**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра систем автоматизированного проектирования**

отчет

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3353 |  | Шинкарь К. Д. |
| Преподаватель |  | Пестерев Д.О. |

Санкт-Петербург

2024

Оглавление

[**Теоретическая часть** 4](#_Toc184165689)

[**Цель работы:** 4](#_Toc184165690)

[**AVL-деревья** 4](#_Toc184165691)

[**1.** **Определение AVL-дерева:** 4](#_Toc184165692)

[**2.** **Алгоритм вставки элемента в AVL-дерево с последующей балансировкой:** 5](#_Toc184165693)

[Вставка элемента: 5](#_Toc184165694)

[Проверка баланса: 5](#_Toc184165695)

[Балансировка дерева: 5](#_Toc184165696)

[Обновление высоты узлов: 6](#_Toc184165697)

[**3.** **Алгоритм вставки в AVL-дерево с последующей балансировкой:** 6](#_Toc184165698)

[Удаление элемента: 6](#_Toc184165699)

[Проверка баланса: 6](#_Toc184165700)

[Балансировка дерева 7](#_Toc184165701)

[Обновление высоты узлов 7](#_Toc184165702)

[**4.** **Верхняя оценка высоты AVL-дерева:** 8](#_Toc184165703)

[Рекуррентное соотношение 8](#_Toc184165704)

[Условие остановки 11](#_Toc184165705)

[Выражение высоты через количество узлов 11](#_Toc184165706)

[Итоговая оценка 12](#_Toc184165707)

[**Красно-чёрные деревья:** 13](#_Toc184165708)

[**1.** **Определение красно-чёрного дерева:** 13](#_Toc184165709)

[**2.** **Алгоритм вставки элемента в красно-чёрное дерево с последующей балансировкой.** 13](#_Toc184165710)

[Вставка элемента: 13](#_Toc184165711)

[Балансировка дерева: 14](#_Toc184165712)

[Типы поворотов и перекраски: 14](#_Toc184165713)

[Обновление цветов: 15](#_Toc184165714)

[Особое внимание: 15](#_Toc184165715)

[**3.** **Алгоритм удаления элемента из красно-чёрного дерева с последующей балансировкой.** 16](#_Toc184165716)

[Поиск узла для удаления: 16](#_Toc184165717)

[Удаление узла: 16](#_Toc184165718)

[Балансировка дерева после удаления элемента: 16](#_Toc184165719)

[Особые случаи и исправления: 16](#_Toc184165720)

[Основные случаи дисбаланса и их решение: 17](#_Toc184165721)

[Итог: 17](#_Toc184165722)

[**4.** **Верхняя оценка высоты красно-черного дерева.** 18](#_Toc184165723)

[Максимальная высота: 18](#_Toc184165724)

[Рекуррентное соотношение: 18](#_Toc184165725)

[Минимальное количество узлов n(h): 18](#_Toc184165726)

[Выражение высоты через количество узлов: 20](#_Toc184165727)

[Итоговая оценка: 20](#_Toc184165728)

[**Асимптотическая сложность бинарного дерева поиска (BST).** 21](#_Toc184165729)

[1. Лучший случай 21](#_Toc184165730)

[2. Средний случай 22](#_Toc184165731)

[3. Худший случай 23](#_Toc184165732)

[Итоговые оценки: 23](#_Toc184165733)

[**Асимптотическая сложность AVL-дерева.** 24](#_Toc184165734)

[1. Лучший случай 24](#_Toc184165735)

[2. Средний случай 25](#_Toc184165736)

[3. Худший случай 25](#_Toc184165737)

[Итоговые оценки: 26](#_Toc184165738)

[**Асимптотическая сложность красно-черного дерева (Red-Black Tree).** 27](#_Toc184165739)

[1. Лучший случай 27](#_Toc184165740)

[2. Средний случай 27](#_Toc184165741)

[3. Худший случай 28](#_Toc184165742)

[Итоговые оценки: 29](#_Toc184165743)

[**Теоретические графики.** 30](#_Toc184165744)

[График асимптотической оценки 3 случаев бинарного дерева: 30](#_Toc184165745)

[График асимптотической оценки 3 случаев AVL-дерева: 31](#_Toc184165746)

[График асимптотической оценки 3 случаев RB-дерева: 31](#_Toc184165747)

[Общий график: 32](#_Toc184165748)

[**Практическая часть** 33](#_Toc184165749)

[**Зависимость высоты бинарного дерева от количества ключей.** 33](#_Toc184165750)

[1. Значение ключа – случайная величина: 33](#_Toc184165751)

[2. Значения ключей распределены равномерно: 34](#_Toc184165752)

[3. Значения ключей распределены в порядке возрастания: 35](#_Toc184165753)

[4. Общий график: 36](#_Toc184165754)

[**Зависимость высоты AVL-дерева от количества ключей.** 37](#_Toc184165755)

[1. Значения ключей монотонно возрастают. 37](#_Toc184165756)

[2. Значение ключа – случайная величина. 38](#_Toc184165757)

[3. Общий график: 39](#_Toc184165758)

[**Зависимость высоты красно-чёрного дерева от количества ключей.** 40](#_Toc184165759)

[1. Значения ключей монотонно возрастают. 40](#_Toc184165760)

[2. Значение ключа – случайная величина. 41](#_Toc184165761)

[3. Общий график. 42](#_Toc184165762)

[**Объединенный практический график.** 43](#_Toc184165763)

[1. Объединенный график. 43](#_Toc184165764)

[**Реализация обходов деревьев.** 44](#_Toc184165765)

[1. BST. 44](#_Toc184165766)

[2. AVL. 47](#_Toc184165767)

[3. RB. 50](#_Toc184165768)

[**Раздел со всем кодом.** 52](#_Toc184165769)

[**Код деревьев.** 52](#_Toc184165770)

[1. BST. 52](#_Toc184165771)

[2. AVL. 58](#_Toc184165772)

[3. RB. 65](#_Toc184165773)

[**Код графиков.** 74](#_Toc184165774)

[Theory BST. 74](#_Toc184165775)

[Theory AVL. 75](#_Toc184165776)

[Theory RB. 76](#_Toc184165777)

[Theory all. 77](#_Toc184165778)

[Random key BST. 80](#_Toc184165779)

[Sorted keys BST best. 82](#_Toc184165780)

[Sorted keys BST worst. 84](#_Toc184165781)

[BST all practice. 86](#_Toc184165782)

[AVL grow. 89](#_Toc184165783)

[AVL random. 91](#_Toc184165784)

[AVL all. 94](#_Toc184165785)

[RB grow. 98](#_Toc184165786)

[RB random. 101](#_Toc184165787)

[RB all. 106](#_Toc184165788)

[Practice all. 111](#_Toc184165789)

[**Ссылка на GitHub.** 118](#_Toc184165790)

# **Теоретическая часть**

# **Цель работы:**

Реализация самобалансирующихся деревьев поиска и экспериментальная проверка оценок высоты данных деревьев.

# **AVL-деревья**

# **Определение AVL-дерева:**

АВЛ-дерево, или AVL tree, — древовидная структура данных с быстрым доступом к информации. Она представляет собой бинарное дерево — иерархическую схему из вершин и путей между ними, где у одной вершины может быть не более двух потомков. Для любого узла высота левого и правого поддеревьев различается не более чем на 1.

# **Алгоритм вставки элемента в AVL-дерево с последующей балансировкой:**

### Вставка элемента:

Выполнить стандартную операцию вставки в двоичное дерево поиска:

1. Сравнить новый элемент с текущим узлом.

2. Рекурсивно перейти в левое или правое поддерево.

3. Добавить элемент в подходящее место.

После вставки вернуться вверх по рекурсии и обновить высоту всех узлов, начиная от листа вверх по дереву.

### Проверка баланса:

Для каждого узла вычислить баланс-фактор:

Баланс-фактор = Высота левого поддерева − Высота правого поддерева

Если баланс-фактор узла выходит за пределы [−1, 1] то дерево нуждается в балансировке.

### Балансировка дерева:

Определить тип дисбаланса:

* 1. Левый-левый (LL): Дисбаланс в левом поддереве левого потомка.  
     Решение: Выполнить правый поворот.
  2. Правый-правый (RR): Дисбаланс в правом поддереве правого потомка.  
     Решение: Выполнить левый поворот.
  3. Левый-правый (LR): Дисбаланс в правом поддереве левого потомка.  
     Решение: Выполнить левый поворот на левом поддереве, затем правый поворот.
  4. Правый-левый (RL): Дисбаланс в левом поддереве правого потомка.  
     Решение: Выполнить правый поворот на правом поддереве, затем левый поворот.

### Обновление высоты узлов:

После балансировки обновить высоты всех узлов, участвовавших в поворотах.

# **Алгоритм вставки в AVL-дерево с последующей балансировкой:**

### Удаление элемента:

Найти удаляемый узел, как в обычном двоичном дереве поиска.

Если узел:

1. **Лист**: Просто удалить его.
2. **Имеет одного потомка**: Удалить узел, заменив его потомком.
3. **Имеет двух потомков**: Найти наименьший элемент в правом поддереве (иногда называют инфиксным преемником), заменить удаляемый узел этим элементом, затем рекурсивно удалить этот элемент.

### **Проверка баланса**:

После удаления вернуться вверх по рекурсии и обновить высоты всех узлов.

Вычислить баланс-фактор для каждого узла. Если баланс-фактор выходит за пределы [−1, 1], дерево нуждается в балансировке.

### **Балансировка дерева**

Определить тип дисбаланса (LL, RR, LR, RL) так же, как при вставке, и выполнить соответствующие повороты.

### **Обновление высоты узлов**

После каждого поворота обновить высоты всех затронутых узлов.

# **Верхняя оценка высоты AVL-дерева:**

h: высота AVL-дерева.

n(h): минимальное количество узлов в AVL-дереве высоты h.

### **Рекуррентное соотношение**

Минимальное количество узлов n(h) в AVL-дереве высоты h определяется следующим образом:

AVL-дерево высоты h состоит из корня и двух поддеревьев:

Одного поддерева высоты h − 1.

