Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Российский химико-технологический университет имени Д.И. Менделеева»

Факультет цифровых технологий и химического инжиниринга

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 7

ПО КУРСУ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

Ведущий преподаватель

Ассистент Крашенинников Р. С.

СТУДЕНТ группы КС-36 Лупинос А. В.

Москва

2025

Задание

В рамках лабораторной работы необходимо изучить одно из двух деревьев поиска: рандомизированное дерево.

Для этого потребуется реализовать выбранное дерево и сравнить его работу с ранее реализованным AVL-деревом. Для анализа работы алгоритма необходимо провести серии тестов:

- В одной серии тестов проводится 50 повторений.
- Требуется провести серии тестов для $N = 2^i$ элементов, где i принимает значения от 10 до 18 включительно.

В рамках одной серии необходимо выполнить следующее:

1. Сгенерировать N случайных значений.
2. Заполнить два дерева N элементами в одинаковом порядке.
3. Для каждой серии тестов замерить максимальную глубину полученных деревьев.
4. Для каждого дерева после заполнения провести 1000 операций вставки и замерить время.
5. Для каждого дерева после заполнения провести 1000 операций удаления и замерить время.
6. Для каждого дерева после заполнения провести 1000 операций поиска и замерить время.
7. Для каждого дерева замерить глубины всех веток.

Для анализа структуры потребуется построить следующие графики:

1. График зависимости среднего времени вставки от количества элементов в изначальном дереве для вашего варианта дерева и AVL-дерева.
2. График зависимости среднего времени удаления от количества элементов в изначальном дереве для вашего варианта дерева и AVL-дерева.

3. График зависимости среднего времени поиска от количества элементов в изначальном дереве для вашего варианта дерева и AVL-дерева.
4. График зависимости максимальной высоты полученного дерева от N.
5. Гистограмму распределения максимальной высоты для последней серии тестов для AVL-дерева и вашего варианта дерева.
6. Гистограмму распределения глубин веток для AVL-дерева и вашего варианта дерева для последней серии тестов.

Задания со звездочкой (+5 дополнительных первичных баллов):

1. Провести аналогичную серию тестов и сравнить результаты для заранее отсортированного набора данных.
2. Реализовать красно-черное дерево и провести все те же проверки с ним.

Описание алгоритма

Дерево — это иерархическая структура данных, состоящая из узлов, связанных между собой направленными ребрами. В отличие от линейных структур, таких как массивы или списки, дерево имеет нелинейную организацию, где каждый узел может иметь потомков. Основные термины, используемые для описания деревьев:

- Узел (node): элемент дерева, содержащий данные и ссылки на своих потомков.
- Корень (root): самый верхний узел дерева, с которого начинается структура.
- Родитель (parent): узел, который имеет потомков.
- Потомок (child): узел, который является непосредственным подузлом другого узла.
- Лист (leaf): узел, не имеющий потомков.
- Поддерево (subtree): часть дерева, начинающаяся с какого-либо узла и включающая всех его потомков.
- Высота (height): максимальное расстояние от корня до листа, измеряемое количеством ребер.
- Глубина (depth): расстояние от корня до конкретного узла.

Дерево является **двоичным**, если каждый узел имеет не более двух потомков — левого и правого. Двоичные деревья широко используются в информатике благодаря своей простоте и эффективности при реализации различных алгоритмов, таких как поиск, сортировка и организация данных. Данное дерево представлено на рисунке 1.

Свойства двоичного дерева:

- Максимальное количество узлов на уровне k равно 2^k, где k — номер уровня (начиная с 0).
- Максимальное количество узлов в дереве высоты h равно $2^{h+1} - 1$.
- Минимальная высота дерева с N узлами равна $log_2 N$.
- Значения узлов уникальны.

Двоичное дерево поиска (Binary Search Tree, BST) — это структура данных, которая представляет собой дерево, где каждый узел имеет не более двух потомков: левого и правого. Узлы в двоичном дереве поиска организованы по определенному правилу: для каждого узла *N* выполняются следующие условия:

- Значение левого поддерева узла *N* (если оно существует) меньше значения узла *N*.
- Значение правого поддерева узла *N* (если оно существует) больше значения узла *N*.

Таким образом, структура двоичного дерева поиска позволяет эффективно выполнять операции поиска, вставки и удаления за счет упорядоченности данных. Визуально двоичное дерево поиска можно представить следующим образом: корень дерева — это первый узел, а каждое поддерево (левое и правое) само является двоичным деревом поиска.

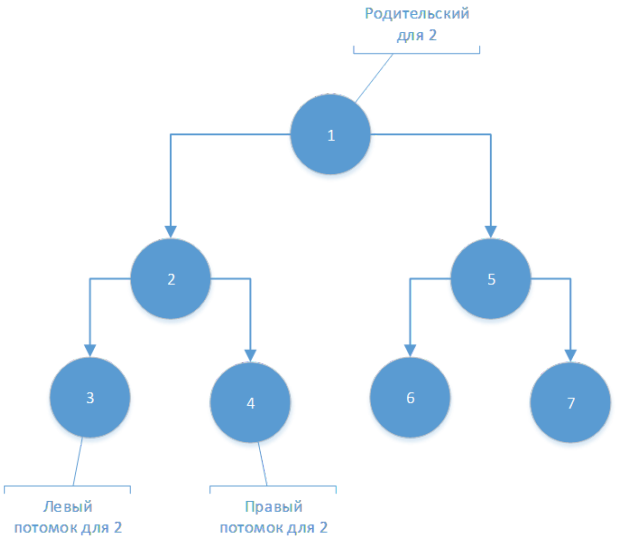

Рисунок 1 – Двоичное дерево

Основные операции и их сложности:

- Поиск:

Для поиска элемента в двоичном дереве поиска мы начинаем с корня и сравниваем искомое значение с текущим узлом. Если значение меньше, переходим в левое поддерево, если больше — в правое. В лучшем случае, когда дерево

сбалансировано, сложность поиска составляет $O(log_2 N)$, где N — количество узлов. Однако в худшем случае, если дерево вырождается в линейную структуру (как в примере выше), сложность становится $O(N)$.

- Вставка:

Для вставки нового элемента мы также начинаем с корня и движемся вниз, пока не найдем подходящее место для нового узла. Сложность вставки аналогична поиску: $O(log_2 N)$ в лучшем случае и $O(N)$ в худшем.

- Удаление:

Удаление узла сложнее, так как может потребоваться реорганизация дерева. Если удаляемый узел имеет два поддерева, его заменяют на минимальный элемент из правого поддерева (или максимальный из левого). Сложность удаления также составляет $O(log_2 N)$ в лучшем случае и $O(N)$ в худшем.

Основной недостаток обычного двоичного дерева поиска заключается в том, что его производительность сильно зависит от порядка вставки элементов. Если элементы вставляются в отсортированном порядке (например, 1, 2, 3, …), дерево становится несбалансированным, и его высота становится равной количеству узлов N. Это приводит к тому, что операции, которые должны выполняться за $O(log_2 N)$, выполняются за $O(N)$, что делает такое дерево неэффективным.

AVL-дерево — это самобалансирующееся двоичное дерево поиска, названное в честь его создателей Адельсона-Вельского и Ландиса. Главное отличие AVL-дерева от обычного BST заключается в том, что оно поддерживает баланс высот своих поддеревьев. Для каждого узла N в AVL-дереве выполняется следующее условие баланса: разница между высотой левого и правого поддерева узла N (так называемый фактор баланса) не превышает по модулю 1 ($|h_{\text{левое}} - h_{\text{правое}}| \leq 1$, где $h_{\text{левое}}$ и $h_{\text{правое}}$ — высоты левого и правого поддеревьев соответственно).

Как работает балансировка? После каждой операции вставки или удаления AVL-дерево проверяет, не нарушился ли баланс. Если фактор баланса какого-либо узла становится больше 1 или меньше −1, дерево выполняет одну из следующих операций балансировки (так называемые повороты):

- Простой левый поворот: используется, когда правое поддерево стало слишком высоким (рис. 2).

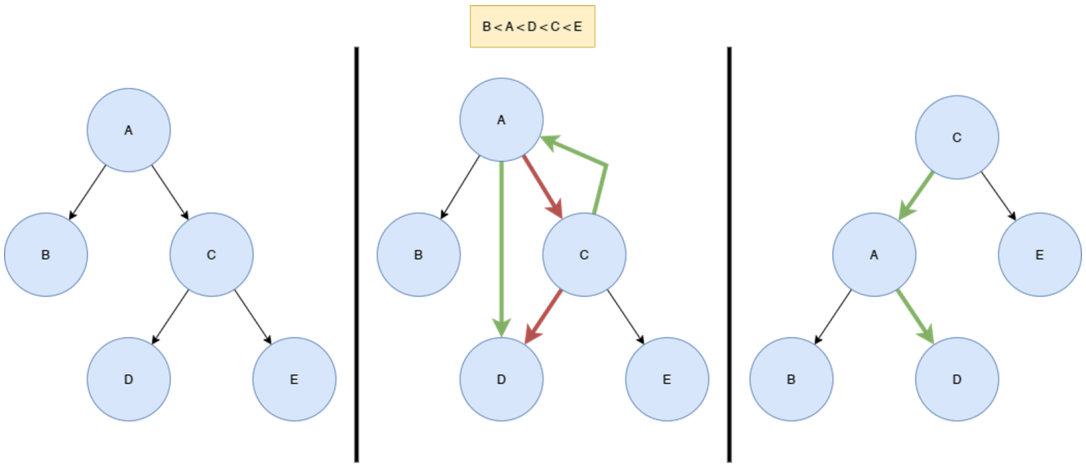

Рисунок 2 – Простой левый поворот

- Простой правый поворот: используется, когда левое поддерево стало слишком высоким (рис. 3).

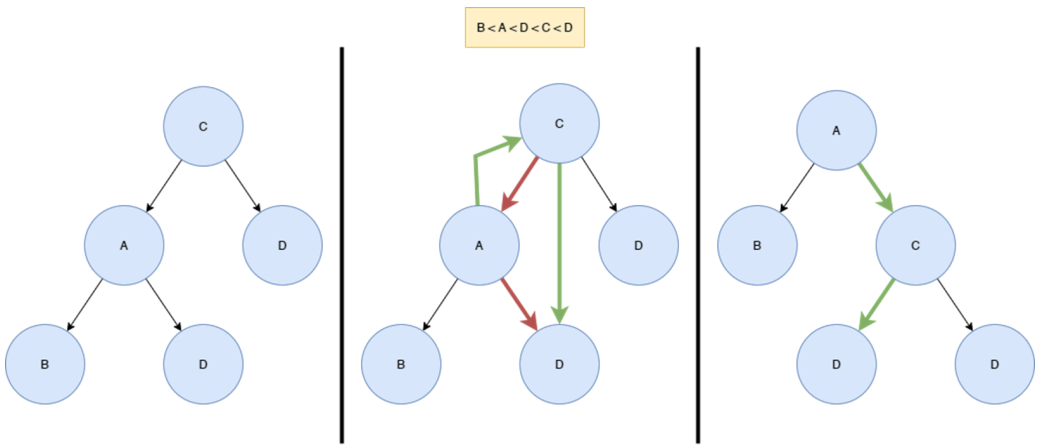

Рисунок 3 – Простой правый поворот

- Сложный левый-правый поворот: выполняется, когда левое поддерево слишком высоко, но само левое поддерево имеет более высокое правое поддерево (рис. 4а и 4б).

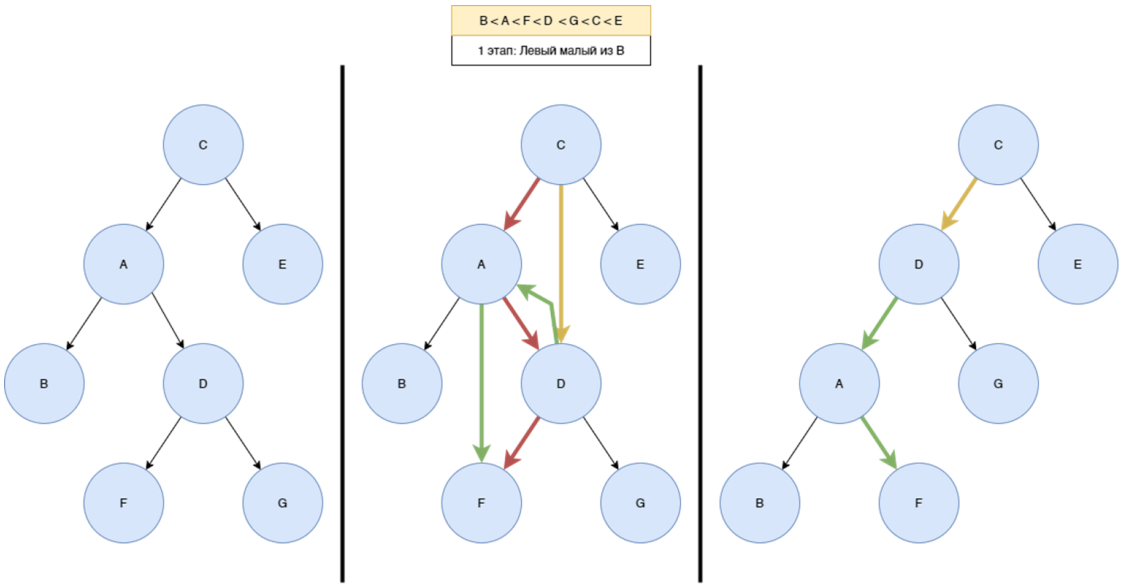

Рисунок 4а – Левый-правый поворот, 1 этап

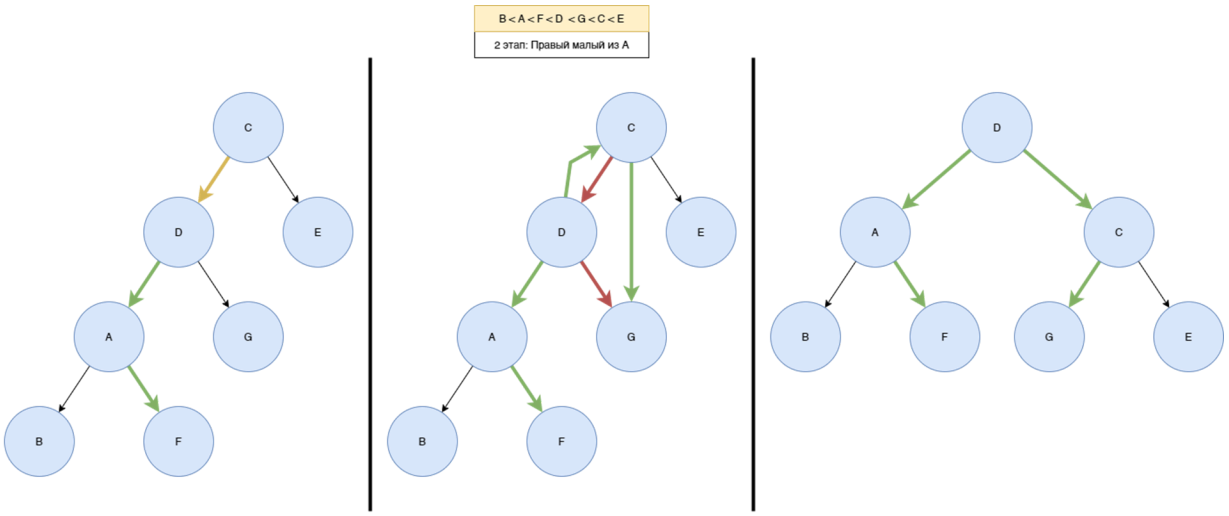

Рисунок 4б – Левый-правый поворот, 2 этап

- Сложный правый-левый поворот: выполняется, когда правое поддерево слишком высоко, но само правое поддерево имеет более высокое левое поддерево (рис. 5а и 5б).

