Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 7

Выполнил студент группы КС-36 Полковникова Д.Д.

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/Polkovnikova\_CS-36

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 14.04.2025

Оглавление

[Описание задачи. 2](#_Toc194912138)

[Описание метода/модели 2](#_Toc194912139)

[Выполнение задачи. 8](#_Toc194912140)

[Заключение. 10](#_Toc194912141)

# Описание задачи.

Рандомизированное дерево против AVL-Дерева

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

Декартово дерево (<https://habr.com/ru/post/101818/>)

Рандомизированное дерево (<https://habr.com/ru/post/145388/>) (мой вариант)

Для этого его потребуется реализовать и сравнить в работе с реализованным ранее AVL-деревом. Для анализа работы алгоритма понадобиться провести серии тестов:

1) В одной серии тестов проводится 50 повторений

2) Требуется провести серии тестов для N = 2^i элементов, при этом i от 10 до 18 включительно.

В рамках одной серии понадобится сделать следующее:

1) Генерируем N случайных значений.

2) Заполнить два дерева N количеством элементов в одинаковом порядке.

3) Для каждого из серий тестов замерить максимальную глубину полученного 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.

7) Для каждого дерева замерить глубины всех веток дерева.

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 максимальной высоты полученного дерева в зависимости от N.

5) Гистограмму среднего распределения максимальной высоты для последней серии тестов для AVL и для вашего варианта.

6) Гистограмму среднего распределения высот веток в AVL дереве и для вашего варианта, для последней серии тестов.

**Задания со звездочкой = + 5 дополнительных первичных баллов:**

7) Аналогичная серия тестов и сравнение ее для отсортированного заранее набора данных

8) Реализовать красно черное дерево и провести все те же проверки с ним.

# Описание метода/модели

**Двоичное дерево –** это иерархическая структура, в которой каждый узел содержит не более чем двух потомков.

Для каждого узла, тот узел, который стоит выше по иерархии для него, называют **родительским узлом**, а те узлы, что стоят ниже, для которых этот узел является родительским, называются **правым и левым наследниками**.

**Самобалансирующиеся деревья**

**Самобалансирующееся дерево** — это бинарное дерево поиска (БДП), которое автоматически поддерживает свою высоту близкой к минимально возможной при вставке и удалении элементов. Это гарантирует, что операции поиска, вставки и удаления выполняются за **O(log n)** в худшем случае.

**AVL-дерево**

**AVL-дерево** — это частный случай самобалансирующегося дерева, в котором для каждой вершины выполняется условие баланса:

**Высота левого поддерева (hₗ) и высота правого поддерева (hᵣ) отличаются не более чем на 1.**  
Формально:

hₗ - hᵣ| \leq 1 ]

Если это условие нарушается, дерево **балансируется** с помощью поворотов.

**Балансировка AVL-дерева**

При вставке или удалении узла может нарушиться баланс. Тогда применяются **повороты**:

1. **Малый (одинарный) поворот** – если дисбаланс вызван "внешним" поддеревом.
2. **Большой (двойной) поворот** – если дисбаланс вызван "внутренним" поддеревом.

**1. Малый поворот (Single Rotation)**

Применяется, когда дисбаланс возникает в одном направлении (левом-левом или правом-правом).

Малый правый поворот (RR-поворот)

Ситуация: Левое поддерево левого потомка слишком высокое (LL-дисбаланс).  
Действие: Поднимаем левого потомка наверх.

Малый левый поворот (LL-поворот)

Ситуация: Правое поддерево правого потомка слишком высокое (RR-дисбаланс).  
Действие: Поднимаем правого потомка наверх.

**2. Большой поворот (Double Rotation)**

Применяется, когда дисбаланс возникает в зигзагообразной структуре (левом-правом или правом-левом поддереве).

Большой лево-правый поворот (LR-поворот)

Ситуация: Правое поддерево левого потомка слишком высокое (LR-дисбаланс).  
Действие:

1. Делаем левый поворот для левого поддерева.
2. Делаем правый поворот для корня.

Большой право-левый поворот (RL-поворот)

Ситуация: Левое поддерево правого потомка слишком высокое (RL-дисбаланс).  
Действие:

1. Делаем правый поворот для правого поддерева.
2. Делаем левый поворот для корня.