Другого поддерева высоты h − 2 (чтобы разница высот поддеревьев была не больше 1).

Это приводит к следующему рекуррентному соотношению:

n(h) = 1 + n(h − 1) + n(h − 2),

n(0) = 1 (дерево с высотой 0 содержит только корень (то есть корневой узел)).

n(1) = 2 (дерево с высотой 1 содержит корень и одного потомка).

#### **Развертывание рекуррентного выражения**

Цель: найти оценку n(h) для больших значений h.

Чтобы понять поведение n(h), развернем рекуррентное соотношение:

n(h) = 1 + n(h − 1) + n(h − 2).

Подставим n(h − 1) и n(h − 2) из того же уравнения:

n(h) = 1 + (1 + n(h − 2) + n(h − 3)) + n(h − 2).

Упростим:

n(h) = 1 + 1 + n(h − 2) + n(h − 3) + n(h − 2).

Объединим одинаковые слагаемые:

n(h) = 2 + 2 ⋅ n(h − 2) + n(h − 3).

Заметим, что каждый следующий шаг добавляет ещё меньшие члены (например, n(h − 3), n(h − 4) и т.д.), что делает n(h) в основном определяемым n(h − 2) и n(h − 3), а вклад более глубоких членов n(h − 4), n(h − 5) и так далее становится всё менее значимым.

n(h − 4) ≈ ​ ⋅ n(h − 2),

n(h − 6) ≈ ​⋅ n(h − 2), …

#### 2. **Минимальная оценка**

Можно увидеть, что n(h) быстро растет, так как каждое добавление включает в себя минимум удвоение предыдущих минимальных узлов. Для оценки развернем рекурсию до основания (например, n(0) = 1):

n(h) > 2⋅n(h − 2).

Докажем неравенство n(h) > 2⋅n(h − 2) методом математической индукции.

База индукции:

Для h = 2:

n(2) = 1 + n(1) + n(0) = 1 + 2 + 1 = 4

Для h = 0:

n(0) = 1

Убедимся, что n(2) > 2⋅n(0):

4 > 2 ⋅ 1 = 2

Верно.

Индукционное предположение:

Пусть для некоторой высоты h − 2 выполняется:

n(h − 2) > 2 ⋅ n(h − 4)

Индукционный переход:

Докажем, что:

n(h) > 2 ⋅ n(h − 2)

Используем рекуррентное выражение:

n(h) = 1 + n(h − 1) + n(h − 2)

По предположению индукции:

n(h − 1) > 2 ⋅ n(h − 3),

n(h − 2) > 2 ⋅ n(h − 4).

Тогда:

n(h) = 1 + n(h − 1) + n(h − 2) > 1 + 2 ⋅ n(h − 3) + 2 ⋅ n(h − 4)

Заметим, что n(h − 3) + n(h − 4) соответствует n(h − 2), поэтому итог:

n(h) > 2 ⋅ n(h − 2)

Подставим n(h - 2):

n(h − 2) > 2 ⋅ n(h − 4)

Развернем это соотношение дальше:

n(h) > 2 ⋅ n(h − 2) > 2 ⋅ (2 ⋅ n(h − 4))

2 ⋅ (2 ⋅ n(h − 4)) = 22 ⋅ n(h − 4).

При k-м шаге:

n(h) > 2k ⋅ n(h − 2k).

### **Условие остановки**

Мы разворачиваем рекурсию до тех пор, пока высота не станет равной нулю (h − 2k =0). Отсюда:

k =

Подставим это в общее выражение:

n(h) > 2h/2 ⋅ n(0)

Так как n(0) = 1, окончательно получаем:

n(h) > 2h/2.

### **Выражение высоты через количество узлов**

Пусть n — общее количество узлов в AVL-дереве высоты h.

Тогда, из n(h) > 2h/2, следует:

n ≥ 2h/2.

Применим логарифм к обеим частям:

log2​n ≥

Умножим обе части на 2:

h ≤ 2⋅log2​n.

### **Итоговая оценка**

Полученное выражение показывает, что высота h AVL-дерева растет логарифмически относительно числа узлов n. Это подтверждает, что высота:

h ∈ O (log n)

# **Красно-чёрные деревья:**

# **Определение красно-чёрного дерева:**

**Красно-чёрное дерево** — это сбалансированное двоичное дерево поиска, в котором каждый узел имеет дополнительный атрибут — цвет, который может быть либо красным, либо чёрным. Оно строго сбалансированно благодаря своим особым свойствам:

1. Каждый узел имеет цвет: либо красный, либо чёрный.
2. Корень дерева всегда чёрный.
3. Каждый лист (nil-узел) чёрный. Листья представляют собой NULL-указатели, и они считаются чёрными.
4. Если узел красный, то оба его дочерних узла обязательно чёрные (никаких двух последовательных красных узлов на одном пути).
5. Для любого узла все пути от этого узла до листьев содержат одинаковое количество чёрных узлов.

# **Алгоритм вставки элемента в красно-чёрное дерево с последующей балансировкой.**

### Вставка элемента:

1. **Выполнить стандартную операцию вставки** в двоичное дерево поиска:
   1. Сравнить новый элемент с текущим узлом.
   2. Рекурсивно перейти в левое или правое поддерево.
   3. Добавить элемент в подходящее место как красный узел (новые узлы всегда вставляются красными, чтобы не нарушить высоту дерева).
2. После вставки вернуться вверх по рекурсии и выполнить **проверку на нарушение свойств красно-чёрного дерева**.

После вставки нового элемента возможны нарушения следующих свойств:

1. **Два последовательных красных узла:** Если у нового узла и его родителя красный цвет — это нарушение.
2. **Корень должен быть чёрным:** Если корень становится красным, его необходимо перекрасить в чёрный.

### Балансировка дерева:

Используется серия **поворотов** и **перекрашиваний** для восстановления свойств. Возможны следующие случаи нарушений и способы их исправления:

1. **Дядя — красный (Перекраска):**

Если у нового узла есть красный родитель и красный дядя (брат родителя):

* + 1. Перекрасить родителя и дядю в чёрный.
    2. Перекрасить дедушку в красный.
    3. Продолжить проверку от дедушки вверх по дереву.

1. **Дядя — чёрный или отсутствует (Повороты):**

Определить тип дисбаланса на основе расположения узла, родителя и дедушки:

### Типы поворотов и перекраски:

* + **Левый-левый (LL):**

Новый узел — левый потомок левого потомка дедушки.

**Решение:** Выполнить правый поворот вокруг дедушки и перекрасить.

* + **Правый-правый (RR):**
    1. Новый узел — правый потомок правого потомка дедушки.
    2. **Решение:** Выполнить левый поворот вокруг дедушки и перекрасить.
  + **Левый-правый (LR):**
    1. Новый узел — правый потомок левого потомка дедушки.
    2. **Решение:** Выполнить левый поворот вокруг родителя, затем правый поворот вокруг дедушки и перекраска.
  + **Правый-левый (RL):**
    1. Новый узел — левый потомок правого потомка дедушки.
    2. **Решение:** Выполнить правый поворот вокруг родителя, затем левый поворот вокруг дедушки и перекраска.

### Обновление цветов:

После балансировки узлов обновить цвета:

Новый корень поддерева должен стать чёрным.

Дочерние узлы после поворотов становятся красными.

### Особое внимание:

Корень дерева всегда должен оставаться чёрным.

Все операции выполняются за **O(log n)**, так как дерево остаётся сбалансированным.

# **Алгоритм удаления элемента из красно-чёрного дерева с последующей балансировкой.**

Удаление элемента из красно-чёрного дерева сложнее вставки, так как необходимо не только удалить узел, но и поддерживать все свойства красно-чёрного дерева. Процесс включает стандартное удаление из двоичного дерева поиска с последующей балансировкой и перекраской.

### Поиск узла для удаления:

Выполнить стандартный поиск узла в двоичном дереве поиска.

Если узел имеет двух дочерних потомков, найти минимальный элемент в правом поддереве (или максимальный в левом поддереве) и заменить удаляемый узел на него.

### Удаление узла:

Удаляемый узел может быть:

* 1. **Листом** (не имеет потомков).
  2. Имеет **одного потомка**.
  3. **Два потомка**: Удаляем узел и заменяем его на его преемника (как в стандартном бинарном дереве поиска).

### Балансировка дерева после удаления элемента:

После удаления могут нарушиться свойства красно-чёрного дерева, особенно если удалённый или заменённый узел был чёрным.

### Особые случаи и исправления:

1. **Удаление красного узла:**  
   Если удаляемый узел красный, дополнительных действий не требуется, так как это не влияет на чёрную высоту дерева.
2. **Удаление чёрного узла:**  
   Удаление чёрного узла приводит к нарушению чёрной высоты. Требуется дополнительная балансировка для восстановления свойств.

### **Основные случаи дисбаланса и их решение:**

1. **Узел имеет чёрного брата, и оба его племянника чёрные (Ребалансировка цвета):**

**Решение:** Сделать брата узла красным, а родителя — чёрным. Переместить проблему вверх на родительский узел.

1. **Узел имеет чёрного брата, у которого есть красный племянник:**

Определить тип дисбаланса и выполнить соответствующие повороты и перекраску.

1. **Узел имеет красного брата:**

Выполнить поворот вокруг родителя и перекрасить. Продолжить балансировку на следующем уровне.

1. **Случай, когда брат узла чёрный, а его дальний племянник красный:**

Перекрасить брата и родителя, затем выполнить поворот.

### **Итог:**

Удаление элемента из красно-чёрного дерева требует тщательной балансировки и перекраски узлов для сохранения всех его свойств. Основное внимание уделяется поддержанию чёрной высоты и недопущению последовательных красных узлов.

# **Верхняя оценка высоты красно-черного дерева.**

**Обозначения:**

h: высота красно-черного дерева.

n(h): минимальное количество узлов в красно-черном дереве высоты h.