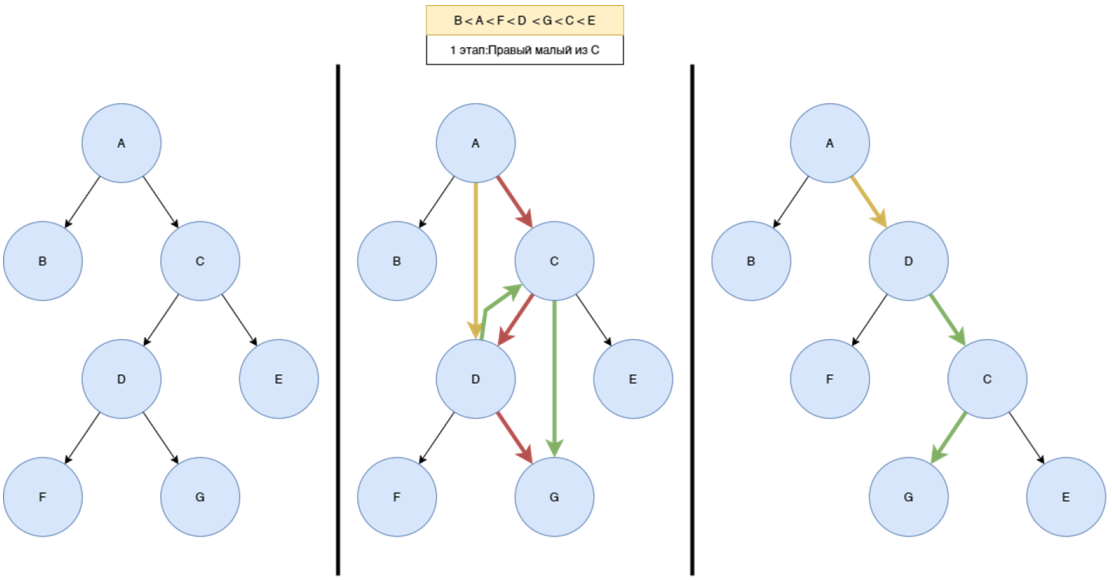

Рисунок 5а – Правый-левый поворот, 1 этап

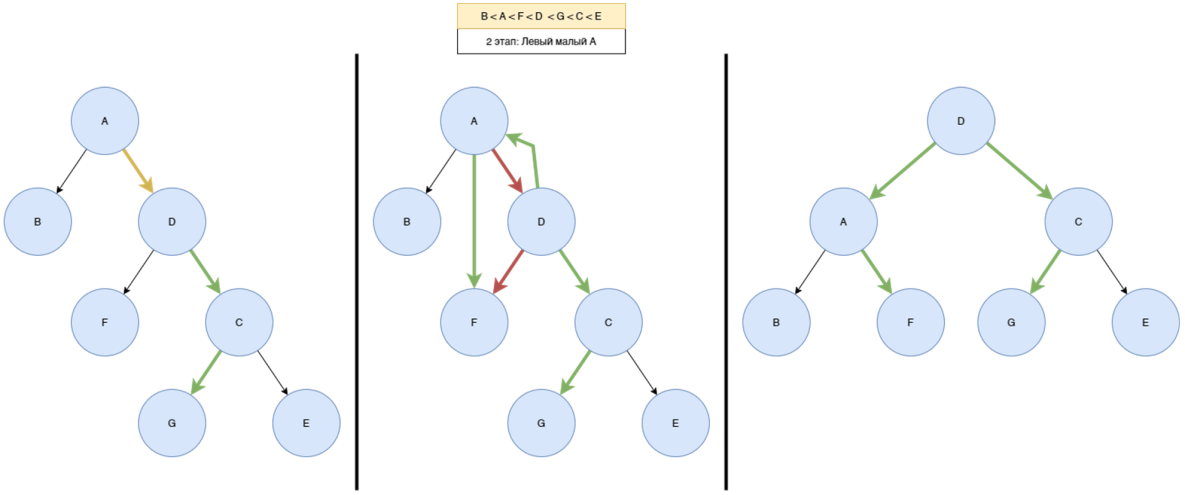

Рисунок 5б – Правый-левый поворот, 2 этап

Основные операции и их сложности:

- Поиск:

Поиск в AVL-дереве аналогичен поиску в обычном BST, но благодаря балансировке высота дерева всегда составляет $O(log_2 N)$. Таким образом, сложность поиска всегда $O(log_2 N)$.

- Вставка:

Вставка выполняется так же, как в BST, но после добавления нового узла проверяется баланс и, при необходимости, выполняются повороты. Повороты

занимают $O(1)$, а высота дерева $O(log_2 N)$, поэтому общая сложность вставки — $O(log_2 N)$.

- Удаление:

Удаление также требует проверки баланса и выполнения поворотов, если фактор баланса нарушен. Сложность удаления — $O(log_2 N)$.

Рандомизированное дерево поиска — это двоичное дерево поиска, которое использует случайность для поддержания баланса. Оно было описано в статье на Habr как структура данных, полагающаяся на вероятностные методы вместо строгих правил балансировки, как в AVL-дереве. Главное отличие рандомизированного дерева от обычного BST заключается в том, что оно поддерживает баланс в среднем случае за счет случайного выбора структуры при вставке узлов.

Как работает балансировка? В рандомизированном дереве баланс достигается с помощью операций разбиения (split) и слияния (merge), которые выполняются с использованием случайности. Процесс вставки нового узла выглядит следующим образом:

1. Обычная вставка: Новый узел добавляется как в обычное BST, то есть он занимает место в зависимости от своего ключа.
2. Рандомизация: С вероятностью, зависящей от размера дерева (например, $\frac{1}{N+1}$, где N — текущее число узлов), новый узел становится корнем. Для этого:
    - Дерево разбивается на два поддерева: одно с ключами меньше нового узла, другое — с ключами больше.
    - Затем выполняется слияние: новый узел становится корнем, а два поддерева прикрепляются как левое и правое поддеревья.

Эти операции гарантируют, что в среднем случае дерево остается сбалансированным, а его высота составляет $O(log_2 N)$.

Основные операции и их сложности:

- Поиск:

Поиск в рандомизированном дереве аналогичен поиску в обычном BST. В среднем случае высота дерева составляет $O(log_2 N)$, поэтому сложность поиска — $O(log_2 N)$.

- Вставка:

Вставка включает добавление узла как в BST и возможное выполнение операций разбиения и слияния с учетом случайности. В среднем случае сложность вставки — $O(log_2 N)$.

- Удаление:

Удаление реализуется через разбиение дерева на поддеревья и слияние оставшихся частей. Сложность удаления в среднем — $O(log_2 N)$.

Примечание: В худшем случае (при крайне неудачных случайных выборах) высота дерева может достигать $O(N)$, но вероятность этого мала, и в среднем дерево обеспечивает логарифмическую производительность.

Красно-черное дерево — это самобалансирующееся двоичное дерево поиска, которое использует цвета узлов (красный или черный) для поддержания баланса. Главное отличие красно-черного дерева от обычного BST заключается в том, что оно гарантирует приблизительный баланс высоты с помощью набора правил, обеспечивая, что высота дерева не превышает $2 \cdot log_2 N$, где N — число узлов.

Как работает балансировка? Баланс в красно-черном дереве поддерживается через следующие правила:

1. Корень всегда черный.
2. Все листья (NULL-узлы) черные.

3. У красного узла не может быть красного родителя (два красных узла не могут быть соседними).
4. Число черных узлов (черная высота) на всех путях от корня до листьев одинаково.

После вставки или удаления узла дерево проверяет, не нарушены ли эти правила. Если нарушение происходит, выполняются:

1. Перекраска узлов: изменение цвета узлов (с красного на черный или наоборот) для восстановления свойств.
2. Повороты:
    a. Простой левый поворот: используется, когда нужно переместить узлы влево (рис. 2).
    b. Простой правый поворот: используется, когда нужно переместить узлы вправо (рис. 3).

Эти операции восстанавливают правила, обеспечивая высоту $O(log_2 N)$.

Рассмотрим подробнее выполнение перекраски узлов. Перекраска — это первый шаг, чтобы их восстановить. Она меняет цвета узлов, не трогая их расположение, то есть не требует сложных перемещений.

Как перекраска работает при вставке узла? Новый узел всегда добавляется красным. Если его родитель тоже красный, это проблема — два красных узла подряд запрещены. Тогда дерево смотрит на брата родителя (то есть на «дядю» нового узла):

- Если дядя красный, можно перекрасить родителя и дядю в черный, а их общего родителя (дедушку) — в красный. Это сохраняет баланс, но иногда создает новую проблему выше по дереву (например, если дедушка стал красным рядом с красным родителем). Тогда перекраска продолжается вверх. Данный случай представлен на рисунке 6.
- Если дядя черный или его вообще нет, перекраска не решает проблему, и дерево использует более сложные действия — повороты, чтобы все исправить.

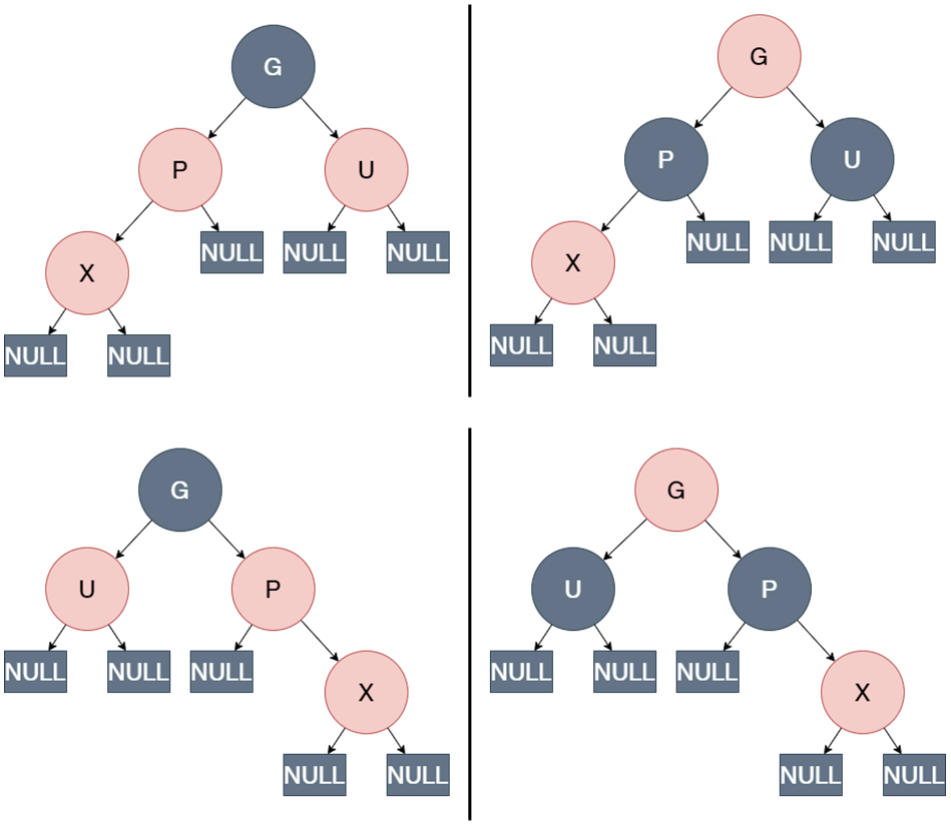

Рисунок 6 – Перекраска в случае красного дяди

Как перекраска работает при удалении узла? Если мы убираем черный узел, то на одном из путей становится меньше черных узлов, чем нужно. Это нарушает правило о равном количестве черных узлов. Тогда дерево может перекрасить, например, брата удаленного узла или его соседей:

- Скажем, брат красный. Его можно сделать черным, а родителя — красным, чтобы выровнять баланс. Иногда после этого нужен поворот, чтобы все стало на свои места.

Перекраска помогает исправить нарушения правил простым изменением цветов, не меняя структуру дерева. Если этого недостаточно, дерево прибегает к поворотам — более сложным операциям, которые перестраивают связи между узлами. Но перекраска всегда пробуется первой, потому что она проще и быстрее.

Основные операции и их сложности

- Поиск:

Поиск в красно-черном дереве аналогичен поиску в BST. Поскольку высота дерева не превышает $2 \cdot log_2 N$, сложность поиска — $O(log_2 N)$.

- Вставка:

Вставка выполняется как в BST, после чего проверяются и восстанавливаются правила с помощью перекрасок и поворотов. Сложность вставки — $O(log_2 N)$.

- Удаление:

Удаление требует восстановления свойств дерева через перекраски и повороты. Сложность удаления — $O(log_2 N)$.

Балансировка при вставке. Когда мы вставляем новый узел в красно-черное дерево, он всегда добавляется как красный. Это может нарушить правило, что два красных узла не могут быть рядом (красный родитель и красный ребенок). Чтобы исправить это, используются перекраска (изменение цвета узлов) и повороты (изменение структуры дерева). Вот основные операции в пунктах:

1. Добавление узла;
2. Новый узел X вставляется как красный;
3. Если его родитель P тоже красный, возникает нарушение (два красных узла подряд);
4. Если родитель черный, балансировка не нужна — дерево остается корректным;
5. Проверка дяди;
6. Дядя U — это брат родителя P, то есть другой ребенок дедушки G:

    Случай 1: Дядя красный;

    a. Перекрашиваем: родитель P и дядя U становятся черными, дедушка G — красным;
    b. Проверяем выше: если дедушка G теперь рядом с красным родителем, повторяем процесс для G;

Случай 2: Дядя черный или отсутствует;

c. Переходим к поворотам, чтобы выровнять структуру;

7. Повороты для случая с черным дядей:
   a. Если X и P на разных сторонах относительно G (например, P — левый ребенок, X — правый):
      i. Делаем первый поворот вокруг P, чтобы выровнять X и P на одной стороне;
      ii. Пример: рисунок 7, левая часть — начальное состояние с черным дядей и красными узлами на разных сторонах; средняя часть — промежуточный поворот;
   b. Если X и P на одной стороне (например, оба слева):
      i. Делаем поворот вокруг G, поднимая P вверх, а G опуская вниз;
      ii. Перекрашиваем: P становится черным, G — красным;
      iii. Пример: рисунок 8, последовательность от левой части (начало) до правой (финал с поворотом и перекраской);
8. Финальная проверка:
   a. После всех операций корень дерева всегда делаем черным, чтобы соблюсти правило красно-черного дерева.

