево:**
  + Плюсы: оптимизировано для дисков (хранит больше ключей в узле, меньше обращений), лучше для диапазонных запросов.
  + Минусы: сложнее реализация, больше памяти на узел.  
    Красно-чёрное дерево лучше для задач в оперативной памяти, B-дерево — для больших баз данных на диске.