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 № 6

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: 07.04.2025

Оглавление

[Описание задачи. 2](#_Toc194912138)

[Описание метода/модели 2](#_Toc194912139)

[Выполнение задачи. 8](#_Toc194912140)

[Заключение. 10](#_Toc194912141)

# Описание задачи.

В рамках лабораторной работы необходимо изучить и реализовать бинарное дерево поиска и его самобалансирующийся вариант в лице AVL дерева.

Для проверки анализа работы структуры данных требуется провести 10 серий тестов.

* В каждой серии тестов требуется выполнять 20 циклов генерации и операций. При этом первые 10 работают с массивом заполненным случайным образом, во второй половине случаев, массив заполняется в порядке возрастания значений индекса, т.е. является отсортированным по умолчанию.
* Требуется создать массив состоящий из 2^(10 + i) элементов, где i это номер серии.
* Массив должен быть помещен в оба варианта двоичных деревьев. При этому замеряется время, затраченное на всю операцию вставки всего массива.
* После заполнения массива, требуется выполнить 1000 операций поиска по обоим вариантам дерева, случайного числа в диапазоне генерируемых значений, замерев время на все 1000 попыток и вычислив время 1 операции поиска.
* Провести 1000 операций поиска по массиву, замерить требуемое время на все 1000 операций и найти время на 1 операцию.
* После, требуется выполнить 1000 операций удаления значений из двоичных деревьев, и замерить время затраченное на все операции, после чего вычислить время на 1 операцию.
* После выполнения всех серий тестов, требуется построить графики зависимости времени затрачиваемого на операции вставки, поиска, удаления от количества элементов. При этом требуется разделить графики для отсортированного набора данных и заполненных со случайным распределением. Также для операции поиска требуется аналогично нанести для сравнения график времени поиска для обычного массива.

# 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**.

Существует рекурсивное определение двоичного дерева:

*<дерево> = (*

*<узел>,*

*<дерево(левое)>, <дерево(правое)>*

*) | пустота*

Т.е. Дерево либо пустое, либо имеет два поддерева, которые в свою очередь тоже либо пустые либо имеют аналогичное устройство.

Вставка O(1)

Удаление O(1)



Рис 1 – Родительский узел и левый и правый потомки для 2.

Список и вектор/массив, но у них есть ряд недостатков связанных в основном с операциями поиска и вставки удаления, так:

* Для списка вставка/удаление имеют сложность О(1), а поиск О(N)
* Для вектора и массива вставка удаление имеют сложность O(N), а поиск О(log(N))

Как видно, у этих структур имеется достоинство только в одном из двух областей, либо в поиске либо в области вставки, как же можно совместить преимущество быстрой вставки/удаления и быстрого поиска.

Именно для этого можно использовать двоичное дерево, превратив его в двоичное дерево поиска, для этого необходимо ввести следующие правила при работе с двоичным деревом:

* Оба поддерева являются двоичными деревьями поиска
* У всех узлов левого поддерева любого узла, хранимое значение всегда меньше либо равно хранимого значения этого узла.
* У всех узлов правого поддерева любого узла, хранимое значение всегда больше либо равно хранимого значения этого узла.

**Двоичное дерево поиска** – это двоичное дерево придерживающееся 2х правил, согласно которым левые потомки всегда меньше или равны текущего элемента, а правые больше или равны текущего элемента.

**Поисковые деревья** — это решения так называемой «словарной проблемы». Предположим, что имеется большое количество ключей, каждый из которых имеет значение. В немецко–английском словаре немецкое слово является ключевым, а английские слова являются значением, которое вы ищете. Аналогично ведет себя телефонная книга с именем и адресом в качестве ключа, а номер телефона — в качестве искомого значения.

Этот подход получил в информатике название «**двоичный поиск**». Она воссоздана очевидным образом с помощью очень известного метода поиска «**двоичный поиск в массиве**». Их поведение оптимально с точки зрения информации, а именно логарифмически.

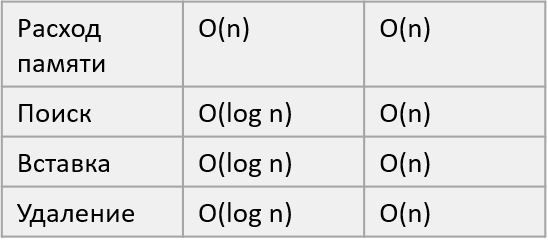


Рис 2 – Для двоичного дерева поиска.

**Обход дерева**

Процесс **обхода дерева** проще всего реализовать как рекурсивный процесс. Последовательно обследуя каждый узел дерева и посещая сначала левую его часть, затем правую. Стандартный порядок обхода выдаст отсортированный по возрастанию результат, его инверсия отсортированный по убыванию. Так же, можно использовать обход в ширину, тогда получится другой порядок обхода.

Для приведенного дерева вывод будет: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10



Рис 3 – Обход дерева.



Рис 4 – Вставка и удаление из дерева.

Процесс вставки в дерево так же рекурсивен. Процесс удаления из дерева так же рекурсивен, особенностью этого процесса является тот факт, что при удалении узла мы должны решить следующие пограничные случаи:

1. Удаляемый узел не имеет потомков. В этом случае узел просто удаляется.
2. Удаляемый узел имеет только левых потомков. В этом случае вместо удаляемого узла становится его левый потомок.
3. Удаляемый узел имеет правых потомков. В этом случае, среди правых потомков требуется найти минимальное значение, т.е. самого левого потомка из правых потомков, удалить его в правом поддереве и вставить на место удаляемого узла, сохранив все связи удаляемого узла.

**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.

# Выполнение задачи.

* **BST** быстро работает **только на случайных данных**, а на отсортированных — **деградирует**.
* **AVL** быстро работает **на любых данных** благодаря балансировке.

Почему так происходит?

**1. BST (Binary Search Tree)**

Случайные данные:

Элементы добавляются в случайном порядке → дерево получается примерно сбалансированным.

Высота дерева ~log n → операции (insert, search, remove) работают за O(log n).

Отсортированные данные:

Элементы добавляются по возрастанию (1, 2, 3, ..., n).

Дерево вырождается в связный список (все элементы идут в правую ветку).

Высота дерева = n → операции работают за O(n) (как линейный поиск).

**2. AVL Tree**

Любые данные (случайные/отсортированные):

После каждой вставки/удаления происходит балансировка (повороты поддеревьев).

Гарантированная высота дерева = O(log n) → все операции работают за O(log n).

Неважно, в каком порядке добавляются элементы — AVL всегда остаётся сбалансированным.

# Заключение.

В ходе лабораторной работы была реализована

**BST (Binary Search Tree)**

* **Плюсы**:
  + Простота реализации.
  + Быстрая вставка/поиск/удаление (O(log n)) **на случайных данных**.
* **Минусы**:
  + **Деградирует до O(n) на отсортированных данных** (вырождается в список).
  + Нет гарантий сбалансированности.
* **Когда использовать**:
  + Если данные **заведомо случайные** (например, хеш-ключи).
  + Когда **память важнее скорости** (AVL требует доп. памяти под балансировку).

**AVL Tree**

* **Плюсы**:
  + **Гарантированная сбалансированность** → всегда O(log n).
  + Предсказуемая производительность **на любых данных** (даже отсортированных).
* **Минусы**:
  + Сложнее реализация (нужны повороты, обновление высот).
  + Немного медленнее на вставке/удалении из-за балансировки.
* **Когда использовать**:
  + Если данные могут быть **частично/полностью отсортированы**.
  + Когда критична **стабильность времени выполнения** (реальные системы, базы данных).