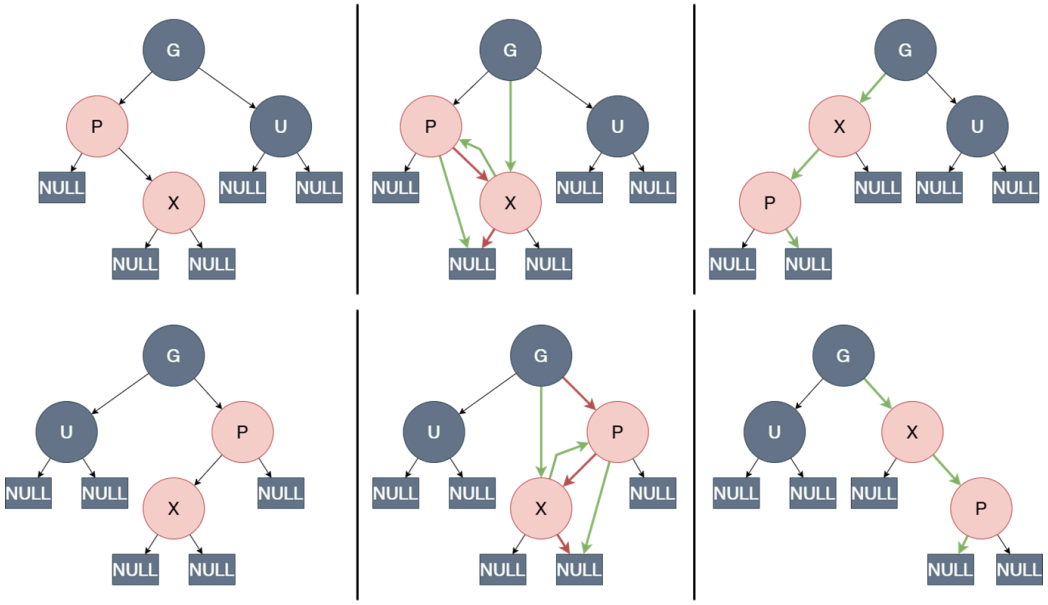

Рисунок 7 – Черный дядя, красные с разных сторон

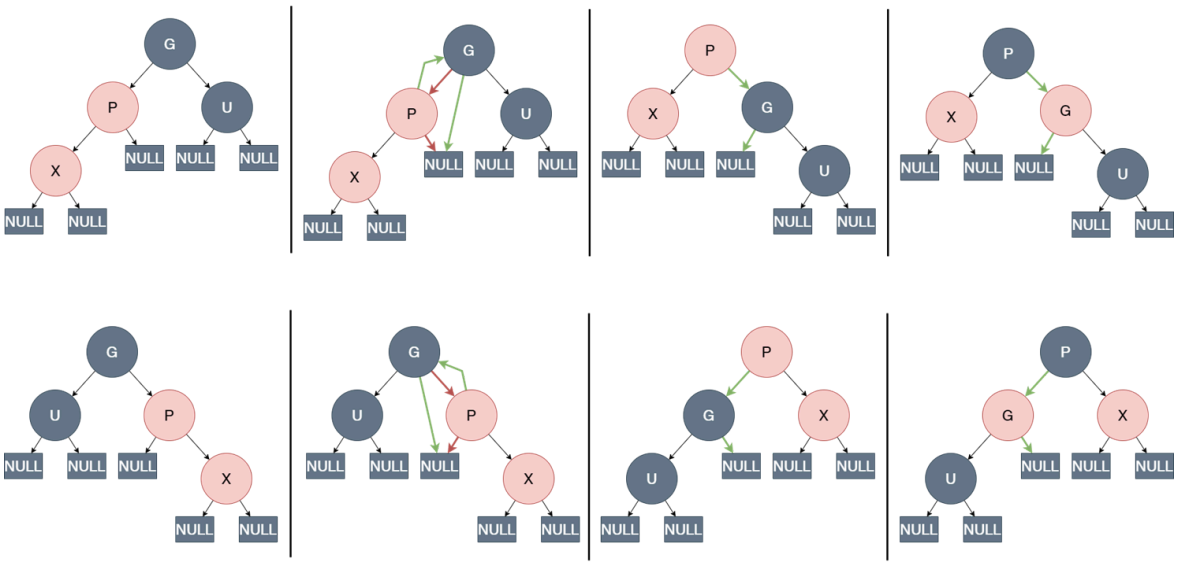

Рисунок 8 – Черный дядя, красные с одной стороны

Описание выполнения задачи

Для реализации задачи была написана программа на C++:

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <chrono>
#include <vector>
#include <random>
#include <numeric>
#include <map> // Добавляем для подсчета частот

using namespace std;

// Генератор случайных чисел для использования в тестах
random_device rd;
mt19937 gen(rd());

// Класс AVLTree — реализация сбалансированного AVL-дерева
class AVLTree {
public:
    struct Node {
        int key;
        Node* left;
        Node* right;
        int height;
        Node(int key) : key(key), left(nullptr), right(nullptr), height(1) {}
    };

    AVLTree() : root(nullptr), size(0) {}
        AVLTree(const AVLTree& other) : root(nullptr), size(0) { root = copy(other.root); size = other.size; }
    ~AVLTree() { destroy(root); }

    void insert(int key) {
```

```
        if (search(key)) return;
        root = insert(root, key);
        ++size;
    }

    void remove(int key) {
        Node* newRoot = remove(root, key);
        if (newRoot != root || root == nullptr) --size;
        root = newRoot;
    }

    bool search(int key) { return search(root, key); }
    int getMaxDepth() { return getMaxDepth(root); }
    int getSize() { return size; }
    vector<int> getLeafDepths() {
        vector<int> leafDepths;
        collectLeafDepths(root, 1, leafDepths);
        return leafDepths;
    }

    template <typename Func>
    double measureTime(Func func, int operations, const vector<int>& keys) {
        AVLTree tempTree(*this);
        for (int i = 0; i < 1000; ++i) func(tempTree, keys[i % keys.size()]);
        auto start = chrono::steady_clock::now();
        for (int i = 0; i < operations; ++i) func(*this, keys[i % keys.size()]);
        auto end = chrono::steady_clock::now();
        return chrono::duration<double>(end - start).count() / operations;
    }

private:
    Node* root;
    int size;

    Node* copy(Node* node) {
```

```
    if (!node) return nullptr;
    Node* newNode = new Node(node->key);
    newNode->height = node->height;
    newNode->left = copy(node->left);
    newNode->right = copy(node->right);
    return newNode;
}

void destroy(Node* node) {
    if (!node) return;
    destroy(node->left);
    destroy(node->right);
    delete node;
}

Node* insert(Node* node, int key) {
    if (!node) return new Node(key);
    if (key < node->key) node->left = insert(node->left, key);
    else if (key > node->key) node->right = insert(node->right, key);
    else return node;

    node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    int balance = getBalance(node);

    if (balance > 1 && key < node->left->key) return rightRotate(node);
    if (balance < -1 && key > node->right->key) return leftRotate(node);
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
```

```
}

Node* remove(Node* root, int key) {
    if (!root) return root;
    if (key < root->key) root->left = remove(root->left, key);
    else if (key > root->key) root->right = remove(root->right, key);
    else {
        if (!root->left) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (!root->right) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* temp = getMinValueNode(root->right);
        root->key = temp->key;
        root->right = remove(root->right, temp->key);
    }

    root->height = 1 + max(getHeight(root->left), getHeight(root->right));
    int balance = getBalance(root);

    if (balance > 1 && getBalance(root->left) >= 0) return rightRotate(root);
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    if (balance < -1 && getBalance(root->right) <= 0) return leftRotate(root);
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
```

```
    }

    bool search(Node* node, int key) {
        if (!node) return false;
        if (key == node->key) return true;
        return key < node->key ? search(node->left, key) : search(node->right, key);
    }

    Node* rightRotate(Node* y) {
        Node* x = y->left;
        Node* T2 = x->right;
        x->right = y;
        y->left = T2;
        y->height = 1 + max(getHeight(y->left), getHeight(y->right));
        x->height = 1 + max(getHeight(x->left), getHeight(x->right));
        return x;
    }

    Node* leftRotate(Node* x) {
        Node* y = x->right;
        Node* T2 = y->left;
        y->left = x;
        x->right = T2;
        x->height = 1 + max(getHeight(x->left), getHeight(x->right));
        y->height = 1 + max(getHeight(y->left), getHeight(y->right));
        return y;
    }

    int getHeight(Node* node) { return node ? node->height : 0; }
     int getBalance(Node* node) { return node ? getHeight(node->left) - getHeight(node->right) : 0;
}

    Node* getMinValueNode(Node* node) {
        Node* current = node;
        while (current && current->left) current = current->left;
```

```
        return current;
    }

    int getMaxDepth(Node* node) {
        if (!node) return 0;
        return 1 + max(getMaxDepth(node->left), getMaxDepth(node->right));
    }

    void collectLeafDepths(Node* node, int currentDepth, vector<int>& leafDepths) {
        if (!node) return;
        if (!node->left && !node->right) leafDepths.push_back(currentDepth);
        collectLeafDepths(node->left, currentDepth + 1, leafDepths);
        collectLeafDepths(node->right, currentDepth + 1, leafDepths);
    }
};

// Класс RandomizedBST — реализация рандомизированного бинарного дерева поиска
class RandomizedBST {
public:
    struct Node {
        int key;
        Node* left;
        Node* right;
        int size;
        Node(int key) : key(key), left(nullptr), right(nullptr), size(1) {}
    };

    RandomizedBST() : root(nullptr), size(0) {}
        RandomizedBST(const RandomizedBST& other) : root(nullptr), size(0) { root = copy(other.root); size = other.size; }
    ~RandomizedBST() { destroy(root); }

    void insert(int key) {
        if (search(key)) return;
        root = insert(root, key);
```

```
        ++size;
    }

    void remove(int key) {
        Node* newRoot = remove(root, key);
        if (newRoot != root || root == nullptr) --size;
        root = newRoot;
    }

    bool search(int key) { return search(root, key); }
    int getMaxDepth() { return getMaxDepth(root); }
    int getSize() { return size; }
    vector<int> getLeafDepths() {
        vector<int> leafDepths;
        collectLeafDepths(root, 1, leafDepths);
        return leafDepths;
    }

    template <typename Func>
    double measureTime(Func func, int operations, const vector<int>& keys) {
        RandomizedBST tempTree(*this);
        for (int i = 0; i < 1000; ++i) func(tempTree, keys[i % keys.size()]);
        auto start = chrono::steady_clock::now();
        for (int i = 0; i < operations; ++i) func(*this, keys[i % keys.size()]);
        auto end = chrono::steady_clock::now();
        return chrono::duration<double>(end - start).count() / operations;
    }

private:
    Node* root;
    int size;

    Node* copy(Node* node) {
        if (!node) return nullptr;
        Node* newNode = new Node(node->key);
```

```
    newNode->size = node->size;
    newNode->left = copy(node->left);
    newNode->right = copy(node->right);
    return newNode;
}

void destroy(Node* node) {
    if (!node) return;
    destroy(node->left);
    destroy(node->right);
    delete node;
}

int getSize(Node* node) { return node ? node->size : 0; }
void updateSize(Node* node) {
    if (node) node->size = getSize(node->left) + getSize(node->right) + 1;
}

Node* merge(Node* left, Node* right) {
    if (!left) return right;
    if (!right) return left;
    int totalSize = getSize(left) + getSize(right);
    uniform_int_distribution<int> dist(0, totalSize - 1);
    if (dist(gen) < getSize(left)) {
        left->right = merge(left->right, right);
        updateSize(left);
        return left;
    } else {
        right->left = merge(left, right->left);
        updateSize(right);
        return right;
    }
}

pair<Node*, Node*> split(Node* node, int key) {
```

```
    if (!node) return {nullptr, nullptr};
    if (node->key < key) {
        auto [left, right] = split(node->right, key);
        node->right = left;
        updateSize(node);
        return {node, right};
    } else {
        auto [left, right] = split(node->left, key);
        node->left = right;
        updateSize(node);
        return {left, node};
    }
}

Node* insert(Node* node, int key) {
    if (!node) return new Node(key);
    int totalSize = getSize(node) + 1;
    uniform_int_distribution<int> dist(0, totalSize - 1);
    if (dist(gen) == 0) {
        Node* newNode = new Node(key);
        auto [left, right] = split(node, key);
        newNode->left = left;
        newNode->right = right;
        updateSize(newNode);
        return newNode;
    }
    if (key < node->key) node->left = insert(node->left, key);
    else node->right = insert(node->right, key);
    updateSize(node);
    return node;
}

Node* remove(Node* node, int key) {
    if (!node) return nullptr;
    if (node->key == key) {
```

```
            Node* result = merge(node->left, node->right);
            delete node;
            return result;
        }
        if (key < node->key) node->left = remove(node->left, key);
        else node->right = remove(node->right, key);
        updateSize(node);
        return node;
    }

    bool search(Node* node, int key) {
        if (!node) return false;
        if (key == node->key) return true;
        return key < node->key ? search(node->left, key) : search(node->right, key);
    }

    int getMaxDepth(Node* node) {
        if (!node) return 0;
        return 1 + max(getMaxDepth(node->left), getMaxDepth(node->right));
    }

    void collectLeafDepths(Node* node, int currentDepth, vector<int>& leafDepths) {
        if (!node) return;
        if (!node->left && !node->right) leafDepths.push_back(currentDepth);
        collectLeafDepths(node->left, currentDepth + 1, leafDepths);
        collectLeafDepths(node->right, currentDepth + 1, leafDepths);
    }
};

// Класс RedBlackTree — реализация красно-черного дерева
class RedBlackTree {
public:
    enum Color { RED, BLACK };

    struct Node {
```

```
        int key;
        Color color;
        Node* left;
        Node* right;
        Node* parent;
        Node(int key) : key(key), color(RED), left(nullptr), right(nullptr), parent(nullptr) {}
    };

    RedBlackTree() : root(nullptr), size(0) {}
     RedBlackTree(const RedBlackTree& other) : root(nullptr), size(0) { root = copy(other.root,
nullptr); size = other.size; }
    ~RedBlackTree() { destroy(root); }

    void insert(int key) {
        Node* node = new Node(key);
        root = insert(root, node);
        fixInsert(node);
        ++size;
    }

    void remove(int key) {
        Node* node = findNode(root, key);
        if (node) {
            remove(node);
            --size;
        }
    }

    bool search(int key) { return findNode(root, key) != nullptr; }
    int getMaxDepth() { return getMaxDepth(root); }
    int getSize() { return size; }
    vector<int> getLeafDepths() {
        vector<int> leafDepths;
        collectLeafDepths(root, 1, leafDepths);
        return leafDepths;
```

```
    }

    template <typename Func>
    double measureTime(Func func, int operations, const vector<int>& keys) {
        RedBlackTree tempTree(*this);
        for (int i = 0; i < 1000; ++i) func(tempTree, keys[i % keys.size()]);
        auto start = chrono::steady_clock::now();
        for (int i = 0; i < operations; ++i) func(*this, keys[i % keys.size()]);
        auto end = chrono::steady_clock::now();
        return chrono::duration<double>(end - start).count() / operations;
    }

private:
    Node* root;
    int size;

