Министерство науки и высшего образования Российской Федерации УНИВЕРСИТЕТ НАУКИ И ТЕХНОЛОГИЙ «МИСИС» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Доклад: “ АВЛ-деревья. Оценка высоты АВЛ-дерева”.**

**по предмету Комбинаторика и теория графов**

Выполнил: Зуйков Арсений Николаевич

группа БИВТ-23-1

Ссылка на реализацию:

[GitHub](https://github.com/jlobsterino/AVL-Tree)

Москва 2024

**Оглавление**

[**Формальная постановка задачи** 4](#_Toc184589326)

[**Описание задачи** 4](#_Toc184589327)

[**Входные данные** 4](#_Toc184589328)

[**Выходные данные** 4](#_Toc184589329)

[**Формальное определение АВЛ-дерева** 4](#_Toc184589330)

[**Свойства бинарного дерева поиска (BST):** 4](#_Toc184589331)

[**Условие сбалансированности АВЛ-дерева:** 4](#_Toc184589332)

[**Высота АВЛ-дерева:** 4](#_Toc184589333)

[**Операции на АВЛ-дереве** 5](#_Toc184589334)

[**Вставка** 5](#_Toc184589335)

[**Удаление** 5](#_Toc184589336)

[**Поиск** 5](#_Toc184589337)

[**Цель задачи** 5](#_Toc184589338)

[**Ограничения** 5](#_Toc184589339)

[**Теоретическое описание алгоритма** 5](#_Toc184589340)

[**Основные понятия** 5](#_Toc184589341)

[**АВЛ-дерево** 5](#_Toc184589342)

[**Высота узла** 6](#_Toc184589343)

[**Балансировочный фактор** 6](#_Toc184589344)

[**Операции на АВЛ-дереве** 6](#_Toc184589345)

[**Вставка элемента** 6](#_Toc184589346)

[**Удаление элемента** 6](#_Toc184589347)

[**Поиск элемента** 7](#_Toc184589348)

[**Балансировка дерева: Операции вращения** 7](#_Toc184589349)

[**Оценка высоты АВЛ-дерева** 7](#_Toc184589350)

[**Характеристика алгоритма** 8](#_Toc184589351)

[**Основные операции в АВЛ-дереве** 8](#_Toc184589352)

[**Временная сложность операций** 8](#_Toc184589353)

[**Поиск элемента** 8](#_Toc184589354)

[**Вставка элемента** 8](#_Toc184589355)

[**Удаление элемента** 9](#_Toc184589356)

[**Балансировка (вращения)** 9](#_Toc184589357)

[**Пространственная сложность** 9](#_Toc184589358)

[**Память на хранение дерева** 9](#_Toc184589359)

[**Память на поддержку операций** 10](#_Toc184589360)

[**Сравнительный анализ алгоритма с аналогичными** 10](#_Toc184589361)

[**АВЛ-дерево (AVL Tree)** 10](#_Toc184589362)

[**Красно-черное дерево (Red-Black Tree)** 10](#_Toc184589363)

[**Бинарное дерево поиска (BST)** 11](#_Toc184589364)

[**Treap (Tree + Heap)** 11](#_Toc184589365)

[**B-дерево (B-Tree)** 11](#_Toc184589366)

[**Перечень инструментов для реализации** 12](#_Toc184589367)

[**Языки программирования** 12](#_Toc184589368)

[**Библиотеки и фреймворки** 12](#_Toc184589369)

[**Инструменты для тестирования и отладки** 13](#_Toc184589370)

[**Инструменты для визуализации** 14](#_Toc184589371)

[**Описание реализации и процесса тестирования на языке Java** 14](#_Toc184589372)

[**Код алгоритма:** 14](#_Toc184589373)

[**Результат работы программы:** 20](#_Toc184589374)

[**Тестирование алгоритма:** 20](#_Toc184589375)

[**Результаты тестов:** 22](#_Toc184589376)

[**Заключение** 22](#_Toc184589377)

# **Формальная постановка задачи**

## **Описание задачи**

АВЛ-дерево (AVL-дерево) — это сбалансированное бинарное дерево поиска (BST), в котором для каждой вершины разница между высотами её левого и правого поддеревьев не превышает 1.