### **Максимальная высота:**

Чтобы оценить верхнюю границу высоты, следует учитывать, что высота дерева h может быть максимум в два раза больше черной высоты, так как между черными узлами может быть максимум один красный узел.

### Рекуррентное соотношение:

Красно-черное дерево с минимальным количеством узлов будет иметь максимально возможную высоту для данной черной высоты. Обозначим черную высоту через bh (black-height):

* Каждый путь от корня до листа содержит минимум bh черных узлов.
* Максимальная высота красно-черного дерева может достигать 2 ⋅ bh, так как между черными узлами может чередоваться максимум один красный узел.

### Минимальное количество узлов n(h):

Минимальное количество узлов n(bh) при черной высоте bh:

Минимальное количество узлов в дереве с черной высотой bh соответствует полному бинарному дереву, состоящему только из черных узлов. Такое дерево имеет 2bh − 1 узлов. Пояснение ниже:

**Количество узлов в полном бинарном дереве:**  
В полном бинарном дереве высоты bh:

На уровне 0 (корень) — 1 узел.

На уровне 1 — 2 узла.

На уровне 2 — 4 узла.

И так далее.

Общее количество узлов n в таком дереве вычисляется как сумма геометрической прогрессии:

n(bh) = 1 + 2 + 4 + 8 + ⋯ + 2bh − 1

**Сумма геометрической прогрессии:**  
Сумма первых bh членов геометрической прогрессии с первым членом 1 и общим знаменателем 2:

n(bh) = 2bh − 1

2bh – 1 получается по следующим причинам:

* + 1. **Полное заполнение уровней:**  
       Если все узлы черные и каждый уровень заполнен полностью, максимальное количество узлов соответствует 2bh−1. Любой путь от корня до листа содержит ровно bh черных узлов
    2. **Нет красных узлов:**  
       В таком дереве нет красных узлов, поэтому минимальное количество узлов равно количеству черных узлов, необходимых для построения такого дерева.

Таким образом, количество узлов n(h) при высоте h:

n(h) ≥ 2bh − 1

Так как максимальная высота h может быть до 2 ⋅ bh

bh ≥

Подставим bh в выражение для количества узлов:

n(h) ≥ 2h/2 − 1

### Выражение высоты через количество узлов:

Пусть n — общее количество узлов в красно-черном дереве высоты h:

n ≥ 2h/2

Применим логарифм по основанию 2 к обеим частям:

log2n ≥

Умножим обе части на 2:

h ≤ 2log2n

### Итоговая оценка:

Полученное выражение показывает, что высота h красно-черного дерева растет логарифмически относительно числа узлов n:

h ∈ O(log n)

# **Асимптотическая сложность бинарного дерева поиска (BST).**

### **1. Лучший случай**

В лучшем случае дерево сбалансировано, а высота бинарного дерева поиска составляет h = log2(n), где n — количество узлов.

Каждый узел разбивает дерево пополам на каждом уровне рекурсии:

T(n) = T() + O(1)

Уровни дерева: в бинарном дереве на каждом уровне узлы удваиваются:

* Уровень 0 (корень): 1 узел
* Уровень 1: 2 узла
* Уровень 2: 4 узла
* Уровень h: 2h узлов

Применим метод подстановки:

1. На первом уровне: T(n) = T() + O(1)
2. На втором уровне: T() = T() + O(1)

Из геометрической прогрессии: n = 1 + 2 + 4 + ⋯ + 2h

n = 2h + 1 − 1

n ≈ 2h + 1

h ≈ log2​(n) − 1

h = O(log2​(n))

1. Продолжая до листьев, получаем log2(n) уровней.

Таким образом, решение рекуррентного уравнения:

T(n) = O(log n)

**Асимптотика в лучшем случае:** O(log n)

### **2. Средний случай**

В среднем случае дерево поиска не всегда идеально сбалансировано, но имеет случайное распределение данных.

#### **Рекуррентное соотношение:**

Средняя высота BST составляет O(log n), но с учетом случайного порядка вставки:

T(n) = )

**​**: Так как каждый из n элементов может стать корнем с одинаковой вероятностью .

T(k − 1) + T(n − k): Если k-й элемент становится корнем, то высота дерева равна высоте максимального из двух поддеревьев (левого и правого). В данном случае мы рассматриваем оба поддерева и суммируем их высоты.

O(1): Это постоянное время, необходимое для создания нового узла и установки связей с поддеревьями.

Рекуррентное уравнение может быть проанализировано через разложение:

При случайной вставке каждый новый узел в среднем делит дерево на две примерно равные части. Поэтому левое и правое поддеревья содержат примерно узлов.

T(n) ≈ T(​) + T(​) + O(1)

T(n) = 2T(​) + O(1)

T(​​) = 2T(​​) + O(1)

Продолжаем раскладывать до базового случая T(1) = O(1)

Получаем сумму:

T(n) = O(1) + 2O(1) + 4O(1) + ⋯ + nO(1)

Это геометрическая прогрессия, и её сумма пропорциональна log n:

**Асимптотика в среднем случае:** O(log n)

### **3. Худший случай**

В худшем случае дерево поиска становится вырожденным (например, когда элементы вставляются в отсортированном порядке), превращаясь в линейный список. Высота дерева равна n.

Каждая операция требует обхода всех n узлов:

T(n) = T(n − 1) + O(1)

1. T(n) = T(n−1) + O(1)
2. Развернем n раз рекуррентное уравнение: T(n) = O(1) + O(1) + ⋯ + O(1)
3. Итог: T(n) = O(n)

**Асимптотика в худшем случае:** O(n)

### **Итоговые оценки:**

Лучший случай: O(log n)

Средний случай: O(log n)

Худший случай: O(n)

# **Асимптотическая сложность AVL-дерева.**

### **1. Лучший случай**

В лучшем случае AVL-дерево остается идеально сбалансированным после каждой операции вставки или удаления. Высота сбалансированного AVL-дерева составляет h = log2(n), где n — количество узлов.

На каждом уровне узел разбивает дерево на два примерно равных поддерева:

T(n) = T() + O(1)

#### **Структура уровней дерева:**

Уровень 0 (корень): 1 узел

Уровень 1: 2 узла

Уровень 2: 4 узла

Уровень h: 2h узлов

#### **Анализ с помощью метода подстановки:**

На первом уровне: T(n) = T() + O(1)

На втором уровне: T() = T() + O(1)

Общее количество узлов: n = 1 + 2 + 4 + … + 2h (геометрическая прогрессия).  
Сумма: n ≈ 2h + 1 – 1 ⇒ h ≈ log2 n

Таким образом, решение рекуррентного уравнения:

T(n) = O(log n)

**Асимптотика в лучшем случае:** O(log n)

### **2. Средний случай**

#### **Описание:**

В среднем случае AVL-дерево остается сбалансированным благодаря поддержанию фактора баланса после каждой вставки или удаления. Даже при случайном порядке вставки высота остается O(log n).

#### **Рекуррентное соотношение:**

При каждой вставке новый элемент становится корнем с одинаковой вероятностью, и дерево делится на два сбалансированных поддерева:

T(n) = T() + T() + O(1)

T(n) = 2T() + O(1)

#### **Разложение рекуррентного уравнения:**

T() = 2T() + O(1)

Продолжаем до базового случая T(1) = O(1):

T(n) = O(1) + 2O(1) + 4O(1) + … + nO(1)

Это геометрическая прогрессия, сумма которой пропорциональна log n.

**Асимптотика в среднем случае:** O(log n)

### **3. Худший случай**

#### **Описание:**

AVL-дерево специально поддерживает балансировку, поэтому высота остается O(log n) даже в худшем случае. Если после вставки или удаления происходит дисбаланс, выполняются повороты для восстановления баланса.

#### **Рекуррентное соотношение:**

Даже в худшем случае дерево остается сбалансированным, а высота составляет log n:

T(n) = T() + O(1)

Каждая операция вставки или удаления требует O(log n) времени для поиска места вставки и выполнения O(1) операций вращения.

#### **Развертывание рекуррентного уравнения:**

На каждом уровне количество узлов уменьшается вдвое:

T(n) = O(log n)

**Асимптотика в худшем случае:** O(log n)

### **Итоговые оценки:**

**Лучший случай:** O(log n)

**Средний случай:** O(log n)

**Худший случай:** O(log n)

Таким образом, AVL-дерево обеспечивает эффективную работу для всех операций благодаря поддержанию сбалансированности на каждом этапе.

# **Асимптотическая сложность красно-черного дерева (Red-Black Tree).**

### **1. Лучший случай**

В лучшем случае красно-чёрное дерево идеально сбалансировано. Высота чёрных узлов составляет hb, а общая высота дерева h находится в пределах:

h ≤ 2hb+1

Поскольку на каждом пути от корня до листа одинаковое количество чёрных узлов, высота красно-чёрного дерева ограничена:

h = O(log n)

На каждом уровне узел делит дерево на два сбалансированных поддерева:

T(n) = T() + O(1)

#### **Анализ уровней:**

* На уровне i: 2i узлов.
* Общее количество уровней: O(log n).

**Асимптотика в лучшем случае:** O(log n)

### **2. Средний случай**

#### **Описание:**

В среднем случае красно-чёрное дерево сохраняет балансировку, так как после каждой вставки или удаления производится не более двух вращений для восстановления правил дерева. Следовательно, средняя высота остаётся O(log n).

При вставке элемента дерево разбивается на два сбалансированных поддерева:

T(n) = 2T() + O(1)

Каждая операция вставки или удаления требует O(log n) времени на поиск и до двух вращений, что также выполняется за O(1) на каждом уровне.

#### **Разложение рекуррентного уравнения:**

T(n) = O(1) + 2O(1) + 4O(1) + … + nO(1)

Это геометрическая прогрессия, сумма которой пропорциональна log n.

**Асимптотика в среднем случае:** O(log n)

### **3. Худший случай**

Даже в худшем случае красно-чёрное дерево сохраняет балансировку благодаря жёстким правилам цветового распределения. После каждой операции вставки или удаления возможны корректирующие вращения и перекраски, что гарантирует, что высота остаётся O(log n).