    Node* copy(Node* node, Node* parent) {
        if (!node) return nullptr;
        Node* newNode = new Node(node->key);
        newNode->color = node->color;
        newNode->parent = parent;
        newNode->left = copy(node->left, newNode);
        newNode->right = copy(node->right, newNode);
        return newNode;
    }

    void destroy(Node* node) {
        if (!node) return;
        destroy(node->left);
        destroy(node->right);
        delete node;
    }

    Node* insert(Node* root, Node* node) {
        if (!root) return node;
```

```
        if (node->key < root->key) {
            root->left = insert(root->left, node);
            if (root->left) root->left->parent = root;
        } else if (node->key > root->key) {
            root->right = insert(root->right, node);
            if (root->right) root->right->parent = root;
        } else {
            delete node;
            return root;
        }
        return root;
    }

    void fixInsert(Node* node) {
        if (!node) return;
        Node* parent = nullptr;
        Node* grandparent = nullptr;
        while (node != root && node->color == RED && node->parent && node->parent->color ==
RED) {
            parent = node->parent;
            grandparent = parent->parent;
            if (parent == grandparent->left) {
                Node* uncle = grandparent->right;
                if (uncle && uncle->color == RED) {
                    grandparent->color = RED;
                    parent->color = BLACK;
                    uncle->color = BLACK;
                    node = grandparent;
                } else {
                    if (node == parent->right) {
                        rotateLeft(parent);
                        node = parent;
                        parent = node->parent;
                    }
                    rotateRight(grandparent);
```

```
            swap(parent->color, grandparent->color);
            node = parent;
        }
    } else {
        Node* uncle = grandparent->left;
        if (uncle && uncle->color == RED) {
            grandparent->color = RED;
            parent->color = BLACK;
            uncle->color = BLACK;
            node = grandparent;
        } else {
            if (node == parent->left) {
                rotateRight(parent);
                node = parent;
                parent = node->parent;
            }
            rotateLeft(grandparent);
            swap(parent->color, grandparent->color);
            node = parent;
        }
    }
    if (root) root->color = BLACK;
}

Node* findNode(Node* node, int key) {
    if (!node) return nullptr;
    if (key == node->key) return node;
    return key < node->key ? findNode(node->left, key) : findNode(node->right, key);
}

void remove(Node* node) {
    if (!node) return;
    Node* child = nullptr;
    Node* parent = node->parent;
```

```
    Color color = node->color;
    if (node->left && node->right) {
        Node* replace = getMinValueNode(node->right);
        if (!replace) return;
        node->key = replace->key;
        node = replace;
        parent = node->parent;
        color = node->color;
    }
    if (node->left) child = node->left;
    else child = node->right;
    if (child) child->parent = parent;
    if (!parent) root = child;
    else if (node == parent->left) parent->left = child;
    else parent->right = child;
    if (color == BLACK) fixRemove(child, parent);
    delete node;
}

void fixRemove(Node* node, Node* parent) {
    if (!parent) return;
    Node* sibling;
    while (node != root && (!node || node->color == BLACK)) {
        if (node == parent->left) {
            sibling = parent->right;
            if (sibling && sibling->color == RED) {
                sibling->color = BLACK;
                parent->color = RED;
                rotateLeft(parent);
                sibling = parent->right;
            }
            if ((!sibling || !sibling->left || sibling->left->color == BLACK) &&
                (!sibling || !sibling->right || sibling->right->color == BLACK)) {
                if (sibling) sibling->color = RED;
                node = parent;
```

```
            parent = node->parent;
        } else {
            if (!sibling || !sibling->right || sibling->right->color == BLACK) {
                if (sibling && sibling->left) sibling->left->color = BLACK;
                if (sibling) sibling->color = RED;
                if (sibling) rotateRight(sibling);
                sibling = parent->right;
            }
            if (sibling) sibling->color = parent->color;
            parent->color = BLACK;
            if (sibling && sibling->right) sibling->right->color = BLACK;
            rotateLeft(parent);
            node = root;
        }
    } else {
        sibling = parent->left;
        if (sibling && sibling->color == RED) {
            sibling->color = BLACK;
            parent->color = RED;
            rotateRight(parent);
            sibling = parent->left;
        }
        if ((!sibling || !sibling->left || sibling->left->color == BLACK) &&
            (!sibling || !sibling->right || sibling->right->color == BLACK)) {
            if (sibling) sibling->color = RED;
            node = parent;
            parent = node->parent;
        } else {
            if (!sibling || !sibling->left || sibling->left->color == BLACK) {
                if (sibling && sibling->right) sibling->right->color = BLACK;
                if (sibling) sibling->color = RED;
                if (sibling) rotateLeft(sibling);
                sibling = parent->left;
            }
            if (sibling) sibling->color = parent->color;
```

```
            parent->color = BLACK;
            if (sibling && sibling->left) sibling->left->color = BLACK;
            rotateRight(parent);
            node = root;
        }
    }
  }
  if (node) node->color = BLACK;
}

void rotateLeft(Node* node) {
  if (!node || !node->right) return;
  Node* right = node->right;
  node->right = right->left;
  if (right->left) right->left->parent = node;
  right->parent = node->parent;
  if (!node->parent) root = right;
  else if (node == node->parent->left) node->parent->left = right;
  else node->parent->right = right;
  right->left = node;
  node->parent = right;
}

void rotateRight(Node* node) {
  if (!node || !node->left) return;
  Node* left = node->left;
  node->left = left->right;
  if (left->right) left->right->parent = node;
  left->parent = node->parent;
  if (!node->parent) root = left;
  else if (node == node->parent->right) node->parent->right = left;
  else node->parent->left = left;
  left->right = node;
  node->parent = left;
}
```

```
    Node* getMinValueNode(Node* node) {
        if (!node) return nullptr;
        Node* current = node;
        while (current && current->left) current = current->left;
        return current;
    }

    int getMaxDepth(Node* node) {
        if (!node) return 0;
        return 1 + max(getMaxDepth(node->left), getMaxDepth(node->right));
    }

    void collectLeafDepths(Node* node, int currentDepth, vector<int>& leafDepths) {
        if (!node) return;
        if (!node->left && !node->right) leafDepths.push_back(currentDepth);
        collectLeafDepths(node->left, currentDepth + 1, leafDepths);
        collectLeafDepths(node->right, currentDepth + 1, leafDepths);
    }
};

// Функция для записи гистограммы в файл
void writeHistogramToFile(const string& filename, const map<int, int>& histogram, const
string& header) {
    ofstream file(filename);
    if (!file.is_open()) {
        cerr << "Failed to open " << filename << endl;
        return;
    }
    file << header << "\n";
    for (const auto& [value, count] : histogram) {
        file << value << "," << count << "\n";
    }
    file.close();
}
```

```
int main() {
    ofstream outFile("results_random.csv");
    if (!outFile.is_open()) {
        cerr << "Failed to open results_random.csv" << endl;
        return 1;
    }
    outFile << "N,AVL Max Depth,AVL Avg Insert Time,AVL Avg Remove Time,AVL Avg Search Time,"
            << "RBST Max Depth,RBST Avg Insert Time,RBST Avg Remove Time,RBST Avg Search Time,"
            << "RB Max Depth,RB Avg Insert Time,RB Avg Remove Time,RB Avg Search Time\n";

    // Векторы для максимальных высот
    vector<int> avlMaxHeightsLastSeries, rbstMaxHeightsLastSeries, rbMaxHeightsLastSeries;
    // Векторы для всех глубин листьев за все итерации
    vector<int> avlLeafDepthsLastSeries, rbstLeafDepthsLastSeries, rbLeafDepthsLastSeries;

    for (int i = 10; i <= 18; ++i) {
        int N = 1 << i;
        cout << "Running tests for N = " << N << " (Random Data)" << endl;

        double totalInsertAVL = 0, totalRemoveAVL = 0, totalSearchAVL = 0;
        double totalInsertRBST = 0, totalRemoveRBST = 0, totalSearchRBST = 0;
        double totalInsertRB = 0, totalRemoveRB = 0, totalSearchRB = 0;
        int maxDepthAVL = 0, maxDepthRBST = 0, maxDepthRB = 0;

        vector<int> testKeys(N);
        for (int j = 0; j < N; ++j) testKeys[j] = j;
        shuffle(testKeys.begin(), testKeys.end(), gen);

        for (int repeat = 0; repeat < 50; ++repeat) {
            cout << "Repeat " << repeat + 1 << " of 50 for N = " << N << endl;

            vector<int> values(N);
```

```
        for (int j = 0; j < N; ++j) values[j] = j;
        shuffle(values.begin(), values.end(), gen);

        AVLTree avl_tree;
        for (int v : values) avl_tree.insert(v);
          totalInsertAVL += avl_tree.measureTime([](AVLTree& t, int val) { t.insert(val); }, 1000,
testKeys);
            totalRemoveAVL += avl_tree.measureTime([](AVLTree& t, int val) { t.remove(val); },
1000, testKeys);
         totalSearchAVL += avl_tree.measureTime([](AVLTree& t, int val) { t.search(val); }, 1000,
testKeys);
        maxDepthAVL = max(maxDepthAVL, avl_tree.getMaxDepth());

        RandomizedBST rbst;
        for (int v : values) rbst.insert(v);
          totalInsertRBST += rbst.measureTime([](RandomizedBST& t, int val) { t.insert(val); },
1000, testKeys);
           totalRemoveRBST += rbst.measureTime([](RandomizedBST& t, int val) { t.remove(val);
}, 1000, testKeys);
           totalSearchRBST += rbst.measureTime([](RandomizedBST& t, int val) { t.search(val); },
1000, testKeys);
        maxDepthRBST = max(maxDepthRBST, rbst.getMaxDepth());

        RedBlackTree rb_tree;
        for (int v : values) rb_tree.insert(v);
            totalInsertRB += rb_tree.measureTime([](RedBlackTree& t, int val) { t.insert(val); },
1000, testKeys);
           totalRemoveRB += rb_tree.measureTime([](RedBlackTree& t, int val) { t.remove(val); },
1000, testKeys);
            totalSearchRB += rb_tree.measureTime([](RedBlackTree& t, int val) { t.search(val); },
1000, testKeys);
        maxDepthRB = max(maxDepthRB, rb_tree.getMaxDepth());

        if (i == 18) {
```

```
                cout << "AVL size: " << avl_tree.getSize() << ", leaf count: " << avl_tree.getLeafDepths().size() << endl;
                cout << "RBST size: " << rbst.getSize() << ", leaf count: " << rbst.getLeafDepths().size() << endl;
                cout << "RB size: " << rb_tree.getSize() << ", leaf count: " << rb_tree.getLeafDepths().size() << endl;

            avlMaxHeightsLastSeries.push_back(avl_tree.getMaxDepth());
            rbstMaxHeightsLastSeries.push_back(rbst.getMaxDepth());
            rbMaxHeightsLastSeries.push_back(rb_tree.getMaxDepth());

            vector<int> leafDepthsAVL = avl_tree.getLeafDepths();
            avlLeafDepthsLastSeries.insert(avlLeafDepthsLastSeries.end(), leafDepthsAVL.begin(), leafDepthsAVL.end());

            vector<int> leafDepthsRBST = rbst.getLeafDepths();
                        rbstLeafDepthsLastSeries.insert(rbstLeafDepthsLastSeries.end(), leafDepthsRBST.begin(), leafDepthsRBST.end());

            vector<int> leafDepthsRB = rb_tree.getLeafDepths();
              rbLeafDepthsLastSeries.insert(rbLeafDepthsLastSeries.end(), leafDepthsRB.begin(), leafDepthsRB.end());
        }
    }

        double avgInsertAVL = totalInsertAVL / 50, avgRemoveAVL = totalRemoveAVL / 50, avgSearchAVL = totalSearchAVL / 50;
      double avgInsertRBST = totalInsertRBST / 50, avgRemoveRBST = totalRemoveRBST / 50, avgSearchRBST = totalSearchRBST / 50;
          double avgInsertRB = totalInsertRB / 50, avgRemoveRB = totalRemoveRB / 50, avgSearchRB = totalSearchRB / 50;

      outFile << N << "," << maxDepthAVL << "," << avgInsertAVL << "," << avgRemoveAVL << "," << avgSearchAVL << ","
```

```
            << maxDepthRBST << "," << avgInsertRBST << "," << avgRemoveRBST << "," <<
avgSearchRBST << ","
                << maxDepthRB << "," << avgInsertRB << "," << avgRemoveRB << "," <<
avgSearchRB << "\n";
    }
    outFile.close();

    // Подсчет частот максимальных высот
    map<int, int> avlMaxHeightHist, rbstMaxHeightHist, rbMaxHeightHist;
    for (int height : avlMaxHeightsLastSeries) avlMaxHeightHist[height]++;
    for (int height : rbstMaxHeightsLastSeries) rbstMaxHeightHist[height]++;
    for (int height : rbMaxHeightsLastSeries) rbMaxHeightHist[height]++;

    // Запись гистограммы максимальных высот
        writeHistogramToFile("AVL_max_heights_random.csv", avlMaxHeightHist, "AVL
Height,Frequency");
        writeHistogramToFile("RBST_max_heights_random.csv", rbstMaxHeightHist, "RBST
Height,Frequency");
        writeHistogramToFile("RB_max_heights_random.csv", rbMaxHeightHist, "RB
Height,Frequency");

    // Подсчет частот высот веток
    map<int, int> avlLeafDepthHist, rbstLeafDepthHist, rbLeafDepthHist;
    for (int depth : avlLeafDepthsLastSeries) avlLeafDepthHist[depth]++;
    for (int depth : rbstLeafDepthsLastSeries) rbstLeafDepthHist[depth]++;
    for (int depth : rbLeafDepthsLastSeries) rbLeafDepthHist[depth]++;

    // Запись гистограммы высот веток
        writeHistogramToFile("AVL_leaf_depths_random.csv", avlLeafDepthHist, "AVL
Depth,Frequency");
        writeHistogramToFile("RBST_leaf_depths_random.csv", rbstLeafDepthHist, "RBST
Depth,Frequency");
        writeHistogramToFile("RB_leaf_depths_random.csv", rbLeafDepthHist, "RB
Depth,Frequency");
```