Задача заключается в поддержании этого свойства после каждой операции вставки или удаления узлов. Если баланс нарушается, выполняется перебалансировка дерева путём одного или двух вращений.

## **Входные данные**

1. Последовательность операций:

* Вставка узлов (ввод ключей).
* Удаление узлов.
* Поиск элементов.

1. Начальные условия:

* Дерево пустое или содержит некоторые ключи.
* Ключи уникальны (без повторений).

## **Выходные данные**

После каждой операции требуется:

1. Обновить структуру АВЛ-дерева с сохранением сбалансированности.
2. Высота дерева после вставки, удаления или поиска.
3. Лог выполнения операций (для тестирования или вывода).

## **Формальное определение АВЛ-дерева**

АВЛ-дерево — это бинарное дерево поиска, в котором выполняются следующие условия:

### **Свойства бинарного дерева поиска (BST):**

* Для любого узла N с ключом k:
* Ключи всех узлов в левом поддереве N меньше k.
* Ключи всех узлов в правом поддереве N больше k.
* Левые и правые поддеревья также являются бинарными деревьями поиска.

### **Условие сбалансированности АВЛ-дерева:**

Для любого узла N разница между высотами левого и правого поддерева не превышает 1: Balance(N)=Height(N.left)−Height(N.right)∈{−1,0,1}

### **Высота АВЛ-дерева:**

* Высота дерева h — это длина самого длинного пути от корня дерева до листа.
* Максимальная высота АВЛ-дерева при n узлах равна: h≤1.44⋅log2(n+2)−0.328 или приближённо: h≈O(log(n))

## **Операции на АВЛ-дереве**

### **Вставка**

* Если вставка нарушает баланс узла, требуется восстановить баланс путём одного из следующих вращений:
* Правое вращение (LL): когда левая ветвь слишком длинная.
* Левое вращение (RR): когда правая ветвь слишком длинная.
* Лево-правое вращение (LR): вставка в правое поддерево левого потомка.
* Право-левое вращение (RL): вставка в левое поддерево правого потомка.

### **Удаление**

1. Удаление узла происходит так же, как в обычном бинарном дереве поиска:
   * Если удаляемый узел — лист, его просто удаляют.
   * Если у узла один потомок, удаляется узел с подменой потомка.
   * Если у узла два потомка, ищется минимальный элемент из правого поддерева, и происходит замена.
2. Проверяется баланс узлов на пути от удалённого элемента к корню, при необходимости выполняются вращения.

### **Поиск**

* Обычный поиск, как в бинарном дереве поиска, с временной сложностью O(log(n)).

## **Цель задачи**

Целью является поддержание сбалансированности дерева после каждой операции, чтобы его высота оставалась логарифмической от числа узлов:

* h=O(log(n))

## **Ограничения**

1. Количество узлов n: 1≤n≤106.
2. Ключи: Все ключи уникальны, представлены целыми числами.
3. Операции: Вставка, удаление и поиск выполняются на сбалансированном дереве.

# **Теоретическое описание алгоритма**

## **Основные понятия**

### **АВЛ-дерево**

АВЛ-дерево — это сбалансированное бинарное дерево поиска (BST), в котором для каждой вершины разница между высотами её левого и правого поддеревьев не превышает 1. Это условие обеспечивает логарифмическую высоту дерева, что ускоряет операции вставки, удаления и поиска.

### **Высота узла**

Высота узла — это длина самого длинного пути от данного узла до его самого глубокого потомка. Узел без потомков (лист) имеет высоту 0.

### **Балансировочный фактор**

Балансировочный фактор узла определяется как разница между высотами его левого и правого поддерева:

* Balance(N)=Height(N.left)−Height(N.right)

Допустимые значения балансировочного фактора:

* Balance(N)∈{−1,0,1}

Если этот фактор выходит за пределы — требуется балансировка дерева.