На каждом уровне поддеревья содержат не более половины узлов предыдущего уровня:

T(n) = T() + O(1)

#### **Обоснование:**

Максимальное количество чёрных узлов на пути от корня к листу:

hb ≈ log2 (n).

Максимальная высота дерева (с учётом красных узлов): h ≈ 2log2 (n).

Таким образом:

T(n) = O(log n)

**Асимптотика в худшем случае:** O(log n)

### **Итоговые оценки:**

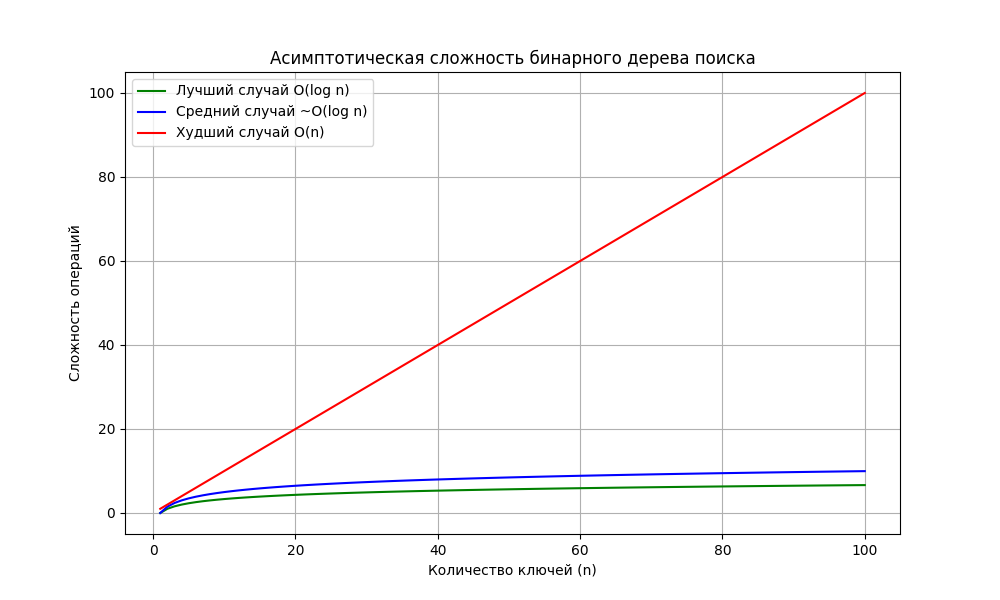
**Лучший случай:** O(log n)

**Средний случай:** O(log n)

**Худший случай:** O(log n)

# **Теоретические графики.**

### **График асимптотической оценки 3 случаев бинарного дерева:**

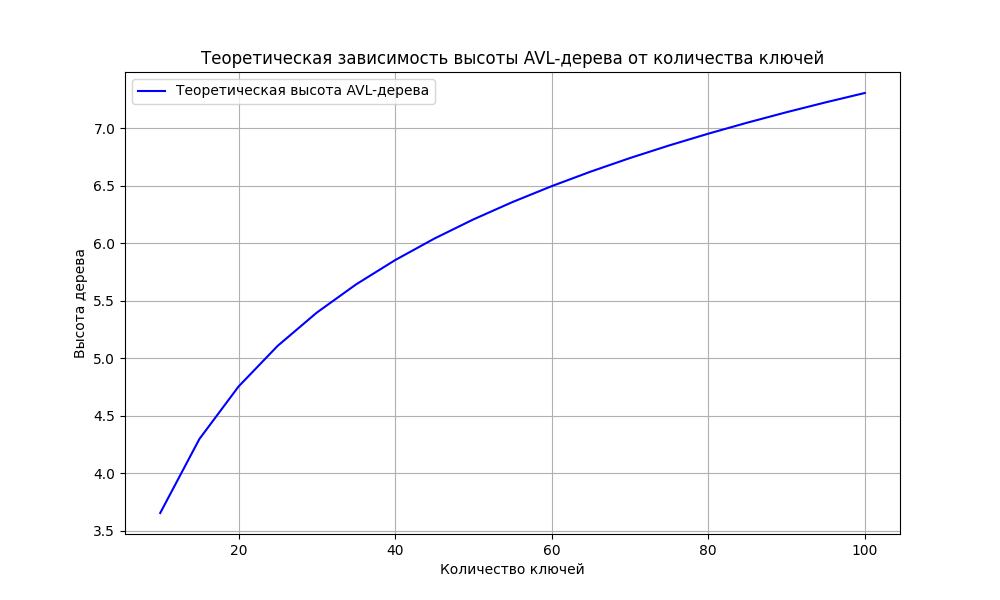


Лучший случай: ключи распределены равномерно

Средний случай: ключи распределены случайным образом

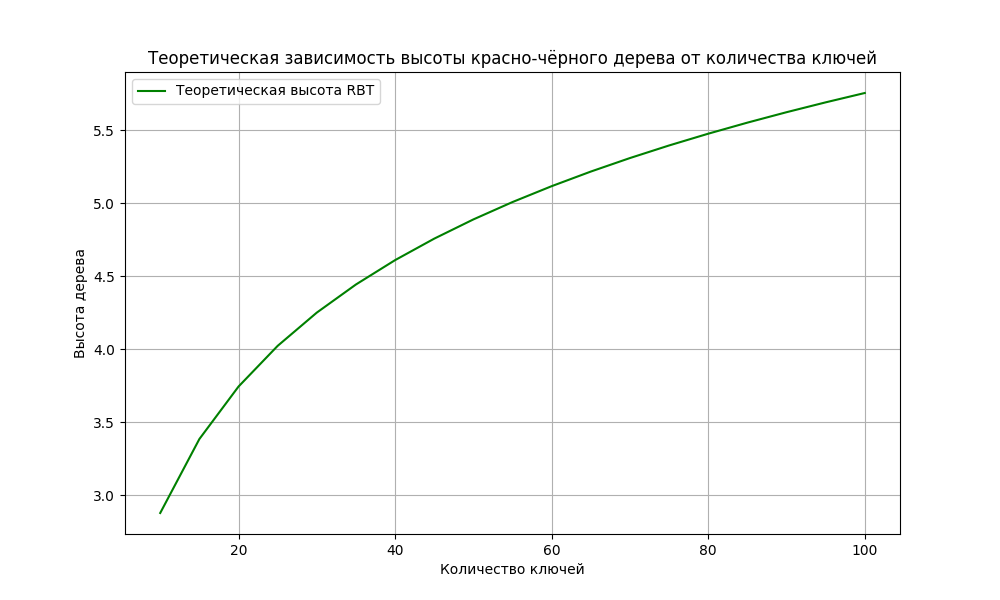
Худший случай: значения ключей монотонно возрастают

### **График асимптотической оценки 3 случаев AVL-дерева:**



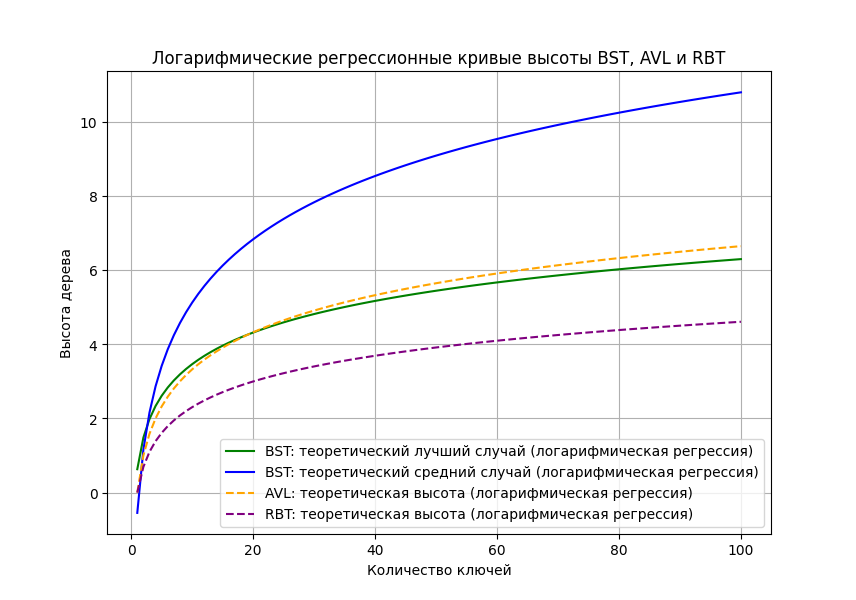
Лучший, средний и худший случай асимптотически совпадают.

### **График асимптотической оценки 3 случаев RB-дерева:**



Лучший, средний и худший случай асимптотически совпадают.

### **Общий график:**

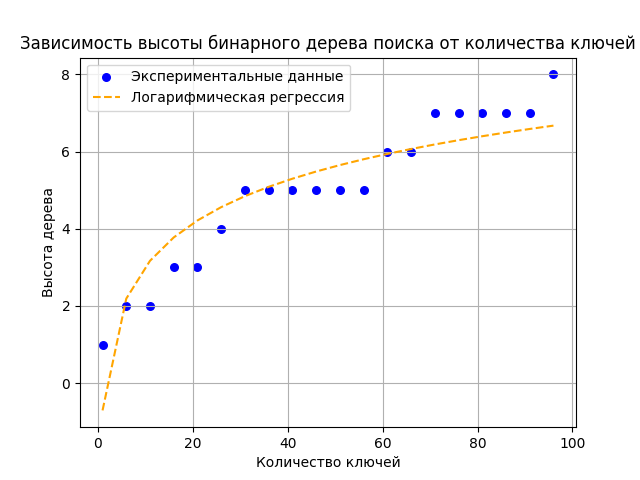


На общем графике регрессионные кривые всех деревьев хорошо просматриваются и асимптотически совпадают. Не включен только худший случай BST, так как он представляет собой линейную зависимость, быстро возрастает и не дает рассмотреть другие графики.