```
    // Для отсортированных данных
    ofstream outFileSorted("results_sorted.csv");
    if (!outFileSorted.is_open()) {
        cerr << "Failed to open results_sorted.csv" << endl;
        return 1;
    }
     outFileSorted << "N,AVL Max Depth,AVL Avg Insert Time,AVL Avg Remove Time,AVL Avg Search Time,"
                   << "RBST Max Depth,RBST Avg Insert Time,RBST Avg Remove Time,RBST Avg Search Time,"
                      << "RB Max Depth,RB Avg Insert Time,RB Avg Remove Time,RB Avg Search Time\n";

                   vector<int> avlMaxHeightsLastSeriesSorted, rbstMaxHeightsLastSeriesSorted, rbMaxHeightsLastSeriesSorted;
                    vector<int> avlLeafDepthsLastSeriesSorted, rbstLeafDepthsLastSeriesSorted, rbLeafDepthsLastSeriesSorted;

    for (int i = 10; i <= 18; ++i) {
        int N = 1 << i;
        cout << "Running tests for N = " << N << " (Sorted Data)" << endl;

        double totalInsertAVL = 0, totalRemoveAVL = 0, totalSearchAVL = 0;
        double totalInsertRBST = 0, totalRemoveRBST = 0, totalSearchRBST = 0;
        double totalInsertRB = 0, totalRemoveRB = 0, totalSearchRB = 0;
        int maxDepthAVL = 0, maxDepthRBST = 0, maxDepthRB = 0;

        vector<int> testKeys(N);
        for (int j = 0; j < N; ++j) testKeys[j] = j;
        shuffle(testKeys.begin(), testKeys.end(), gen);

        for (int repeat = 0; repeat < 50; ++repeat) {
            cout << "Repeat " << repeat + 1 << " of 50 for N = " << N << endl;

            vector<int> values(N);
```

```
        for (int j = 0; j < N; ++j) values[j] = j;

        AVLTree avl_tree;
        for (int v : values) avl_tree.insert(v);
          totalInsertAVL += avl_tree.measureTime([](AVLTree& t, int val) { t.insert(val); }, 1000,
testKeys);
            totalRemoveAVL += avl_tree.measureTime([](AVLTree& t, int val) { t.remove(val); },
1000, testKeys);
         totalSearchAVL += avl_tree.measureTime([](AVLTree& t, int val) { t.search(val); }, 1000,
testKeys);
        maxDepthAVL = max(maxDepthAVL, avl_tree.getMaxDepth());

        RandomizedBST rbst;
        for (int v : values) rbst.insert(v);
          totalInsertRBST += rbst.measureTime([](RandomizedBST& t, int val) { t.insert(val); },
1000, testKeys);
          totalRemoveRBST += rbst.measureTime([](RandomizedBST& t, int val) { t.remove(val);
}, 1000, testKeys);
          totalSearchRBST += rbst.measureTime([](RandomizedBST& t, int val) { t.search(val); },
1000, testKeys);
        maxDepthRBST = max(maxDepthRBST, rbst.getMaxDepth());

        RedBlackTree rb_tree;
        for (int v : values) rb_tree.insert(v);
            totalInsertRB += rb_tree.measureTime([](RedBlackTree& t, int val) { t.insert(val); },
1000, testKeys);
          totalRemoveRB += rb_tree.measureTime([](RedBlackTree& t, int val) { t.remove(val); },
1000, testKeys);
            totalSearchRB += rb_tree.measureTime([](RedBlackTree& t, int val) { t.search(val); },
1000, testKeys);
        maxDepthRB = max(maxDepthRB, rb_tree.getMaxDepth());

        if (i == 18) {
                        cout << "AVL size: " << avl_tree.getSize() << ", leaf count: " <<
avl_tree.getLeafDepths().size() << endl;
```

```
                cout << "RBST size: " << rbst.getSize() << ", leaf count: " << rbst.getLeafDepths().size() << endl;
                cout << "RB size: " << rb_tree.getSize() << ", leaf count: " << rb_tree.getLeafDepths().size() << endl;

            avlMaxHeightsLastSeriesSorted.push_back(avl_tree.getMaxDepth());
            rbstMaxHeightsLastSeriesSorted.push_back(rbst.getMaxDepth());
            rbMaxHeightsLastSeriesSorted.push_back(rb_tree.getMaxDepth());

            vector<int> leafDepthsAVL = avl_tree.getLeafDepths();
                avlLeafDepthsLastSeriesSorted.insert(avlLeafDepthsLastSeriesSorted.end(), leafDepthsAVL.begin(), leafDepthsAVL.end());

            vector<int> leafDepthsRBST = rbst.getLeafDepths();
                rbstLeafDepthsLastSeriesSorted.insert(rbstLeafDepthsLastSeriesSorted.end(), leafDepthsRBST.begin(), leafDepthsRBST.end());

            vector<int> leafDepthsRB = rb_tree.getLeafDepths();
                rbLeafDepthsLastSeriesSorted.insert(rbLeafDepthsLastSeriesSorted.end(), leafDepthsRB.begin(), leafDepthsRB.end());
        }
    }

        double avgInsertAVL = totalInsertAVL / 50, avgRemoveAVL = totalRemoveAVL / 50, avgSearchAVL = totalSearchAVL / 50;
    double avgInsertRBST = totalInsertRBST / 50, avgRemoveRBST = totalRemoveRBST / 50, avgSearchRBST = totalSearchRBST / 50;
        double avgInsertRB = totalInsertRB / 50, avgRemoveRB = totalRemoveRB / 50, avgSearchRB = totalSearchRB / 50;

        outFileSorted << N << "," << maxDepthAVL << "," << avgInsertAVL << "," << avgRemoveAVL << "," << avgSearchAVL << ","
            << maxDepthRBST << "," << avgInsertRBST << "," << avgRemoveRBST << "," << avgSearchRBST << ","
```

```
                    << maxDepthRB << "," << avgInsertRB << "," << avgRemoveRB << "," <<
avgSearchRB << "\n";
    }
    outFileSorted.close();

    // Подсчет частот максимальных высот для отсортированных данных
    map<int, int> avlMaxHeightHistSorted, rbstMaxHeightHistSorted, rbMaxHeightHistSorted;
    for (int height : avlMaxHeightsLastSeriesSorted) avlMaxHeightHistSorted[height]++;
    for (int height : rbstMaxHeightsLastSeriesSorted) rbstMaxHeightHistSorted[height]++;
    for (int height : rbMaxHeightsLastSeriesSorted) rbMaxHeightHistSorted[height]++;

    // Запись гистограммы максимальных высот
        writeHistogramToFile("AVL_max_heights_sorted.csv", avlMaxHeightHistSorted, "AVL
Height,Frequency");
      writeHistogramToFile("RBST_max_heights_sorted.csv", rbstMaxHeightHistSorted, "RBST
Height,Frequency");
          writeHistogramToFile("RB_max_heights_sorted.csv", rbMaxHeightHistSorted, "RB
Height,Frequency");

    // Подсчет частот высот веток для отсортированных данных
    map<int, int> avlLeafDepthHistSorted, rbstLeafDepthHistSorted, rbLeafDepthHistSorted;
    for (int depth : avlLeafDepthsLastSeriesSorted) avlLeafDepthHistSorted[depth]++;
    for (int depth : rbstLeafDepthsLastSeriesSorted) rbstLeafDepthHistSorted[depth]++;
    for (int depth : rbLeafDepthsLastSeriesSorted) rbLeafDepthHistSorted[depth]++;

    // Запись гистограммы высот веток
        writeHistogramToFile("AVL_leaf_depths_sorted.csv", avlLeafDepthHistSorted, "AVL
Depth,Frequency");
      writeHistogramToFile("RBST_leaf_depths_sorted.csv", rbstLeafDepthHistSorted, "RBST
Depth,Frequency");
          writeHistogramToFile("RB_leaf_depths_sorted.csv", rbLeafDepthHistSorted, "RB
Depth,Frequency");