## **Операции на АВЛ-дереве**

АВЛ-дерево поддерживает три основные операции:

### **Вставка элемента**

Алгоритм вставки выполняется следующим образом:

1. Вставляем элемент, как в обычном бинарном дереве поиска (BST).
2. Обновляем высоты всех узлов на пути от нового узла к корню.
3. Проверяем балансировочный фактор:

* Если разница высот не превышает 1, балансировка не требуется.
* Если балансировочный фактор превышает 1 или меньше -1, выполняется вращение.

Случаи вращения при вставке:

1. Правое вращение (LL) — левая ветвь длиннее на 2 узла, а новый элемент вставлен в левое поддерево левого потомка.
2. Левое вращение (RR) — правая ветвь длиннее на 2 узла, а новый элемент вставлен в правое поддерево правого потомка.
3. Левое-правое вращение (LR) — левое поддерево длиннее на 2 узла, а новый элемент вставлен в правое поддерево левого потомка.
4. Правое-левое вращение (RL) — правое поддерево длиннее на 2 узла, а новый элемент вставлен в левое поддерево правого потомка.

### **Удаление элемента**

Удаление узла в АВЛ-дереве выполняется аналогично удалению в бинарном дереве поиска:

1. Если удаляется лист — он просто удаляется.
2. Если у узла один потомок — узел заменяется потомком.
3. Если у узла два потомка:

* Ищется минимальный элемент в правом поддереве.
* Заменяется удаляемый узел найденным элементом.
* Выполняется удаление найденного узла.

После удаления выполняется проверка балансировочного фактора на каждом узле, начиная с родителя удалённого узла. Если баланс нарушен, выполняются вращения.

### **Поиск элемента**

Алгоритм поиска в АВЛ-дереве аналогичен поиску в бинарном дереве поиска:

* Сравниваем ключ с текущим узлом.
* Если ключ меньше, спускаемся в левое поддерево.
* Если ключ больше, спускаемся в правое поддерево.
* Если ключ равен, элемент найден.

## **Балансировка дерева: Операции вращения**

**Правое вращение (LL)**

* Выполняется, если левое поддерево стало слишком длинным.

**Левое вращение (RR)**

* Выполняется, если **правое поддерево** стало слишком длинным.

**Лево-правое вращение (LR)**

* Выполняется, если левое поддерево стало длиннее из-за элемента, вставленного в правое поддерево левого потомка.

1. Сначала выполняется левое вращение в левом поддереве.
2. Затем выполняется правое вращение на корне.

**Право-левое вращение (RL)**

* Выполняется, если правое поддерево стало длиннее из-за элемента, вставленного в левое поддерево правого потомка.
  + Сначала выполняется левое вращение в левом поддереве.
  + Затем выполняется левое вращение на корне.

## **Оценка высоты АВЛ-дерева**

Для дерева с n узлами высота АВЛ-дерева h удовлетворяет рекурсивной формуле:

* h≤1.44⋅log2(n+2)−0.328

Приближённое значение высоты:

* h≈O(log(n))

# **Характеристика алгоритма**

## **Основные операции в АВЛ-дереве**

Основные операции на АВЛ-дереве включают:

* **Вставка элемента**
* **Удаление элемента**
* **Поиск элемента**
* **Балансировка дерева (вращения)**

## **Временная сложность операций**

### **Поиск элемента**

**Описание:**  
Поиск элемента в АВЛ-дереве выполняется аналогично поиску в обычном бинарном дереве поиска (BST):

1. Сравниваем ключ с корнем:

* Если ключ меньше, переходим в левое поддерево.
* Если ключ больше, переходим в правое поддерево.
* Если ключ равен, элемент найден.

1. Продолжаем рекурсивно, пока не найдём ключ или не дойдём до пустого узла.