# **Практическая часть**

# **Зависимость высоты бинарного дерева от количества ключей.**

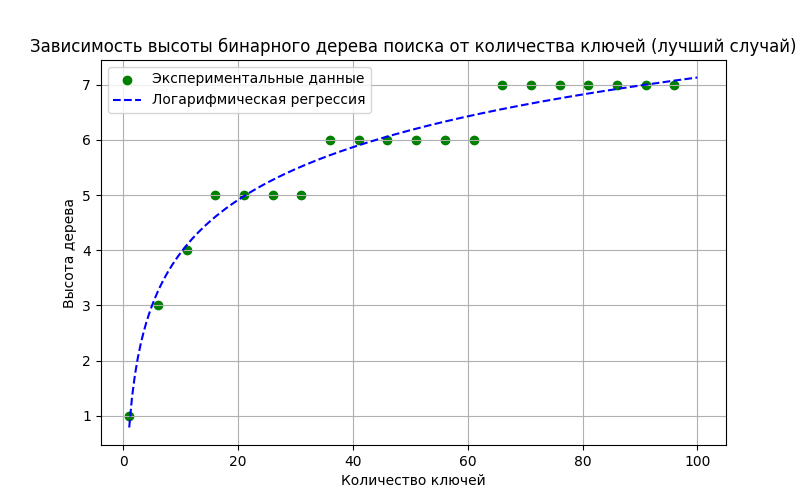
### **Значение ключа – случайная величина:**



H(n) = 1.616871 \* log(n) + -0.707531

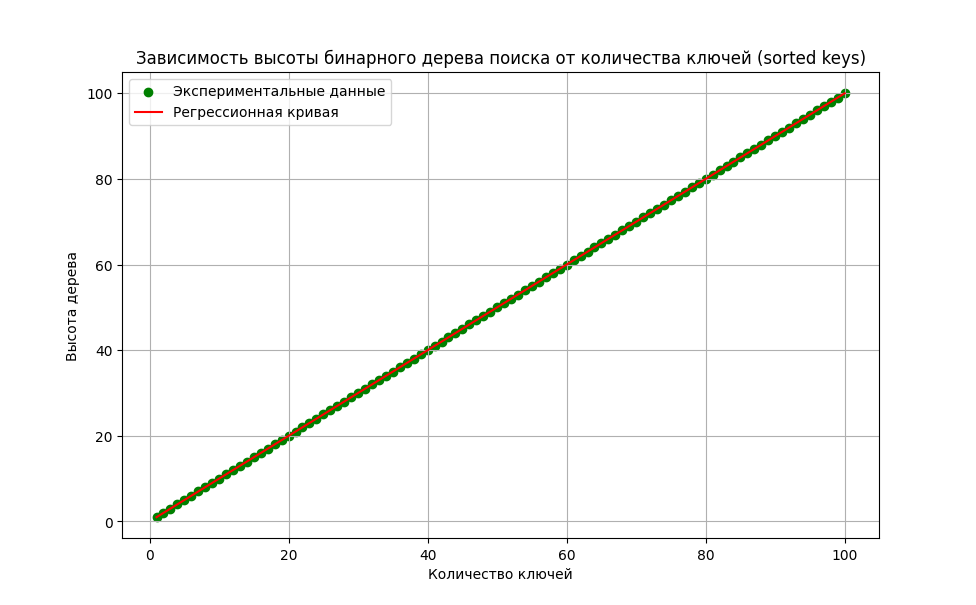
Когда значение ключа – случайная величина, заметно, что на каждом уровне располагается разное количество элементов. Это не противоречит логарифмической асимптотике среднего и лучшего случаев бинарного дерева. На общем графике более заметно какому из двух случаев наиболее подобна данная регрессионная кривая. Стоит отметить, что точки экспериментальных данных выводятся с шагом 5.

### **Значения ключей распределены равномерно:**



H(n) = 1.377034 \* log(n) + 0.789090

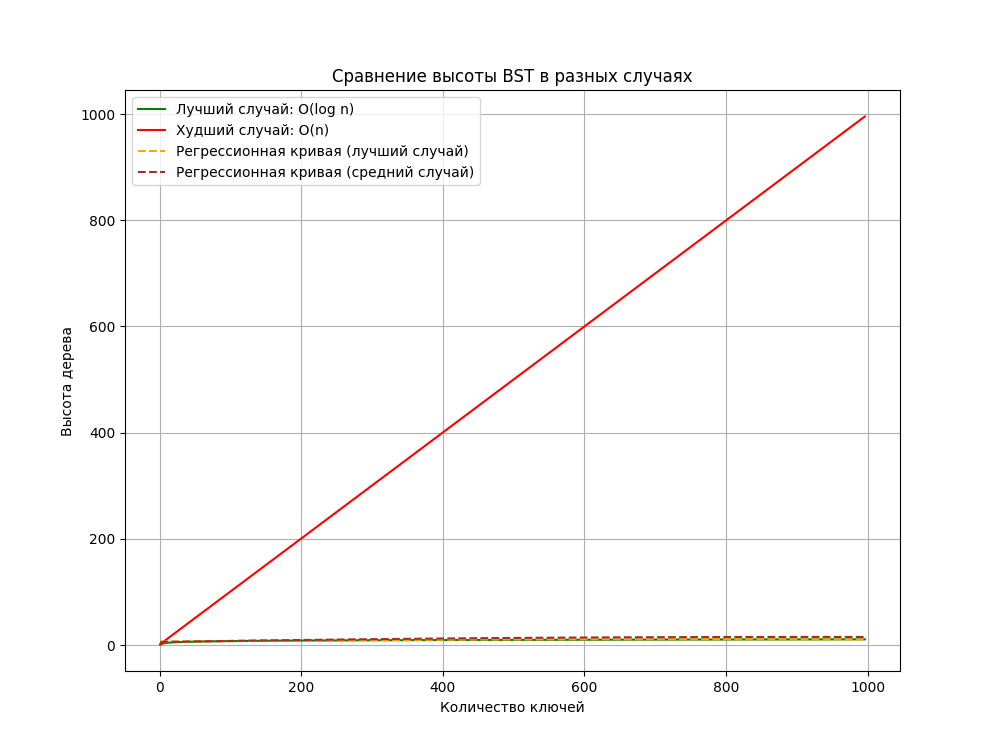
### **Значения ключей распределены в порядке возрастания:**



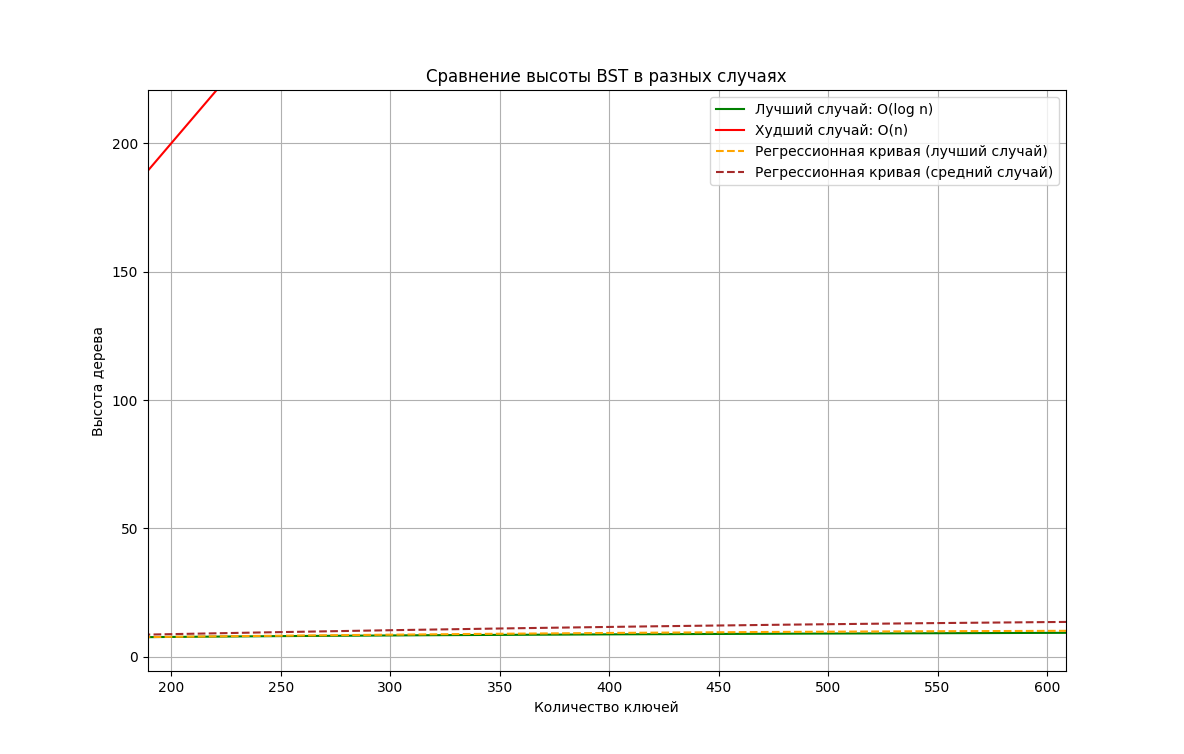
-0.000000x^2 + 1.000000x + -0.000000

Элементы выводятся с шагом 5. Линейная зависимость – худший случай для бинарного дерева, возникает она в том случае, когда элементы в последовательности идут по возрастанию или по убыванию.

### **Общий график:**



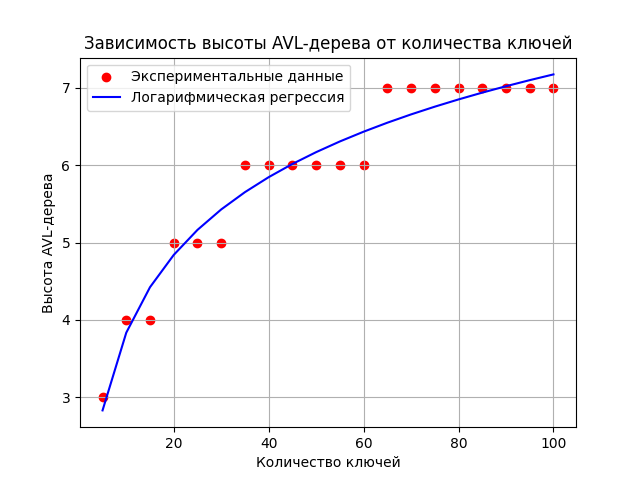
Для наглядности было необходимо увеличить количество ключей. На общем графике из-за скорости роста линейной зависимости плохо просматриваются логарифмические кривые, но заметно, что средний и лучший случай практически совпадают. Приблизим график:



Здесь мы видим, что регрессионная кривая лучшего случая практически совпадает с теоретической кривой, а регрессионная кривая среднего случая проходит чуть выше, начиная примерно с 200 элементов.