    cout << "Results have been written to files." << endl;
    return 0;
```

}

Код реализует три типа деревьев поиска — AVL-дерево, рандомизированное бинарное дерево поиска (RBST) и красно-черное дерево (RB) — и проводит тесты для сравнения их производительности.

Основные функции программы:

1. Реализация деревьев поиска: каждое дерево реализовано в виде отдельного класса (AVLTree, RandomizedBST, RedBlackTree) с набором стандартных операций:

- Вставка (insert): добавляет новый ключ в дерево.
    - Для AVL: использует балансировку через повороты, если нарушен фактор баланса.
    - Для RBST: использует рандомизацию с операциями split и merge.
    - Для RB: добавляет узел как красный и вызывает fixInsert для балансировки.
- Удаление (remove): удаляет узел с заданным ключом.
    - Для AVL: удаляет узел и балансирует дерево через повороты.
    - Для RBST: использует merge для объединения поддеревьев после удаления.
    - Для RB: удаляет узел и вызывает fixRemove для восстановления свойств.
- Поиск (search): проверяет, есть ли ключ в дереве.
    - Реализация стандартная для всех деревьев: рекурсивный спуск по дереву.
- Получение максимальной глубины (getMaxDepth): возвращает максимальную высоту дерева; используется для анализа структуры дерева.
- Получение глубин листьев (getLeafDepths): собирает глубины всех листьев для построения гистограмм; рекурсивно обходит дерево и записывает глубину каждого листа.
- Измерение времени операций (measureTime): замеряет среднее время выполнения операций (вставка, удаление, поиск); выполняет 1000 операций и делит общее время на количество операций.

2. Проведение тестов. В функции main программа проводит тесты для двух сценариев: случайные данные и отсортированные данные. Тесты выполняются для $N = 2^i$ элементов, где i от 10 до 18 (от 1024 до 262144 элементов). Для каждого N:

- Создание данных:
  - Для случайных данных: генерируются N чисел, которые перемешиваются.
  - Для отсортированных данных: числа от 0 до $N - 1$ вставляются по порядку.
- 50 повторений:
  - Для каждого повторения создаются три дерева (AVL, RBST, RB), заполняются N элементами.
  - Замеряется:
    - Максимальная глубина дерева.
    - Среднее время вставки, удаления и поиска (по 1000 операций).
  - Для последней серии собираются данные для гистограмм:
    - Максимальные высоты деревьев.
    - Глубины всех листьев.
- Запись результатов:
  - В файл results_random.csv и results_sorted.csv записываются максимальные глубины и средние времена операций.
  - В отдельные файлы записываются гистограммы:
    - AVL_max_heights_random.csv, RBST_max_heights_random.csv, RB_max_heights_random.csv — гистограммы максимальных высот.
    - AVL_leaf_depths_random.csv, RBST_leaf_depths_random.csv, RB_leaf_depths_random.csv — гистограммы глубин листьев.
    - Аналогичные файлы для отсортированных данных.

3. Построение гистограмм. Собираются максимальные глубины деревьев и глубины всех листьев за 50 повторений для $N = 2^{18}$. Используется std::map

для подсчета частот, затем данные записываются в CSV-файлы с помощью функции writeHistogramToFile.

Специфические элементы реализации:

1. Генерация случайных данных.

- Генератор случайных чисел:
    - Используется random_device и mt19937 для генерации случайных чисел.
    - random_device rd обеспечивает начальную "затравку" для генератора mt19937 gen(rd()).
    - Это важно для RBST, где случайность используется для балансировки, и для перемешивания данных в тестах.
- Перемешивание:
    - shuffle(testKeys.begin(), testKeys.end(), gen) используется для создания случайного порядка вставки в тестах.

2. Реализация AVL-дерева.

- Балансировка:
    - После каждой вставки и удаления проверяется фактор баланса (getBalance): разница высот левого и правого поддерева.
    - Если фактор > 1 или < -1, выполняются повороты:
        - Простые: rightRotate, leftRotate.
        - Сложные: комбинации поворотов (например, leftRotate на левом поддереве, затем rightRotate на узле).
    - Высота узла:
        - Хранится в поле height и обновляется после каждой операции: node->height = 1 + max(getHeight(node->left), getHeight(node->right)).

3. Реализация рандомизированного дерева (RBST)

- Балансировка через случайность:
    - Используются операции split и merge:
        - split разделяет дерево на два поддерева по ключу.

        - merge объединяет два поддерева, выбирая корень случайным образом с вероятностью, пропорциональной размерам поддеревьев.
    - При вставке с вероятностью $\frac{1}{N+1}$ новый узел становится корнем, иначе вставка выполняется как в обычном BST.
- Поле size:
    - Каждый узел хранит размер своего поддерева (node->size), что позволяет вычислять вероятности для merge.
    - Обновляется через updateSize после каждой операции.

4. Реализация красно-черного дерева.

- Цвет узлов:
    - Узлы имеют поле color (перечисление RED или BLACK).
    - Новый узел всегда красный, что может нарушить свойства дерева.
- Балансировка при вставке (fixInsert):
    - Если два красных узла рядом:
        - Если дядя красный: перекрашиваем (дядя и родитель → черные, дедушка → красный).
        - Если дядя черный: выполняем повороты (rotateLeft, rotateRight) и перекрашиваем.
- Балансировка при удалении (fixRemove):
    - Если удален черный узел, восстанавливаем черную высоту:
        - Если брат красный: перекрашиваем и поворачиваем.
        - Если брат черный: в зависимости от цвета его детей выполняем перекраску или повороты.
- Указатель на родителя:
    - Узлы хранят указатель parent, что упрощает навигацию вверх по дереву при балансировке.

5. Измерение времени.

- Метод measureTime:

- Использует шаблон Func для универсального замера времени операций (вставка, удаление, поиск).
- Выполняет 1000 "прогревочных" операций на копии дерева, чтобы минимизировать влияние кэша.
- Замеряет время с помощью chrono::steady_clock и возвращает среднее время на операцию.

- Особенность:
  - Для тестов создается копия дерева (tempTree), чтобы не портить основное дерево во время "прогрева".

6. Сбор данных для гистограмм.

- Глубины листьев:
  - Метод getLeafDepths рекурсивно обходит дерево и собирает глубины всех листьев.
  - Для $N = 2^{18}$ данные собираются за все 50 повторений и объединяются в один вектор (avlLeafDepthsLastSeries и т.д.).
- Максимальные высоты. Для $N = 2^{18}$ максимальные высоты каждого дерева за 50 повторений записываются в отдельные векторы (avlMaxHeightsLastSeries и т.д.).
- Частоты: используется std::map для подсчета частот (map<int, int>), где ключ — высота или глубина, а значение — количество таких высот/глубин.

7. Работа с файлами.

- Запись результатов:
  - Функция writeHistogramToFile записывает гистограммы в CSV-файлы.
  - Основные результаты (максимальные глубины, средние времена) записываются в results_random.csv и results_sorted.csv.

Выводы

В ходе выполнения лабораторной работы были реализованы классы AVLTree, RandomizedBST и RedBlackTree для самобалансирующегося AVL-дерева, рандомизированного бинарного дерева поиска и красно-чёрного дерева соответственно, с методами вставки, удаления и поиска.

Проведены эксперименты со сгенерированными случайно и отсортированными массивами для размеров данных от 2^{10} до 2^{18}, где измерялось время операций вставки, удаления и поиска, а также максимальная высота деревьев и глубины листьев. Для каждого размера данных выполнено 50 серий тестов с случайными и отсортированными данными, в каждой серии проводилось 1000 операций вставки, удаления и поиска. Результаты представлены в таблицах (CSV-файлы results_random.csv и results_sorted.csv).

Были построены следующие графики:

- Графики зависимости среднего времени выполнения операций вставки, удаления и поиска от количества элементов для случайных (рис. 9а – 9в) и отсортированных (рис. 10а – 10в) массивов при помощи полученных в результате работы программы данных.
- График зависимости максимальной высоты деревьев от количества элементов N для каждого типа дерева (рис. 9г для случайных данных и рис. 10г – для отсортированных).
- Гистограммы распределения максимальных высот для последней серии тестов ($N = 2^{18}$) для AVL, RBST и красно-чёрного деревьев для случайных (рис. 11а – 11в) и отсортированных (рис. 12а – 12в) данных.
- Гистограммы распределения высот веток в для последней серии тестов ($N = 2^{18}$) для AVL, RBST и красно-чёрного деревьев для случайных (рис. 13а – 13в) и отсортированных (рис. 14а – 14в) данных.

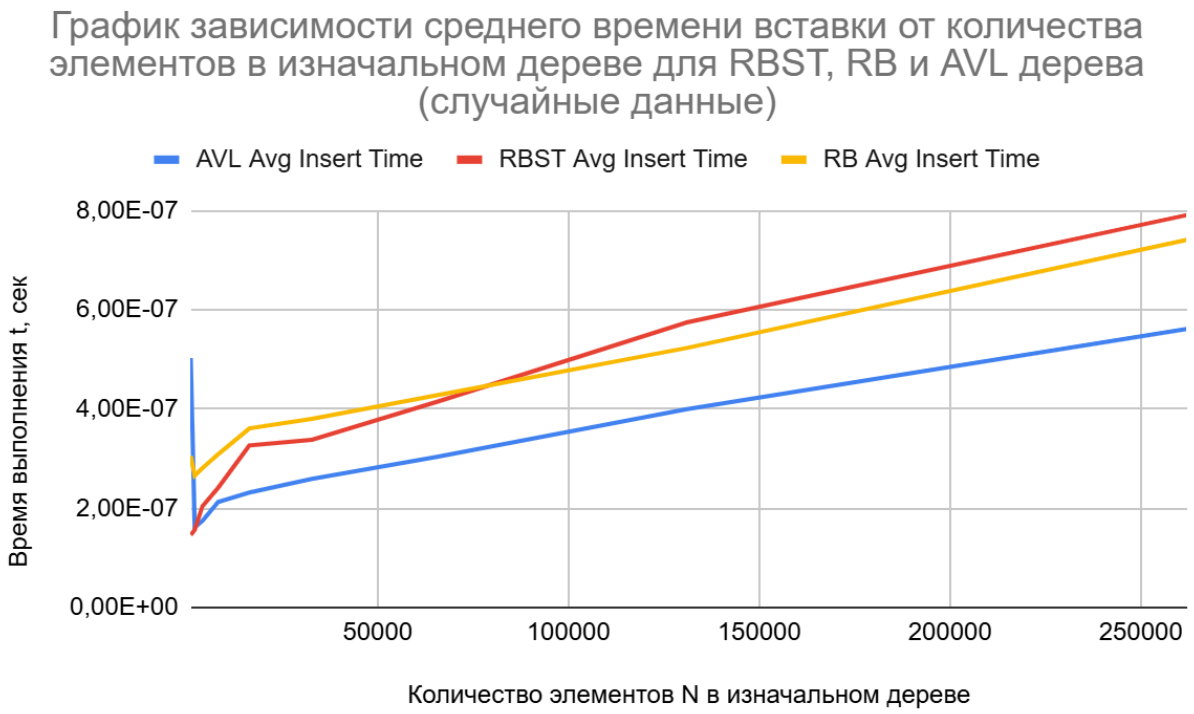

Рисунок 9а – График зависимости времени выполнения операции вставки от количества элементов для случайных данных