**Когда применять повороты?**

После **вставки** или **удаления** проверяем баланс всех предков измененного узла. Если баланс нарушен (разница высот > 1), выбираем поворот по **структуре дерева**:

| **Тип дисбаланса** | **Поворот** |
| --- | --- |
| LL (левое поддерево левого потомка тяжелое) | **Правый малый** (rotate\_right) |
| RR (правое поддерево правого потомка тяжелое) | **Левый малый** (rotate\_left) |
| LR (правое поддерево левого потомка тяжелое) | **Большой LR** (rotate\_lr) |
| RL (левое поддерево правого потомка тяжелое) | **Большой RL** (rotate\_rl) |

**Когда использовать AVL-дерево?**

* Когда **часто происходят поиски**, но редко вставки/удаления (т.к. балансировка затратна).
* Когда **критична гарантированная O(log n) сложность** (в отличие от обычного БДП, которое может выродиться в O(n)).
* В базах данных, файловых системах, кэшах.

**Альтернативы**

* **Красно-черные деревья** – менее строгий баланс, но быстрее вставка/удаление.
* **Splay-деревья** – адаптируются под частые запросы.
* **B-деревья** – оптимизированы для дисковых структур.

**Вывод**

AVL-деревья обеспечивают **строгий баланс**, что делает их идеальными для задач, где важна **предсказуемая производительность поиска**. Балансировка выполняется через **малые и большие повороты** в зависимости от структуры дисбаланса.

**Почему высота ≈ log₂(n)?**

В **идеально сбалансированном** BST:

* Каждый уровень содержит **в 2 раза больше узлов**, чем предыдущий.
* Для **n узлов** количество уровней (высота) будет:

n≈2^h ⟹*h*≈log2​*n*

В AVL-дереве баланс **немного слабее**, но всё равно:

h≤1.44⋅log2(n+2)

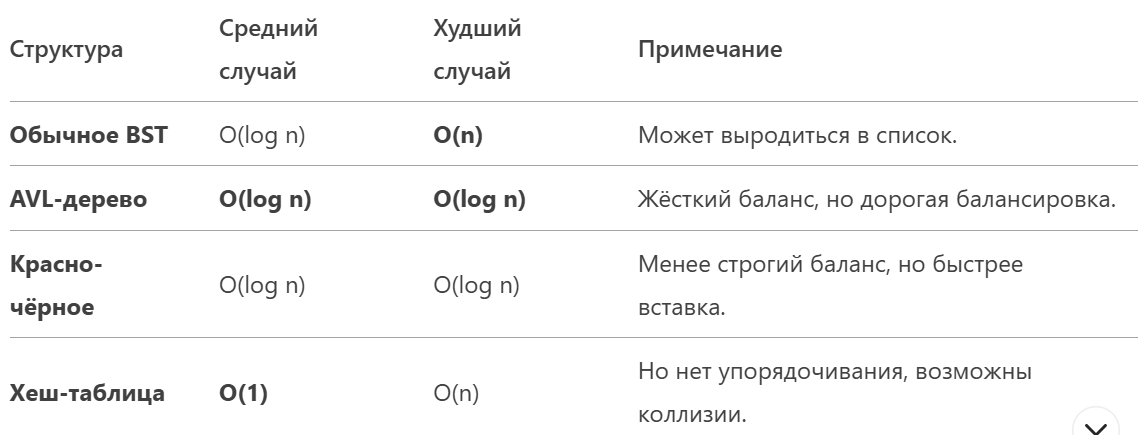
(Это следует из свойств чисел Фибоначчи, так как наихудший случай — дерево, построенное по принципу чисел Фибоначчи.)

**Как это влияет на сложность операций?**

* **Поиск, вставка, удаление** требуют **прохода от корня до листа** → **максимум h шагов**.
* Поскольку **h = O(log n)**, то и операции **O(log n)**.

**2. Почему в обычном BST может быть O(n)?**

Если дерево **не балансируется**, то при **неудачных** вставках оно может **выродиться** в **связанный список**.



# Выполнение задачи.

Для рандомизированного дерева и AVL-дерева графики получились следующими:

Гистограммы для рандомизированного дерева:

Гистограммы для AVL-дерева:

**1. Рандомизированные деревья (Randomized BST)**

**Плюсы:**

**Простота реализации:**Не требуют сложных операций балансировки, только базовые повороты

**Хорошее среднее время операций:** O(log n) в среднем случае

**Амортизированная эффективность:** Нет необходимости поддерживать дополнительные поля (только размер поддерева)

**Устойчивость к определённым паттернам ввода:** Равномерное распределение обеспечивается рандомизацией

**Быстрая вставка: Вставка в корень может быть эффективнее, чем полная балансировка**

**Минусы:**

**Нет гарантий производительности**: В худшем случае (крайне маловероятном) может выродиться в O(n)

**Больший разброс глубины**: Максимальная глубина может значительно превышать log₂n

**Менее предсказуемое поведение**: Производительность может варьироваться между запусками

**Требует генератора случайных чисел**: Добавляет небольшие накладные расходы

**2. AVL-деревья**

**Плюсы:**

**Строгая балансировка**: Гарантированная высота ~1.44log₂n (идеально сбалансировано)

**Стабильная производительность**: Все операции гарантированно O(log n)

**Предсказуемость**: Время операций практически одинаково для одинаковых n

**Эффективность поиска**: Идеально подходит для частых операций поиска

**Детерминированное поведение**: Результаты воспроизводимы между запусками

**Минусы:**

**Сложность реализации**: Требуется обработка 4 типов поворотов

**Дополнительные накладные расходы**: Необходимо хранить и обновлять высоту узлов

**Медленнее при частых модификациях**: Балансировка после каждой вставки/удаления

**Избыточная балансировка**: Может выполнять повороты даже когда это не критично

**Ключевые отличия на практике (по вашим тестам):**

1. **Глубина дерева**:
   * AVL: Все деревья имели глубину 18-19 (для N=262144)
   * Рандомизированные: Разброс глубин был значительно больше (30-50)
2. **Распределение веток**:
   * AVL: Узкий диапазон глубин веток (почти идеальная балансировка)
   * Рандомизированные: Широкое распределение глубин
3. **Стабильность**:
   * AVL: Практически идентичные результаты при каждом запуске
   * Рандомизированные: Результаты варьируются между запусками

**Когда что выбирать:**

* **Выбирайте AVL**, если:
  + Критичен гарантированный быстрый поиск
  + Важна предсказуемость производительности
  + Операции поиска преобладают над вставками/удалениями
* **Выбирайте рандомизированные**, если:
  + Нужна простая реализация
  + Входные данные достаточно случайны
  + Допустима небольшая вариативность в производительности
  + Частые вставки/удаления (меньше накладных расходов)

Данные результаты можно объяснить следующим:

1. **Балансировка**:
   * AVL-дерево поддерживает строгую балансировку (разница высот поддеревьев не более 1)
   * Используются 4 типа поворотов для балансировки
2. **Структура узла**:
   * Хранит высоту поддерева вместо размера
   * Нет вероятностной логики при вставке
3. **Производительность**:
   * Вставка и удаление гарантированно O(log n)
   * Более строгие требования к балансировке
4. **Результаты тестирования**:
   * Максимальная глубина будет ближе к log2(N)
   * Распределение высот веток будет более узким
   * Времена операций могут быть стабильнее

**Дополнительное задание**

Для сортированного списка будут следующие графики:

Гистограммы для рандомизированного дерева:

Гистограммы для AVL-дерева:

**Красно-чёрное дерево**

Красно-чёрное дерево для отсортированного списка:

# Заключение.

Все три структуры данных обеспечивают **логарифмическую сложность операций** (O(log n)) для вставки, удаления и поиска, но имеют разные особенности.

**1. Красно-чёрное дерево (Red-Black Tree)**

Плюсы:

Хорошо сбалансировано, но не идеально (высота ≤ 2 log₂(n+1)).

Меньше поворотов при вставке/удалении, чем в AVL.

Быстрее на частых вставках/удалениях, чем AVL.

Используется в std::map и std::set (C++).

Минусы:

Менее строгий баланс, чем у AVL → поиск может быть немного медленнее.

Лучше использовать:

Когда нужно часто модифицировать дерево (вставки/удаления).

В реализациях ассоциативных массивов (например, в стандартных библиотеках).

**2. AVL-дерево**

Плюсы:

Идеально сбалансировано (высота ≤ 1.44 log₂(n+2)).

Самый быстрый поиск из-за строгой балансировки.

Минусы:

Частые повороты при вставке/удалении → медленнее модифицируется, чем красно-чёрное.

Требует больше памяти на хранение баланс-факторов.

Лучше использовать:

Когда поиск критически важен, а изменения редки (например, кэши, статические БД).

**3. Рандомизированное BST (Randomized BST)**

Плюсы:

Простота реализации (нет сложных балансировок).

Хорошо работает на случайных данных (в среднем O(log n)).

Меньше накладных расходов, чем у красно-чёрного и AVL.

Минусы:

Может деградировать на упорядоченных данных (если нет защиты).

Нет гарантии баланса (только вероятностная).

Лучше использовать:

Когда данные поступают случайно и нужна простая реализация.

В алгоритмах, где не требуется строгий баланс (рандомизированные структуры).