**Временная сложность:**  
В худшем случае нам придётся пройти от корня дерева до самого глубокого узла. Высота АВЛ-дерева равна O(log(n)), поэтому:

* Tпоиск=O(log(n))

### **Вставка элемента**

**Описание:**

1. Элемент вставляется как в обычное бинарное дерево поиска (BST).
2. Обновляются высоты узлов на пути от места вставки к корню.
3. Проверяется баланс каждого узла на пути к корню.
4. Если балансировочный фактор узла выходит за пределы [−1,1], выполняется одно или два вращения (правое, левое, двойное вращение).

**Временная сложность:**

* Поиск позиции для вставки: O(log(n)).
* Обновление высот узлов: O(log(n)), так как мы обновляем высоты на пути от места вставки к корню.
* Балансировка дерева: Одно или два вращения, которые выполняются за O(1).

**Итоговая сложность вставки:**

* Tвставка=O(log(n))

### **Удаление элемента**

**Описание:**  
Удаление узла аналогично удалению из BST с дополнительной балансировкой:

1. Найти удаляемый узел.
2. Удалить его, соблюдая правила BST.
3. Обновить высоты узлов на пути к корню.
4. Выполнить балансировку для всех узлов на пути к корню.

**Временная сложность:**

* Поиск узла: O(log(n))
* Удаление узла: O(log(n))
* Балансировка: Выполняется максимум два вращения на каждом уровне, что также укладывается в O(log(n)).

**Итоговая сложность удаления:**

* Tудаление=O(log(n))

### **Балансировка (вращения)**

АВЛ-дерево поддерживает баланс за счёт следующих операций:

* Правое вращение (LL) — O(1)
* Левое вращение (RR) — O(1)
* Лево-правое вращение (LR) — O(1)
* Право-левое вращение (RL) — O(1)

Так как вращения выполняются локально (на двух или трёх узлах), их сложность составляет:

* Tвращение=O(1)

## **Пространственная сложность**

### **Память на хранение дерева**

АВЛ-дерево хранит nnn узлов, где каждый узел содержит:

* Ключ элемента: O(1)
* Ссылки на левого и правого потомков: O(1)
* Высоту узла: O(1)

Итого, для каждого узла требуется:

* Sузел=O(1)

Таким образом, общая пространственная сложность для хранения всех узлов:

* Sхранение=O(n)

### **Память на поддержку операций**

Дополнительная память используется для поддержки следующих структур:

* **Рекурсивные вызовы:** при выполнении операций вставки, удаления, поиска используется стек вызовов глубиной O(log(n))).
* **Вспомогательные переменные:** для хранения временных узлов при выполнении вращений.

Таким образом, дополнительная память:

* Sдоп=O(log(n))

# **Сравнительный анализ алгоритма с аналогичными**

АВЛ-дерево — это самобалансирующееся бинарное дерево поиска (BST), обеспечивающее логарифмическую сложность операций вставки, удаления и поиска. Для сравнения с АВЛ-деревом рассмотрим его аналоги: красно-черное дерево, бинарное дерево поиска (BST), Treap (сдвигающаяся куча) и B-дерево.

## **АВЛ-дерево (AVL Tree)**

**Преимущества:**

* Строгая балансировка: Обеспечивает минимальную высоту дерева — h=O(log(n)).
* Оптимальные операции: Вставка, удаление и поиск выполняются за O(log(n)).
* Гарантированная производительность: Высота дерева всегда остаётся минимальной.

**Недостатки:**

* Сложность реализации: Требуются сложные операции вращения.
* Частые вращения: При вставке и удалении выполняется больше вращений, чем в других структурах данных.
* Память: Требуется дополнительная память для хранения высоты узлов.

## **Красно-черное дерево (Red-Black Tree)**

**Преимущества:**

* Балансировка с меньшим числом вращений: Выполняется меньше операций балансировки по сравнению с АВЛ-деревом.
* Гибкость: Высота дерева h≤2log(n), что ненамного хуже, чем в АВЛ-дереве.
* Простота реализации: Реализовать проще, чем АВЛ-дерево.

**Недостатки:**

* Менее сбалансированное дерево: Высота дерева может быть немного больше, чем у АВЛ-дерева.
* Уменьшенная эффективность поиска: В среднем поиск в красно-черном дереве чуть медленнее из-за более высокой высоты дерева.