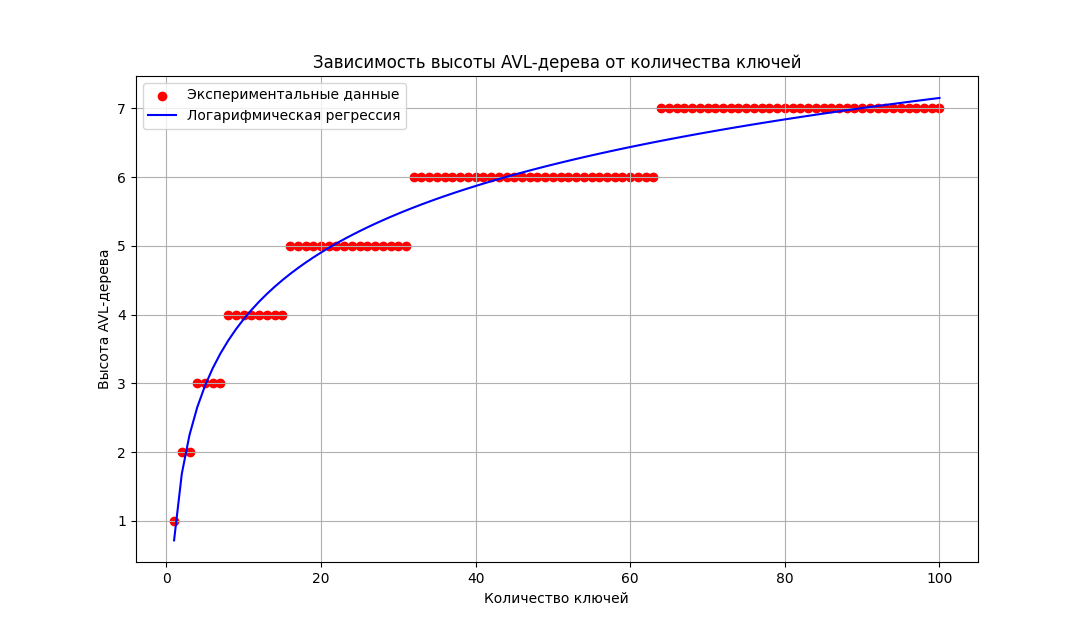
# **Зависимость высоты AVL-дерева от количества ключей.**

### **Значения ключей монотонно возрастают.**



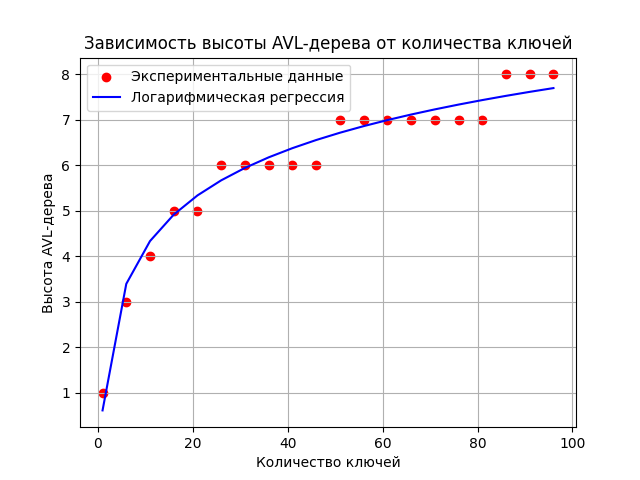
h(N) = 1.451125 \* log(N) + 0.492789

По графику, где точки выводятся с шагом 5 довольно сложно предположить, как элементы оказываются распределены. Добавим для наглядности тот же график, но где точки будут выводиться с шагом 1.



Заметим, что на каждом новом уровне количество ключей вдвое больше, чем на предыдущем. Это свидетельствует о геометрической прогрессии, происходит эффективная балансировка дерева.

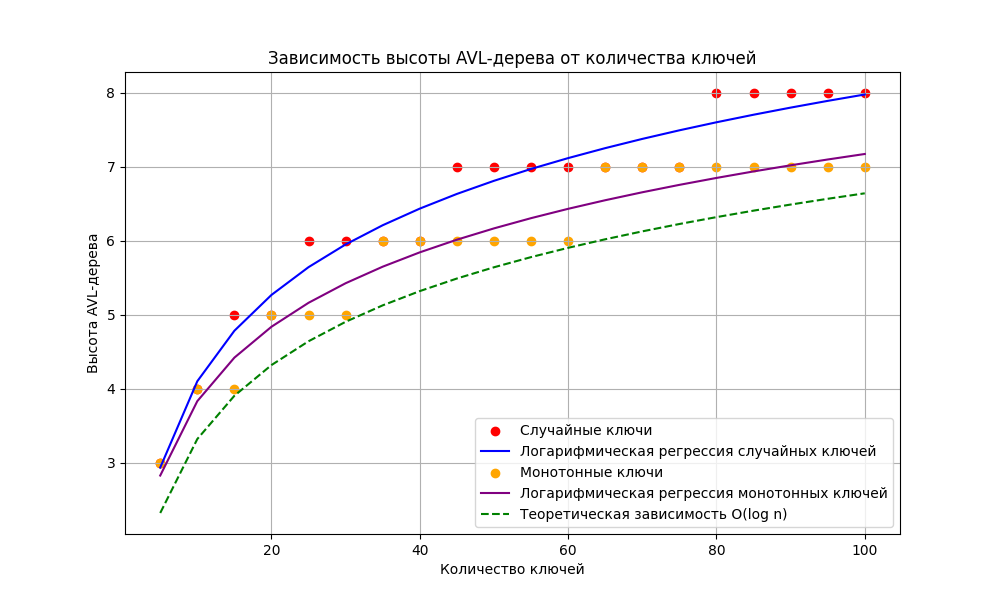
### **Значение ключа – случайная величина.**



h(N) = 1.552550 \* log(N) + 0.612765

Определенной закономерности здесь не наблюдается, но на данном графике на каждом новом уровне количество элементов больше, чем на предыдущем. Кроме того, если сравнить высоту этого дерева, с высотой дерева на предыдущем графике, можно заметить, что она выше. Нагляднее это просматривается на общем графике.

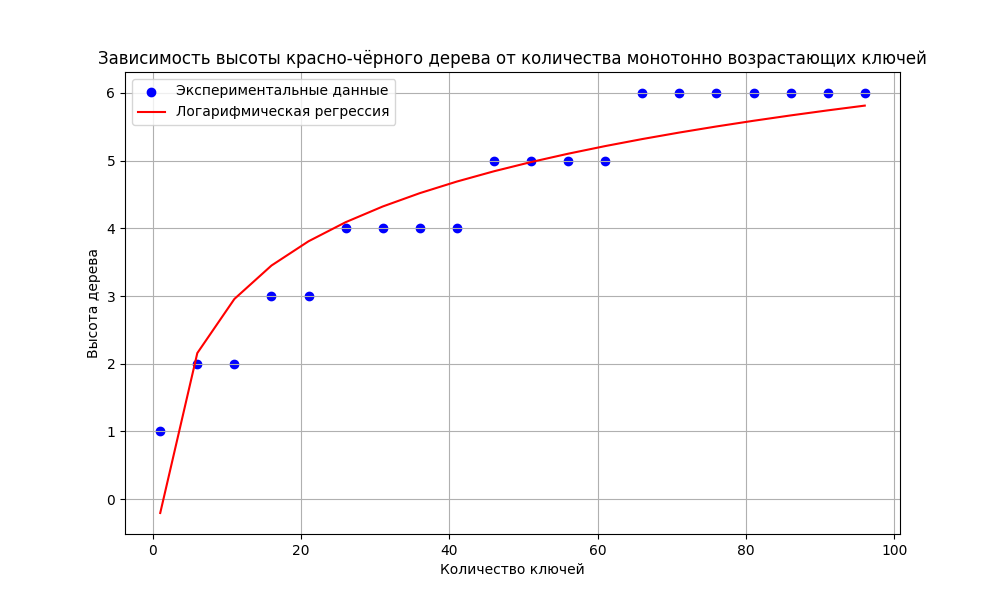
### **Общий график:**



На данном графике хорошо видно, что график регрессионной кривой, отображающей зависимость высоты AVL-дерева от количества случайных ключей возрастает быстрее, чем график регрессионной кривой, отображающей зависимость высоты AVL-дерева от количества монотонно возрастающих ключей. Это соответствует теории, так как для всех 3 случаев асимптотика AVL-дерева равна O(log n), так как AVL-дерево специально поддерживает балансировку.