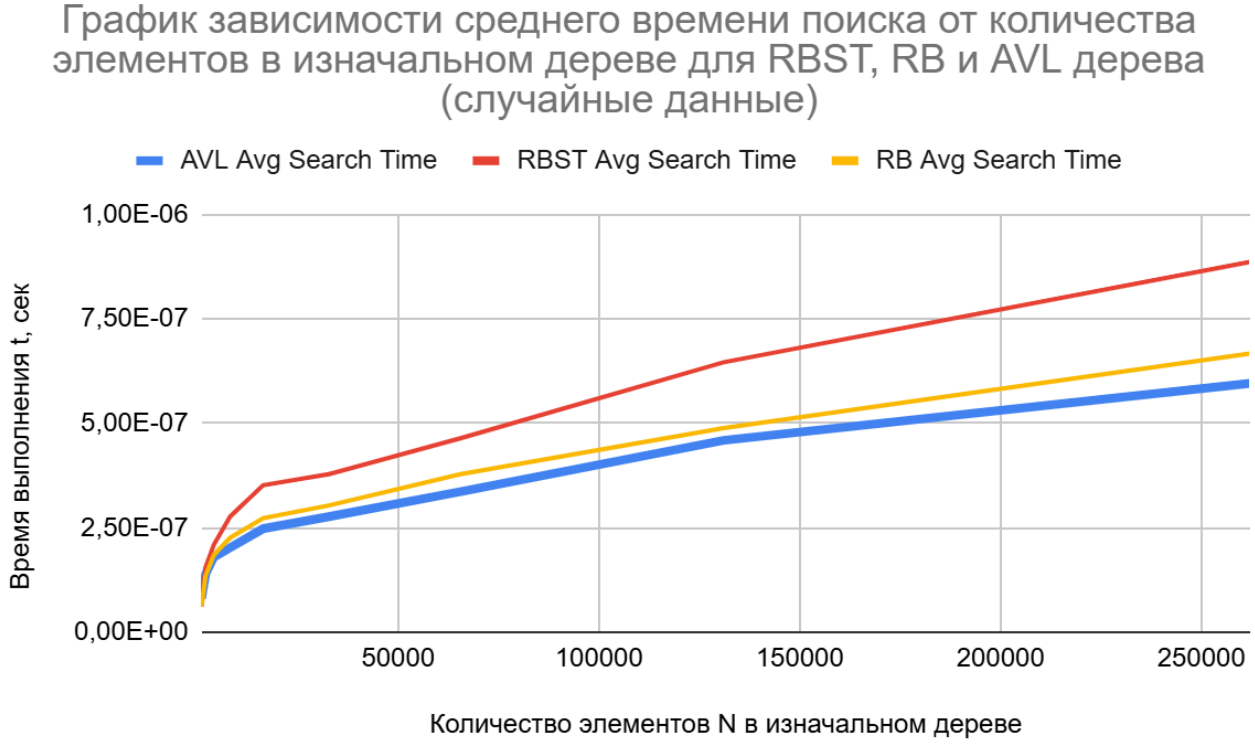

Рисунок 9б – График зависимости времени выполнения операции поиска от количества элементов для случайных данных

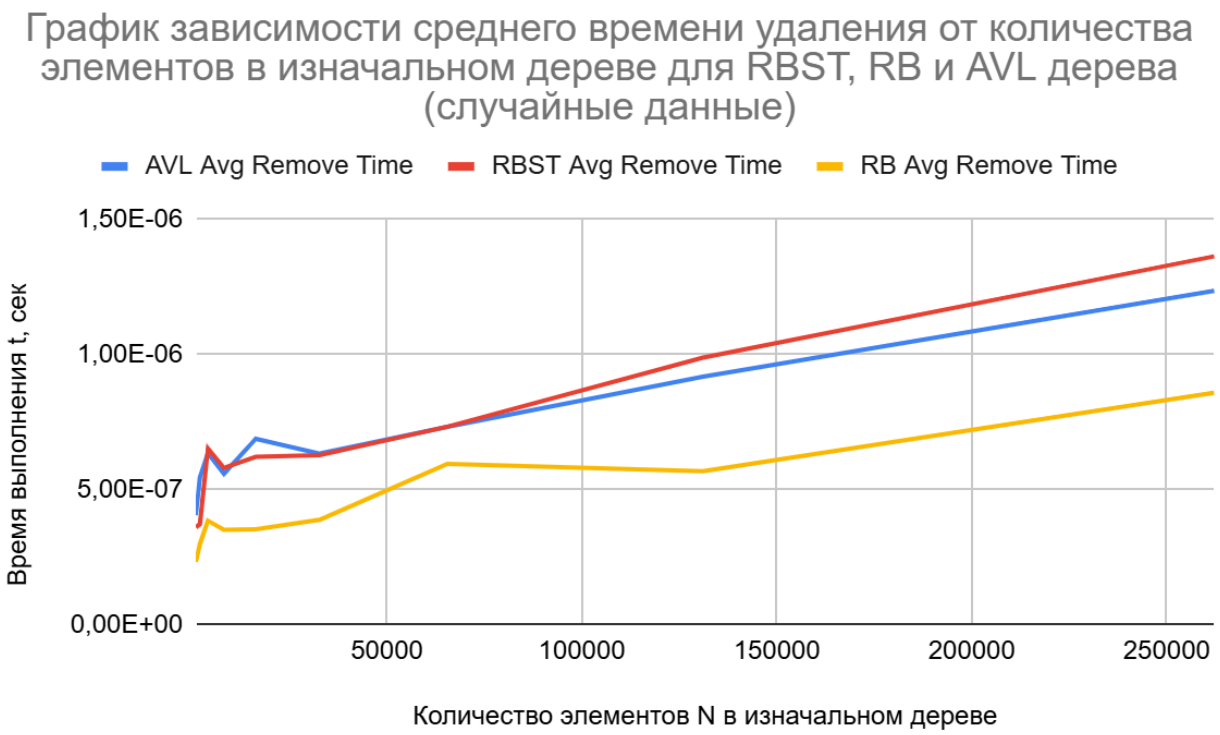

Рисунок 9в – График зависимости времени выполнения операции удаления от количества элементов для случайных данных

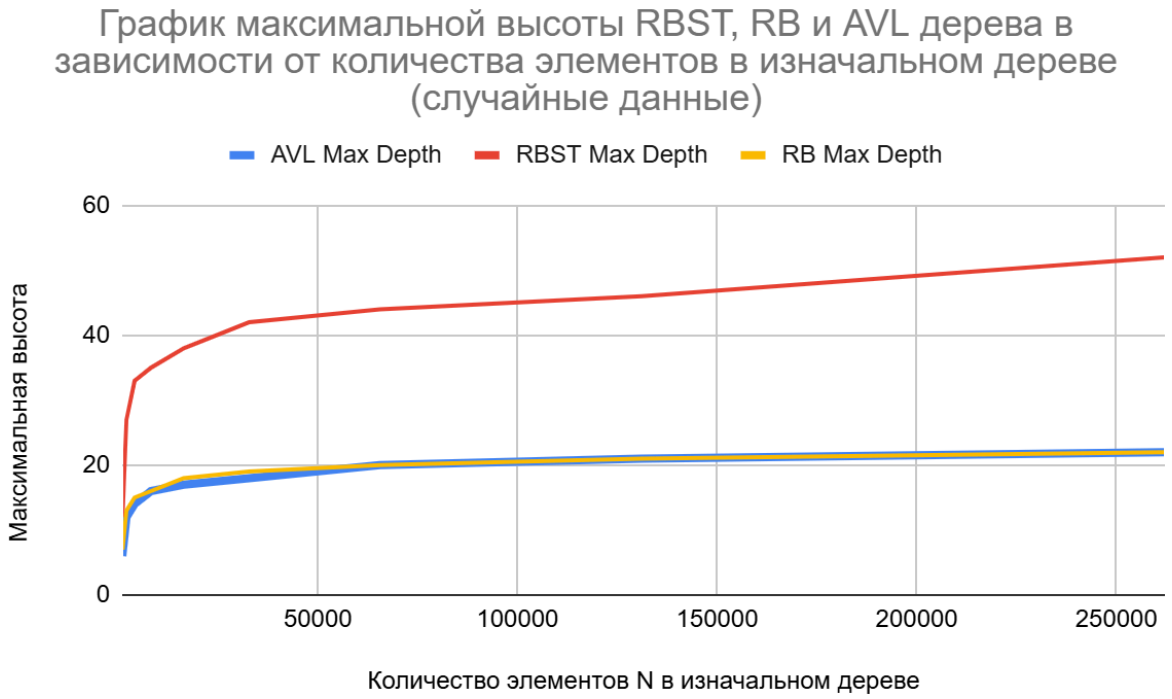

Рисунок 9г – График зависимости максимальной высоты деревьев от количества элементов N для случайных данных

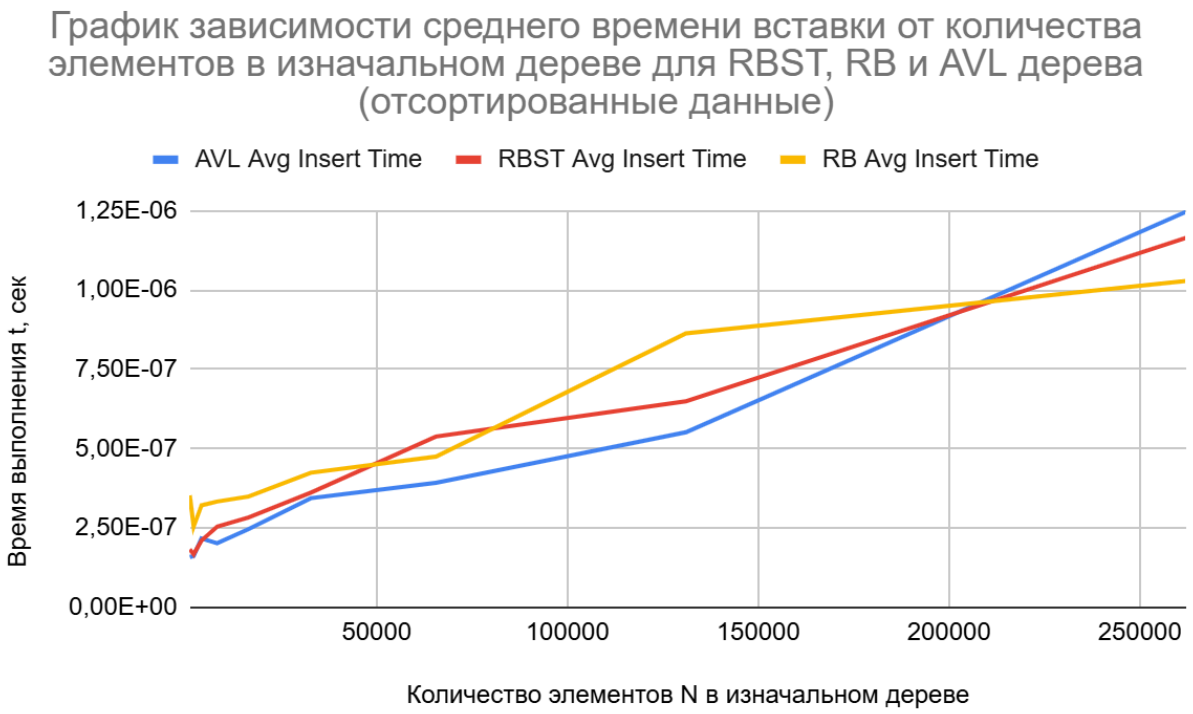

Рисунок 10а – График зависимости времени выполнения операции вставки от количества элементов для отсортированных данных

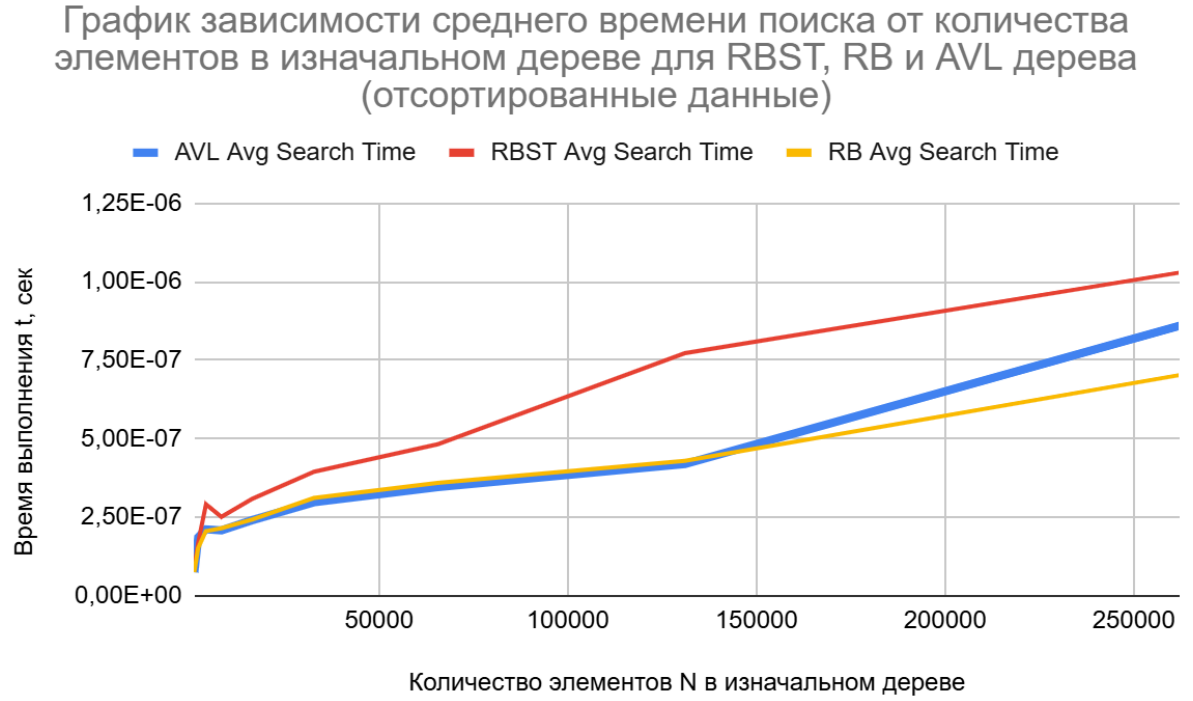

Рисунок 10б – График зависимости времени выполнения операции поиска от количества элементов для отсортированных данных

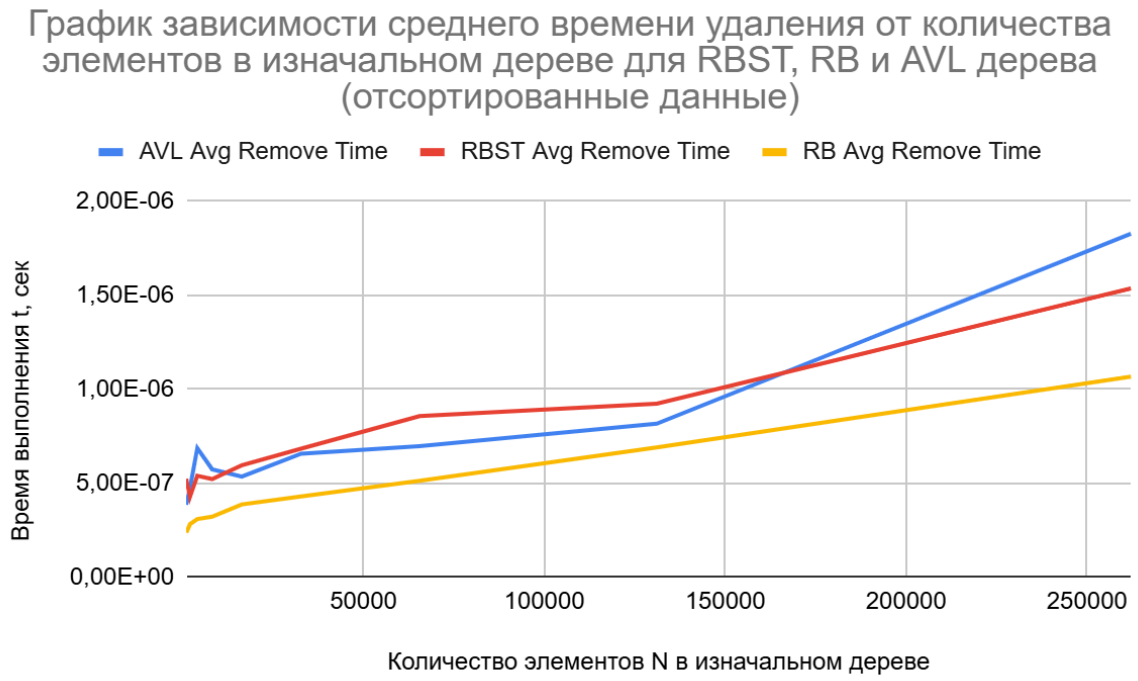

Рисунок 10в – График зависимости времени выполнения операции удаления от количества элементов для отсортированных данных

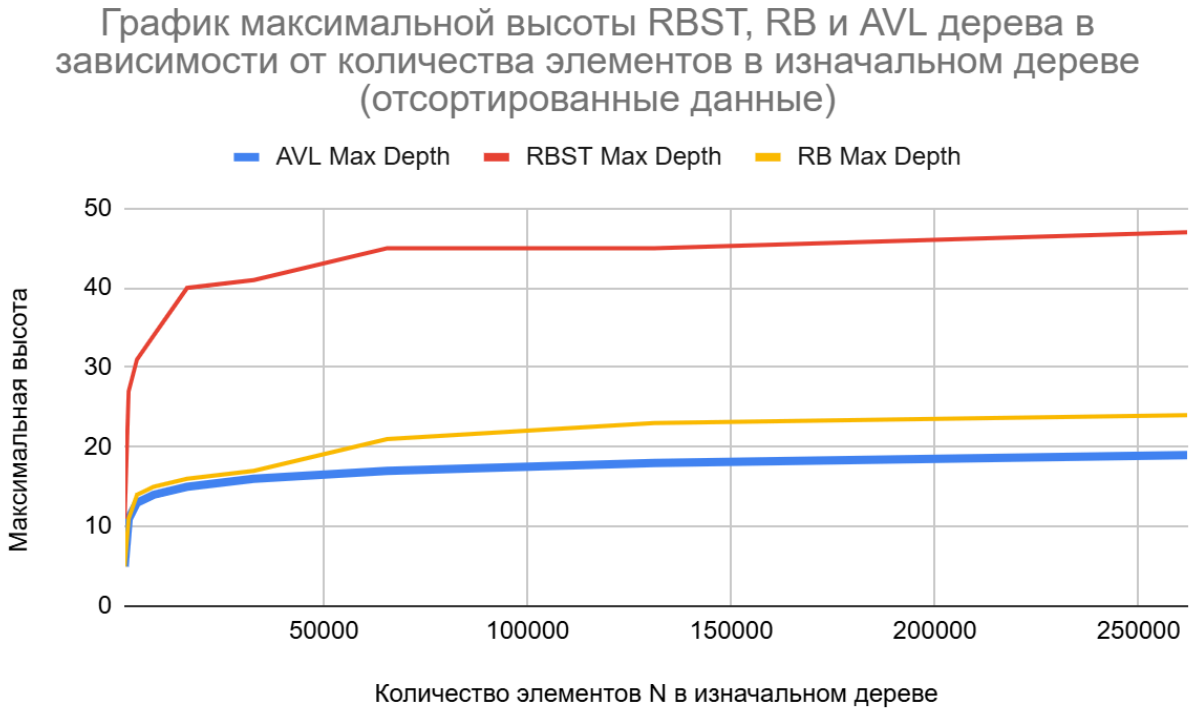

Рисунок 10г – График зависимости максимальной высоты деревьев от количества элементов N для отсортированных данных

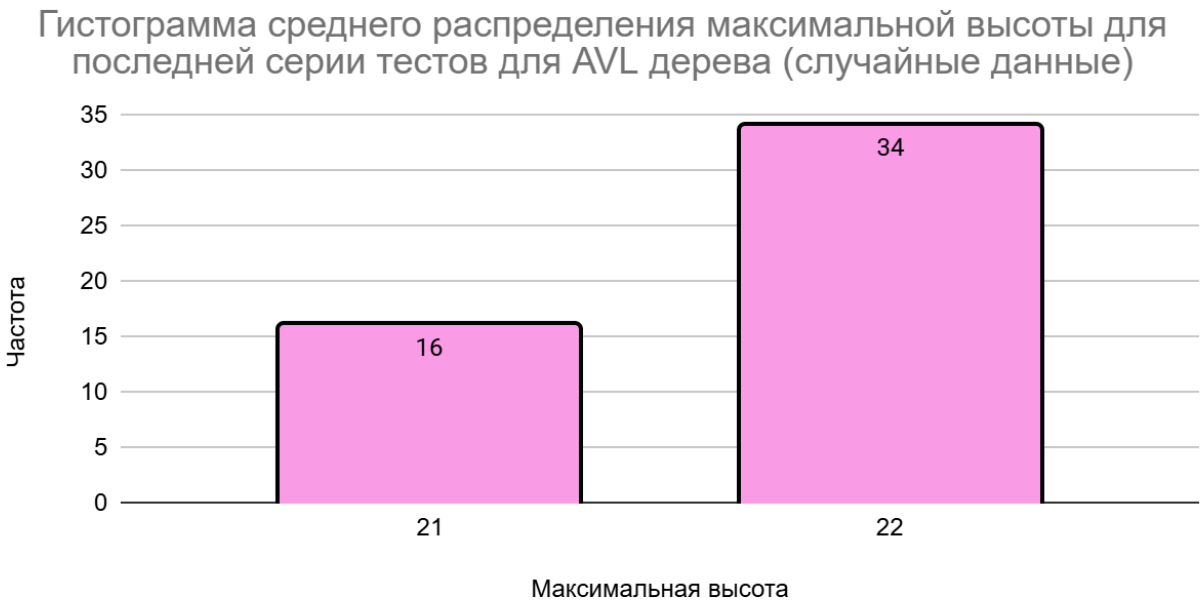

Рисунок 11а – Гистограмма распределения максимальных высот для последней серии тестов для AVL для случайных данных

Рисунок 11б – Гистограмма распределения максимальных высот для последней серии тестов для RBST для случайных данных

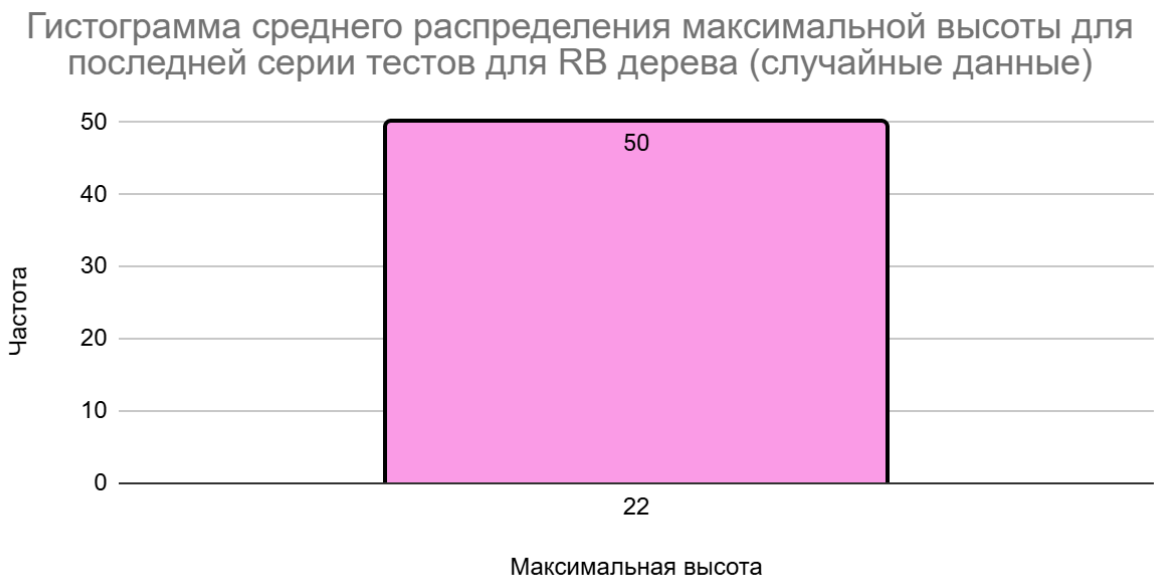

Рисунок 11в – Гистограмма распределения максимальных высот для последней серии тестов для красно-черного дерева для случайных данных

Рисунок 12а – Гистограмма распределения максимальных высот для последней серии тестов для AVL для отсортированных данных

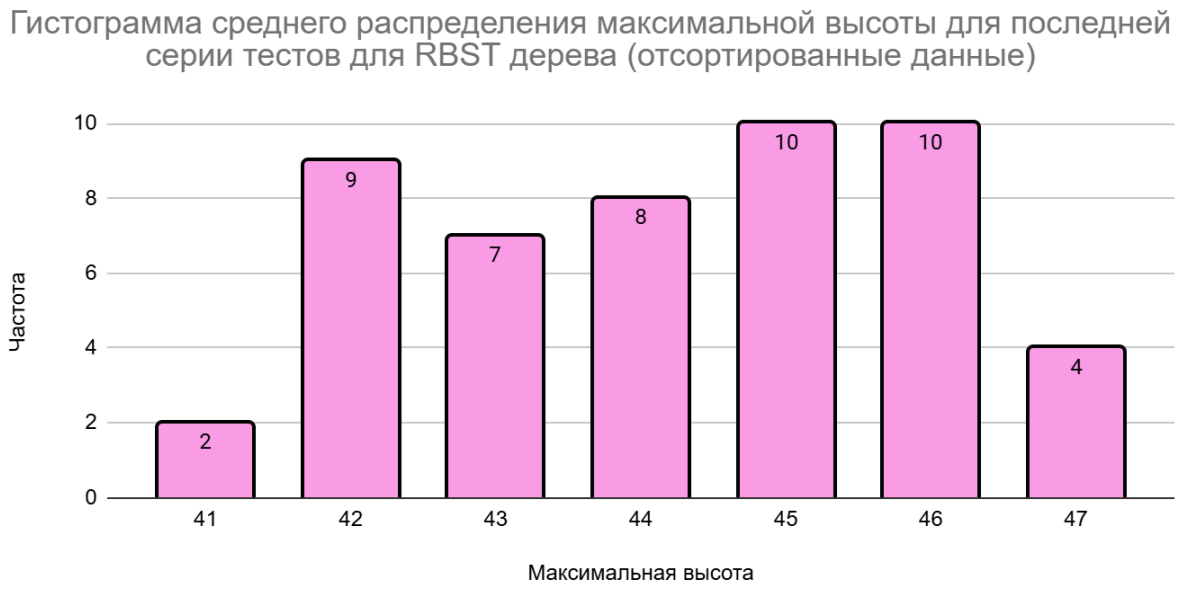

Рисунок 12б – Гистограмма распределения максимальных высот для последней серии тестов для RBST для отсортированных данных

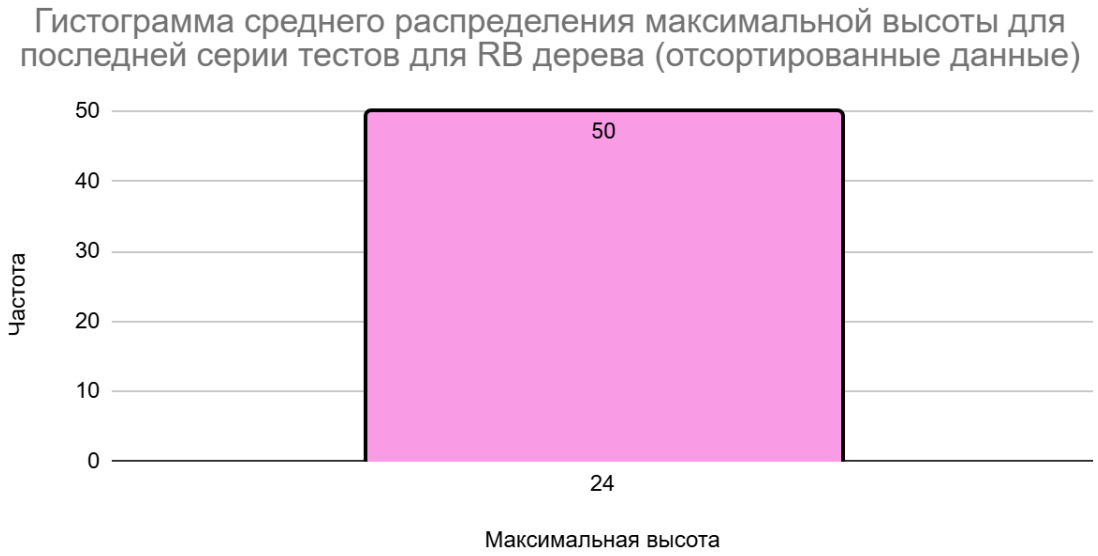

Рисунок 12в – Гистограмма распределения максимальных высот для последней серии тестов для красно-черного дерева для отсортированных данных

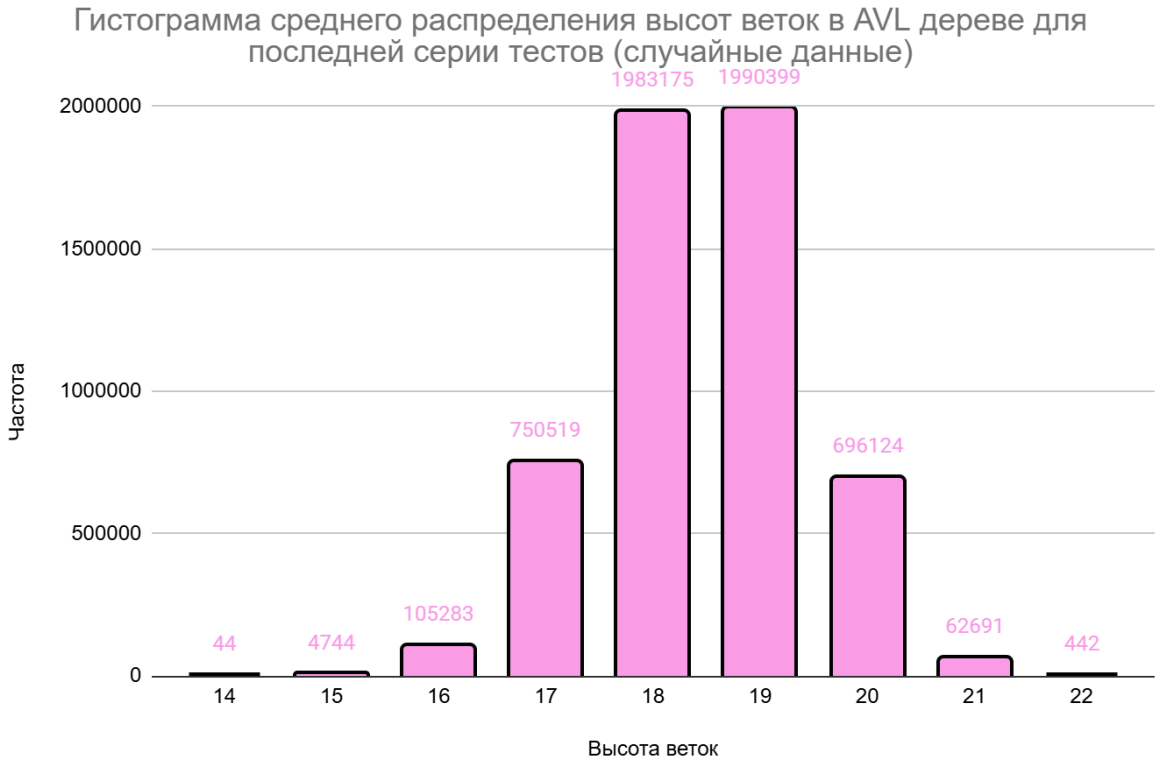

Рисунок 13а – Гистограмма распределения высот веток для последней серии тестов для AVL для случайных данных

Рисунок 13б – Гистограмма распределения высот веток для последней серии тестов для RBST для случайных данных

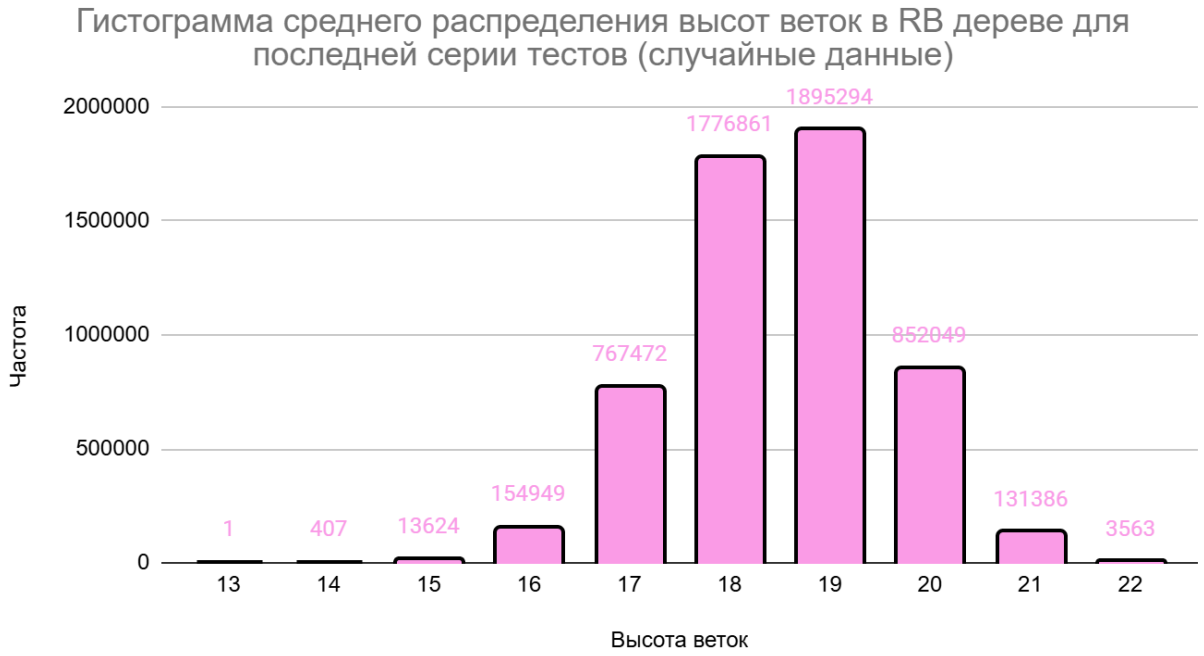

Рисунок 13в – Гистограмма распределения высот веток для последней серии тестов для красно-черного дерева для случайных данных

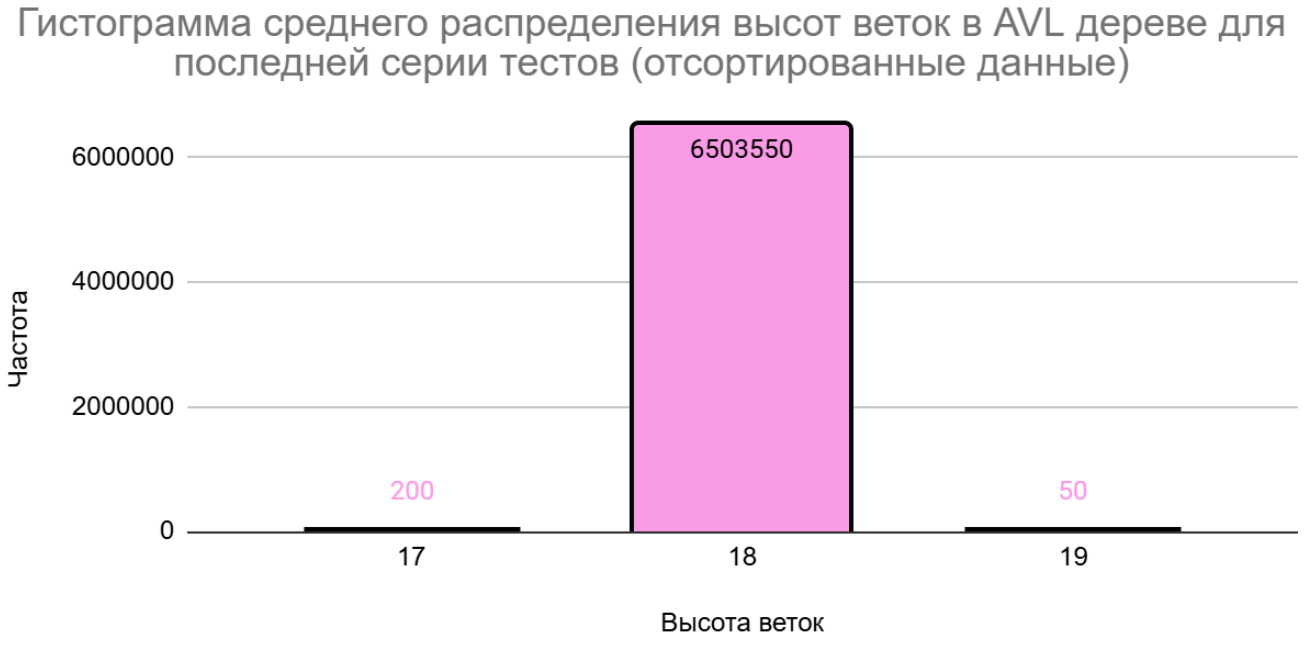

Рисунок 14а – Гистограмма распределения высот веток для последней серии тестов для AVL для отсортированных данных

Рисунок 14б – Гистограмма распределения высот веток для последней серии тестов для RBST для отсортированных данных

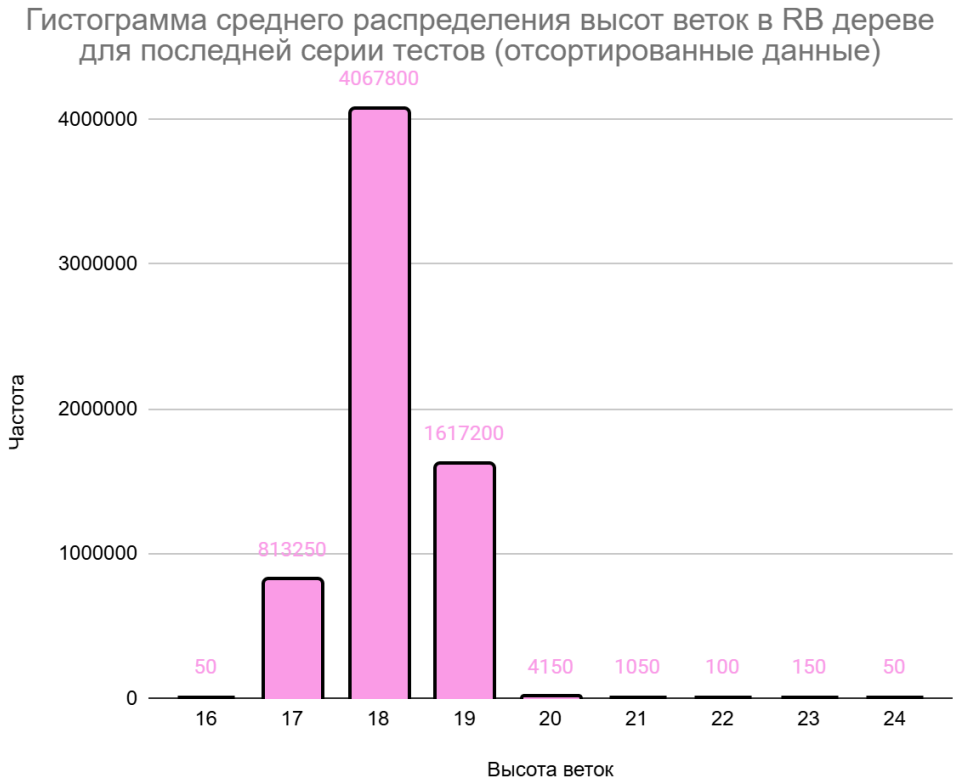

Рисунок 14в – Гистограмма распределения высот веток для последней серии тестов для красно-черного дерева для отсортированных данных

В ходе лабораторной работы установлено, что AVL-дерево и красно-чёрное дерево (RB) значительно превосходят рандомизированное бинарное дерево поиска (RBST) по производительности для отсортированных данных, обеспечивая логарифмическую сложность $O(log_2 N)$ для операций вставки, поиска и удаления. RBST в таких сценариях демонстрирует немного более больший рост времени операций.

Для случайных данных все три структуры демонстрируют логарифмический рост времени, но RBST работает медленнее. AVL и RB показывают близкие результаты, но AVL немного быстрее на поиске благодаря меньшей высоте.

Гистограммы распределения высот веток подтверждают, что AVL и RB остаются более сбалансированными: для случайных данных высоты веток AVL сосредоточены в диапазоне 18–20, а RB — в 14–22, тогда как RBST имеет широкий разброс (6–52). Для отсортированных данных RBST вырождается сильнее (высоты веток от 6 до 52), в то время как AVL (17–19) и RB (16–24) сохраняют сбалансированность. Максимальные высоты также показывают преимущество AVL и RB: их значения сосредоточены в узких диапазонах (19–22 для AVL, 22–24 для RB), тогда как RBST имеет высоты от 40 до 52.

Результаты показывают, что все три реализованных дерева имеют логарифмическую сложность $O(log_2 N)$, однако рандомизированное бинарное деревов поиска в большинстве рассмотренных графиках выполняет операции чуть медленнее.