## **Бинарное дерево поиска (BST)**

**Преимущества:**

* Простота реализации: Требуется минимальное количество кода для создания.
* Гибкость: Поддерживает любые типы данных и легко расширяется.

**Недостатки:**

* Нет балансировки: Дерево может выродиться в список с высотой O(n).
* Неоптимальное время выполнения: Все операции имеют сложность O(n) в худшем случае.
* Ненадёжность: Дерево становится неэффективным при неравномерных вставках.

## **Treap (Tree + Heap)**

**Преимущества:**

* Простота балансировки: Использование случайных приоритетов обеспечивает автоматическую балансировку.
* Гибкость: Обеспечивает операции вставки, удаления и поиска за O(log(n)).
* Лёгкость реализации: Легче реализовать, чем АВЛ-дерево.

**Недостатки:**

* Зависимость от случайности: Дерево сильно зависит от качества генерации случайных чисел.
* Непредсказуемая высота: В худшем случае дерево может стать несбалансированным.
* Меньшая надёжность: Подходит только для сценариев, где допустимы случайные структуры.

## **B-дерево (B-Tree)**

**Преимущества:**

* Минимальная высота: Дерево остаётся низким даже при большом количестве узлов, обеспечивая h=O(log(n)).
* Эффективность дисковых операций: Дерево минимизирует количество операций ввода-вывода благодаря высокой ветвистости.
* Использование в базах данных: Широко используется в СУБД, файловых системах и поисковых системах.

**Недостатки:**

* Сложность реализации: Требует управления большим количеством указателей и узлов.
* Вставка и удаление: Более сложные алгоритмы вставки и удаления по сравнению с другими деревьями.
* Дополнительные ресурсы: Требуется больше памяти для управления индексами и поддеревьями.

# **Перечень инструментов для реализации**

## **Языки программирования**

**1. Java**

**Преимущества:**

* Надёжность и безопасность.
* Широкий выбор стандартных библиотек.
* Поддержка многопоточности и платформенной независимости.

**Недостатки:**

* Вербозность кода (большие объёмы кода).
* Медленное прототипирование.

**2. Python**

**Преимущества:**

* Простота синтаксиса, быстрое прототипирование.
* Обширная стандартная библиотека.
* Широкая поддержка инструментов для анализа данных и визуализации.

**Недостатки:**

* Меньшая производительность по сравнению с Java и JavaScript.
* Динамическая типизация, возможные ошибки на этапе выполнения.

**3. JavaScript**

**Преимущества:**

* Широкое использование в веб-разработке.
* Отличная поддержка визуализации благодаря библиотекам D3.js, Vis.js.
* Возможность работы как на клиентской, так и на серверной стороне.

**Недостатки:**

* Отсутствие строгой типизации.
* Меньшая поддержка стандартных структур данных.

## **Библиотеки и фреймворки**

**1. Java**

* **JGraphT:** Библиотека для работы с графами и деревьями. Поддерживает алгоритмы поиска, построение и визуализацию деревьев.
* **Java Collections Framework (JCF):** Встроенные структуры данных TreeMap, TreeSet, обеспечивающие функциональность сбалансированных деревьев.
* **JavaFX:** Фреймворк для создания графических приложений с поддержкой визуализации деревьев.

**2. Python**

* **SortedContainers:** Библиотека для работы с отсортированными структурами данных.
* **AnyTree:** Универсальная библиотека для работы с деревьями.
* **Binarytree:** Библиотека для создания, тестирования и визуализации бинарных деревьев.

**3. JavaScript**

* **D3.js:** Мощная библиотека для работы с данными, графиками и диаграммами. Поддерживает визуализацию деревьев.
* **Vis.js:** Лёгкая библиотека для построения графов и деревьев с интерактивными элементами.
* **Cytoscape.js:** Интерактивная библиотека для работы с графами и сетями.

## **Инструменты для тестирования и отладки**

**1. Java**

* **JUnit:** Стандартный инструмент для модульного тестирования Java-кода.
* **IntelliJ IDEA Debugger / Eclipse Debugger:** Интегрированные отладчики популярных IDE.