# **Зависимость высоты красно-чёрного дерева от количества ключей.**

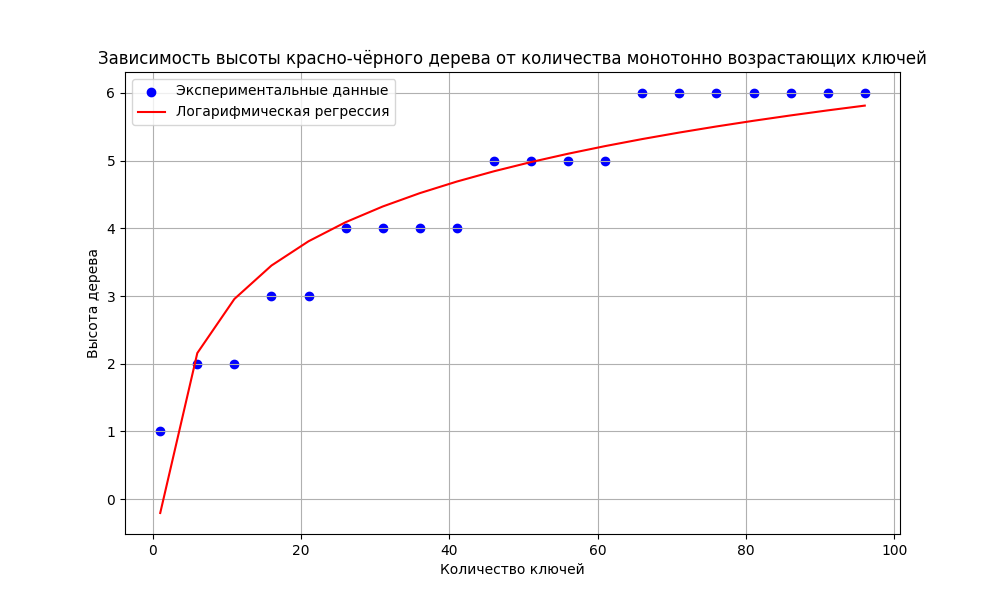
### **Значения ключей монотонно возрастают.**



h = 1.318475 \* log(N) + -0.204196

По графику, где точки выводятся с шагом 5 можно заметить, что элементы оказываются распределены таким образом, что на каждом нечетном уровне, кроме первого, их столько же, сколько на предыдущем. Кроме того, прослеживается геометрическая прогрессия: число точек каждый раз увеличивается в 2 раза.

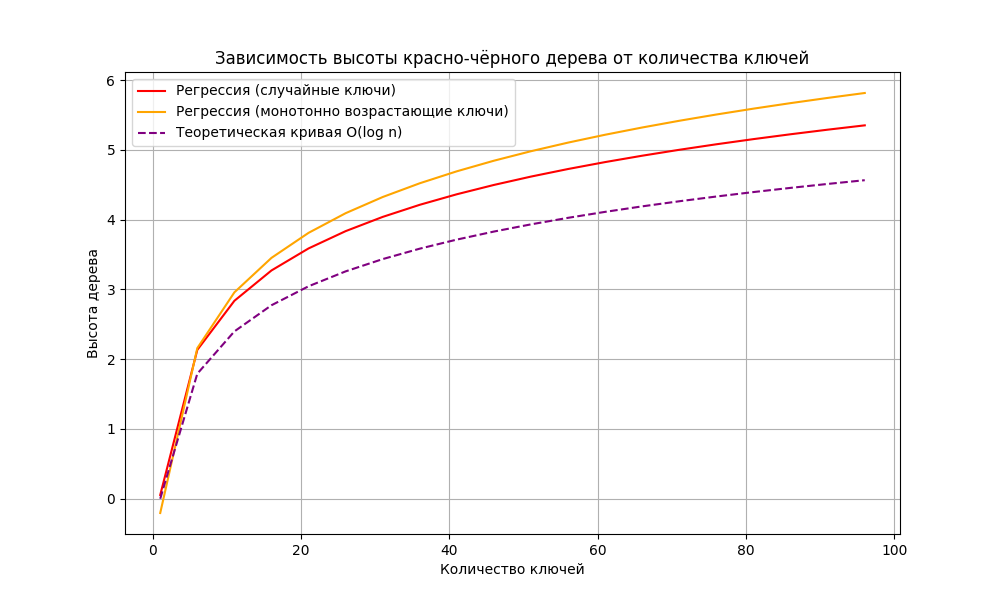
### **Значение ключа – случайная величина.**



h = 1.318475 \* log(N) + -0.204196

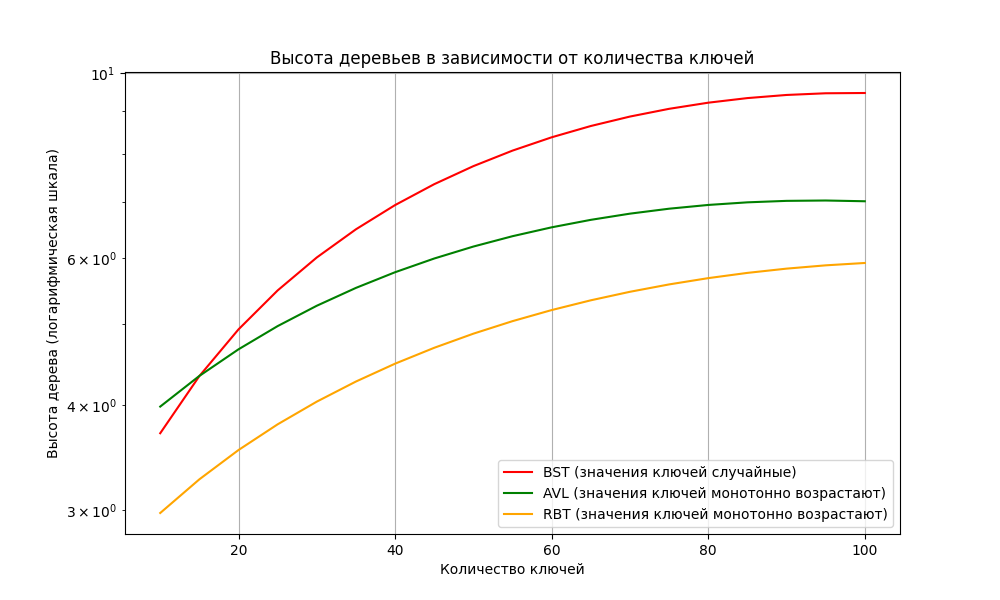
Аналогично, по графику со случайными значениями ключей, где точки выводятся с шагом 5 можно заметить, что элементы оказываются распределены таким образом, что на каждом нечетном уровне, кроме первого, их столько же, сколько на предыдущем. В отличие от графика, где отображены монотонно возрастающие значения, геометрическая прогрессия не прослеживается.

### **Общий график.**

 На общем графике, где точки также выводятся с шагом 5 можно заметить, что асимптотика обоих случаев совпадает с теоретической для всех трех случаев: O(log n), но график регрессии монотонно возрастающих ключей растет быстрее графика регрессии случайных ключей, так как балансировка выполняется чаще.

# **Объединенный практический график.**

### **Объединенный график.**

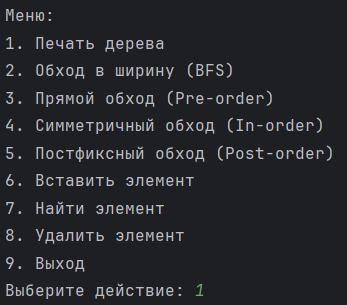


На общем графике хорошо просматривается, что асимптотика всех трех случаев соответствует O(log n).

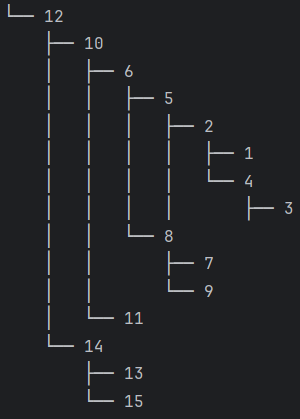
# **Реализация обходов деревьев.**

### **BST.**

При запуске программы пользователю предложены следующие действия:

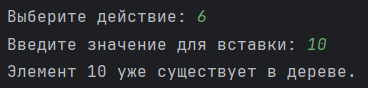


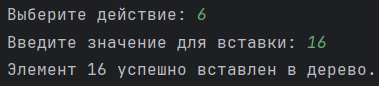
Выбрав 1, пользователь получит следующую картину, так как уже существует дерево, автоматически заполненное элементами от 1 до 15:

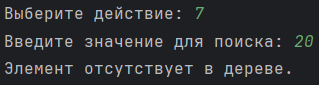


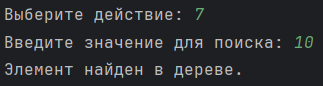
Примечание: Если один из последних вставленных в ветвь элементов меньше, чем тот, после которого он был вставлен (например, 2 < 5, 3 < 4), то символы разветвления могут выводиться несколько некорректно и отступ вниз учитывать не стоит.

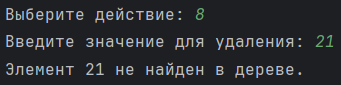
Пользователем были выбраны следующие действия:

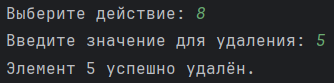




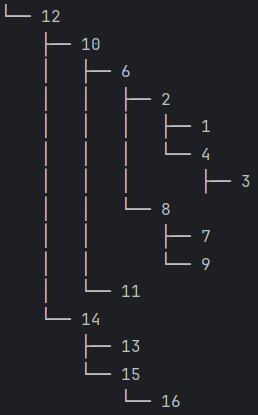




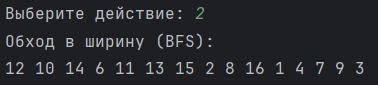
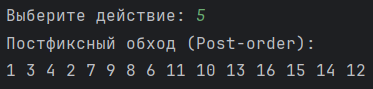
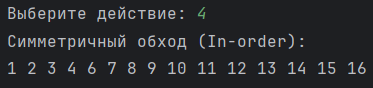
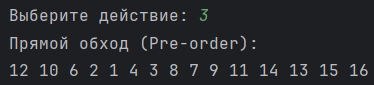




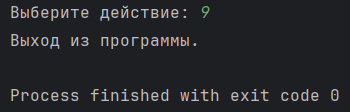
Результат получился следующий:



Обходы в свою очередь выглядят так:

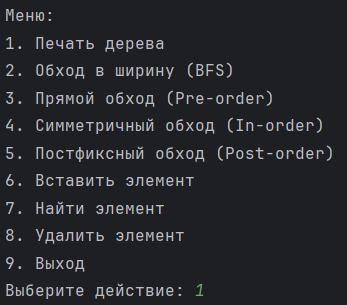
 

При нажатии клавиши 9 происходит выход из программы:

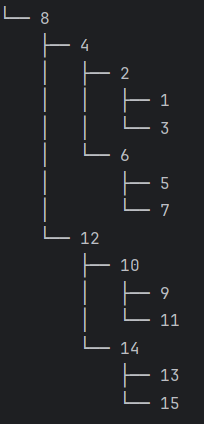


### **AVL.**

При запуске программы пользователю предложены следующие действия:

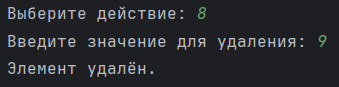
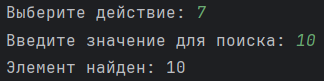
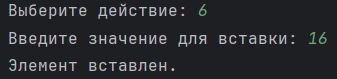


Выбрав 1, пользователь получит следующую картину, так как уже существует дерево, автоматически заполненное элементами от 1 до 15:

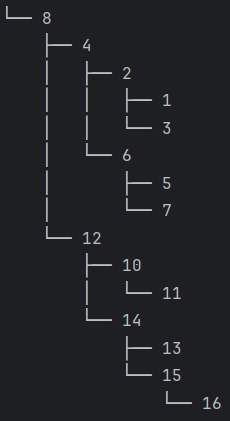


Это дерево соответствует необходимым условиям AVL-дерева.