**2. Python**

* **unittest:** Встроенный модуль для модульного тестирования Python-кода.
* **pytest:** Расширенный фреймворк для тестирования сложных сценариев.
* **PDB (Python Debugger):** Встроенный отладчик для пошагового выполнения кода.
* **PyCharm Debugger:** Визуальный отладчик с поддержкой точек останова и анализа стека вызовов.

**3. JavaScript**

* **Jest:** Фреймворк для модульного тестирования JavaScript-кода.
* **Mocha + Chai:** Гибкий инструмент для тестирования асинхронного кода.
* **Google Chrome DevTools:** Консоль разработчика для пошагового выполнения кода и анализа ошибок в браузере.

## **Инструменты для визуализации**

**1. Java**

* **Graphviz (через JGraphT):** Визуализация графов и деревьев через генерацию изображений.
* **JavaFX Canvas API:** Рисование деревьев и графов с помощью интерфейса JavaFX.

**2. Python**

* **Matplotlib:** Библиотека для построения графиков и деревьев.
* **NetworkX:** Библиотека для построения и визуализации графов и сетей.
* **Graphviz (PyGraphviz):** Визуализация деревьев в формате DOT.
* **Binarytree:** Отображение бинарных деревьев в текстовом и графическом формате.

**3. JavaScript**

* **D3.js:** Генерация интерактивных визуализаций с использованием SVG, Canvas и WebGL.
* **Vis.js:** Генерация деревьев и графов с поддержкой взаимодействия с пользователем.
* **Cytoscape.js:** Мощная библиотека для построения интерактивных графов и сетей.

# **Описание реализации и процесса тестирования на языке Java**

## **Код алгоритма:**

// Класс узла дерева  
class Node {  
 int key, height;  
 Node left, right;  
  
 // Конструктор узла  
 Node(int key) {  
 this.key = key;  
 height = 1; // Начальная высота узла  
 }  
}

// Основной класс АВЛ-дерева  
public class AVLTree {  
  
 private Node root;  
  
 // Получение высоты узла  
 int height(Node N) {  
 return (N == null) ? 0 : N.height;  
 }  
  
 // Получение балансировочного фактора узла  
 int getBalance(Node N) {  
 return (N == null) ? 0 : height(N.left) - height(N.right);  
 }  
  
 // Правое вращение  
 Node rotateRight(Node y) {  
 Node x = y.left;  
 Node T2 = x.right;  
  
 // Выполнение вращения  
 x.right = y;  
 y.left = T2;  
  
 // Обновление высоты  
 y.height = Math.*max*(height(y.left), height(y.right)) + 1;  
 x.height = Math.*max*(height(x.left), height(x.right)) + 1;  
  
 return x; // Новый корень  
 }  
  
 // Левое вращение  
 Node rotateLeft(Node x) {  
 Node y = x.right;  
 Node T2 = y.left;  
  
 // Выполнение вращения  
 y.left = x;  
 x.right = T2;  
  
 // Обновление высоты  
 x.height = Math.*max*(height(x.left), height(x.right)) + 1;  
 y.height = Math.*max*(height(y.left), height(y.right)) + 1;  
  
 return y; // Новый корень  
 }  
  
 // Вставка ключа  
 Node insert(Node node, int key) {  
 // Выполнение обычной вставки  
 if (node == null) 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; // Дубликаты ключей не допускаются  
  
 // Обновление высоты текущего узла  
 node.height = 1 + Math.*max*(height(node.left), height(node.right));  
  
 // Проверка балансировки узла  
 int balance = getBalance(node);  
  
 // Случаи дисбаланса:  
  
 // LL - Левое левое вращение  
 if (balance > 1 && key < node.left.key)  
 return rotateRight(node);  
  
 // RR - Правое правое вращение  
 if (balance < -1 && key > node.right.key)  
 return rotateLeft(node);  
  
 // LR - Левое правое вращение  
 if (balance > 1 && key > node.left.key) {  
 node.left = rotateLeft(node.left);  
 return rotateRight(node);  
 }  
  
 // RL - Правое левое вращение  
 if (balance < -1 && key < node.right.key) {  
 node.right = rotateRight(node.right);  
 return rotateLeft(node);  
 }  
  
 return node; // Возврат сбалансированного узла  
 }  
  
 // Поиск ключа в дереве  
 boolean search(Node node, int key) {  
 if (node == null) return false;  
 if (key == node.key) return true;  
 if (key < node.key) return search(node.left, key);  
 return search(node.right, key);  
 }  
  
 // Удаление узла  
 Node deleteNode(Node root, int key) {  
 if (root == null) return root;  
  
 // Стандартное удаление в BST  
 if (key < root.key)  
 root.left = deleteNode(root.left, key);  
 else if (key > root.key)  
 root.right = deleteNode(root.right, key);  
 else {  
 // Узел с одним или без потомков  
 if ((root.left == null) || (root.right == null)) {  
 Node temp = (root.left != null) ? root.left : root.right;  
  
 if (temp == null) {  
 root = null;  
 } else {  
 root = temp;  
 }  
 } else {  
 // Узел с двумя потомками  
 Node temp = getMinValueNode(root.right);  
 root.key = temp.key;  
 root.right = deleteNode(root.right, temp.key);  
 }  
 }  
  
 if (root == null) return root;  
  
 // Обновление высоты текущего узла  
 root.height = 1 + Math.*max*(height(root.left), height(root.right));  
  
 // Проверка балансировки  
 int balance = getBalance(root);  
  
 // Случаи дисбаланса:  
  
 // LL  
 if (balance > 1 && getBalance(root.left) >= 0)  
 return rotateRight(root);  
  
 // LR  
 if (balance > 1 && getBalance(root.left) < 0) {  
 root.left = rotateLeft(root.left);  
 return rotateRight(root);  
 }  
  
 // RR  
 if (balance < -1 && getBalance(root.right) <= 0)  
 return rotateLeft(root);  
  
 // RL  
 if (balance < -1 && getBalance(root.right) > 0) {  
 root.right = rotateRight(root.right);  
 return rotateLeft(root);  
 }  
  
 return root;  
 }  
  
 // Минимальный узел в дереве  
 Node getMinValueNode(Node node) {  
 Node current = node;  
 while (current.left != null)  
 current = current.left;  
 return current;  
 }  
  
 // Вставка ключа (вызов)  
 public void insert(int key) {  
 root = insert(root, key);  
 }  
  
 // Удаление ключа (вызов)  
 public void delete(int key) {  
 root = deleteNode(root, key);  
 }  
  
 // Поиск ключа (вызов)  
 public boolean search(int key) {  
 return search(root, key);  
 }  
  
 // Обход дерева  
 void inOrderTraversal(Node node) {  
 StringBuilder output = new StringBuilder();  
 inOrderTraversal(node, output);  
 System.*out*.println(output.toString());  
 }  
  
 // Основной метод для тестов  
 void inOrderTraversal(Node node, StringBuilder output) {  
 if (node != null) {  
 inOrderTraversal(node.left, output);  
 output.append(node.key).append(" ");  
 inOrderTraversal(node.right, output);  
 }  
 }  
}

public class Runner {  
 public static void main(String[] args) {  
 AVLTree tree = new AVLTree();  
 // Вставка узлов  
 tree.insert(10);  
 tree.insert(20);  
 tree.insert(30);  
 tree.insert(40);  
 tree.insert(50);  
 tree.insert(25);  
  
 System.*out*.println("Обход дерева (по возрастанию):");  
 tree.inOrderTraversal(tree.root);  
  
 System.*out*.println("\nПоиск элемента 30: " + tree.search(30));  
  
 // Удаление узла  
 tree.delete(30);  
 System.*out*.println("После удаления элемента 30:");  
 tree.inOrderTraversal(tree.root);  
 }  
}

### **Результат работы программы:**

Обход дерева (по возрастанию):

10 20 25 30 40 50

Поиск элемента 30: true

После удаления элемента 30:

10 20 25 40 50

Process finished with exit code 0

## **Тестирование алгоритма:**

import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
  
public class AVLTreeTest {  
  
 @Test  
 public void testInsertionAndInOrderTraversal() {  
 AVLTree tree = new AVLTree();  
  
 // Вставляем элементы  
 tree.insert(10);  
 tree.insert(20);  
 tree.insert(30);  
 tree.insert(25);  
  
 // Создаём StringBuilder для сбора результата  
 StringBuilder output = new StringBuilder();  
  
 // Вызываем обход дерева  
 tree.inOrderTraversal(tree.root, output);  
  
 // Проверяем правильность обхода  
 *assertEquals*("10 20 25 30 ", output.toString());  
 }  
  
 // 2. Тест на удаление элемента  
 @Test  
 public void testDeletion() {  
 AVLTree tree = new AVLTree();  
  
 // Вставка узлов  
 tree.insert(10);  
 tree.insert(20);  
 tree.insert(30);  
 tree.insert(40);  
 tree.insert(50);  
 tree.insert(25);  
  
 // Удаление элемента 30  
 tree.delete(30);  
  
 // Проверка обхода дерева после удаления  
 StringBuilder output = new StringBuilder();  
 tree.inOrderTraversal(tree.root, output);  
 *assertEquals*("10 20 25 40 50 ", output.toString());  
 }  
  
 // 3. Тест на поиск существующего элемента  
 @Test  
 public void testSearchExistingElement() {  
 AVLTree tree = new AVLTree();  
  
 // Вставка узлов  
 tree.insert(10);  
 tree.insert(20);  
 tree.insert(30);  
  
 // Проверка поиска существующего элемента  
 *assertTrue*(tree.search(20));  
 }  
  
 // 4. Тест на поиск несуществующего элемента  
 @Test  
 public void testSearchNonExistingElement() {  
 AVLTree tree = new AVLTree();  
  
 // Вставка узлов  
 tree.insert(10);  
 tree.insert(20);  
 tree.insert(30);  
  
 // Проверка поиска несуществующего элемента  
 *assertFalse*(tree.search(40));  
 }  
  
 // 5. Тест на удаление корня дерева  
 @Test  
 public void testDeleteRoot() {  
 AVLTree tree = new AVLTree();  
  
 // Вставка узлов  
 tree.insert(10);  
 tree.insert(20);  
 tree.insert(30);  
  
 // Удаление корня  
 tree.delete(20);  
  
 // Проверка обхода дерева  
 StringBuilder output = new StringBuilder();  
 tree.inOrderTraversal(tree.root, output);  
 *assertEquals*("10 30 ", output.toString());  
 }  
}

### **Результаты тестов:**

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

Все тесты завершились успешно

# **Заключение**

АВЛ-дерево — это эффективная структура данных, обеспечивающая балансировку бинарных деревьев поиска. Благодаря поддержанию сбалансированности после каждой операции, оно гарантирует логарифмическую сложность операций вставки, удаления и поиска, что делает его одним из наиболее оптимальных вариантов для реализации словарей, индексов баз данных и файловых систем.

Основные преимущества АВЛ-дерева включают строгую балансировку, минимальную высоту и предсказуемую производительность операций. Однако эти свойства достигаются за счёт сложной реализации и дополнительных вычислительных затрат на обновление высот узлов и выполнение вращений.

АВЛ-дерево является одним из классических примеров самобалансирующихся структур данных, демонстрируя, как алгоритмические принципы могут использоваться для обеспечения высокой эффективности в реальных приложениях. Его сравнение с альтернативными структурами, такими как красно-черные деревья, B-деревья и сдвигающиеся кучи, показывает, что выбор структуры зависит от конкретных требований системы, таких как скорость операций, объем хранимых данных и сложность реализации.

Таким образом, изучение алгоритмов работы АВЛ-дерева позволяет лучше понять принципы структур данных, их поведение в условиях изменения данных и способы эффективной реализации алгоритмов для управления большими объемами информации.