Пользователем были выбраны следующие действия:

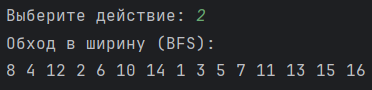


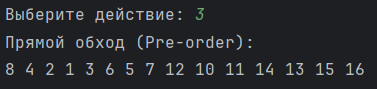
Результат получился следующий:

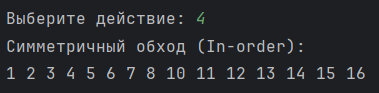


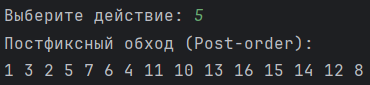
Операции были выполнены корректно, дерево соответствует необходимым условиям AVL-дерева.

Обходы в свою очередь выглядят так:

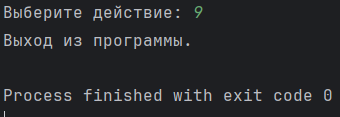






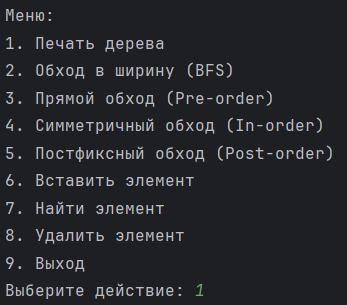


При нажатии клавиши 9 происходит выход из программы:

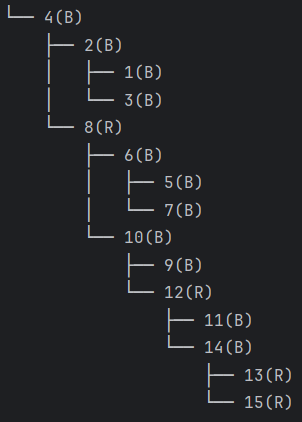


### **RB.**

При запуске программы пользователю предложены следующие действия:

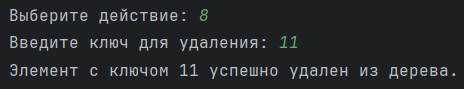
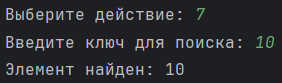
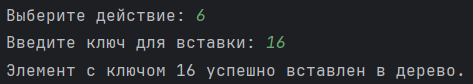


Выбрав 1, пользователь получит следующую картину, так как уже существует дерево, автоматически заполненное элементами от 1 до 15:

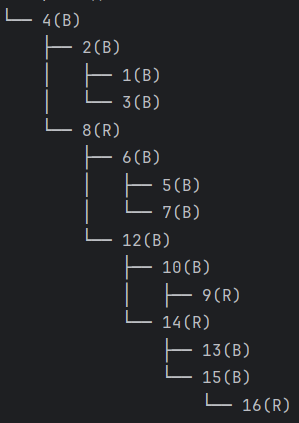


Это дерево соответствует необходимым условиям RB-дерева.

Пользователем были выбраны следующие действия:

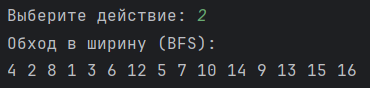
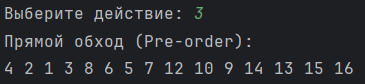
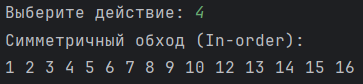
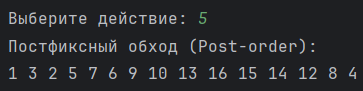


Результат получился следующий:

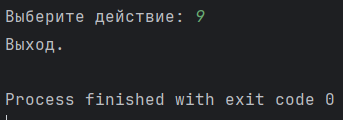


Операции были выполнены корректно, дерево соответствует необходимым условиям RB-дерева.

Обходы в свою очередь выглядят так:

При нажатии клавиши 9 происходит выход из программы:



# **Раздел со всем кодом.**

# **Код деревьев.**

### **BST.**

import java.util.Collections;  
import java.util.LinkedList;  
import java.util.Queue;  
import java.util.Scanner;  
import java.util.Random;  
import java.util.ArrayList;  
  
class Node {  
 int key;  
 Node leftChild, rightChild;  
  
 public Node(int key) {  
 this.key = key;  
 this.leftChild = this.rightChild = null;  
 }  
}  
  
public class BinaryTree {  
 Node root;  
  
 public BinaryTree() {  
 this.root = null;  
 }  
  
 // Метод для печати дерева  
 public void printTree(Node node, String indent, boolean last) {  
 if (node != null) {  
 System.*out*.println(indent + (last ? "└── " : "├── ") + node.key);  
 indent += last ? " " : "│ ";  
  
 printTree(node.leftChild, indent, false);  
 printTree(node.rightChild, indent, true);  
 }  
 }  
  
 // Вставка элемента  
 public void insert(int key) {  
 if (search(key)) {  
 System.*out*.println("Элемент " + key + " уже существует в дереве.");  
 } else {  
 root = insertRecursive(root, key);  
 System.*out*.println("Элемент " + key + " успешно вставлен в дерево.");  
 }  
 }  
  
 private Node insertRecursive(Node node, int key) {  
 if (node == null) {  
 return new Node(key);  
 }  
  
 if (key < node.key) {  
 node.leftChild = insertRecursive(node.leftChild, key);  
 } else if (key > node.key) {  
 node.rightChild = insertRecursive(node.rightChild, key);  
 }  
  
 return node;  
 }  
  
 // Поиск элемента  
 public boolean search(int key) {  
 return searchRecursive(root, key);  
 }  
  
 private boolean searchRecursive(Node node, int key) {  
 if (node == null) {  
 return false;  
 }  
  
 if (key == node.key) {  
 return true;  
 }  
  
 return key < node.key ? searchRecursive(node.leftChild, key) : searchRecursive(node.rightChild, key);  
 }  
  
 // Удаление элемента  
 public void delete(int key) {  
 if (!search(key)) {  
 System.*out*.println("Элемент " + key + " не найден в дереве.");  
 } else {  
 root = deleteRecursive(root, key);  
 System.*out*.println("Элемент " + key + " успешно удалён.");  
  
 // Проверка на пустоту дерева  
 if (root == null) {  
 System.*out*.println("Дерево пустое.");  
 }  
 }  
 }  
  
  
 private Node deleteRecursive(Node node, int key) {  
 if (node == null) {  
 return null;  
 }  
  
 if (key < node.key) {  
 node.leftChild = deleteRecursive(node.leftChild, key);  
 } else if (key > node.key) {  
 node.rightChild = deleteRecursive(node.rightChild, key);  
 } else {  
 // Узел с одним дочерним элементом или без детей  
 if (node.leftChild == null) {  
 return node.rightChild;  
 } else if (node.rightChild == null) {  
 return node.leftChild;  
 }  
  
 // Узел с двумя детьми: получаем наименьший элемент в правом поддереве  
 node.key = minValue(node.rightChild);  
  
 // Удаляем наименьший элемент из правого поддерева  
 node.rightChild = deleteRecursive(node.rightChild, node.key);  
 }  
  
 return node;  
 }  
  
 // Находим минимальное значение в дереве  
 private int minValue(Node node) {  
 int minValue = node.key;  
 while (node.leftChild != null) {  
 minValue = node.leftChild.key;  
 node = node.leftChild;  
 }  
 return minValue;  
 }  
  
 // Основной метод  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 Random random = new Random();  
 BinaryTree tree = new BinaryTree();  
  
 ArrayList<Integer> list = new ArrayList<>();  
 for (int i = 1; i <= 15; i++) {  
 list.add(i);  
 }  
 Collections.*shuffle*(list);  
  
 for (int value : list) {  
 tree.insert(value);  
 }  
  
 while (true) {  
 System.*out*.println("\nМеню:");  
 System.*out*.println("1. Печать дерева");  
 System.*out*.println("2. Обход в ширину (BFS)");  
 System.*out*.println("3. Прямой обход (Pre-order)");  
 System.*out*.println("4. Симметричный обход (In-order)");  
 System.*out*.println("5. Постфиксный обход (Post-order)");  
 System.*out*.println("6. Вставить элемент");  
 System.*out*.println("7. Найти элемент");  
 System.*out*.println("8. Удалить элемент");  
 System.*out*.println("9. Выход");  
 System.*out*.print("Выберите действие: ");  
  
 int choice = scanner.nextInt();  
 switch (choice) {  
 case 1:  
 tree.printTree(tree.root, "", true);  
 break;  
 case 2:  
 System.*out*.println("Обход в ширину (BFS):");  
 tree.breadthFirstTraversal();  
 break;  
 case 3:  
 System.*out*.println("Прямой обход (Pre-order):");  
 tree.depthFirstPreOrder(tree.root);  
 System.*out*.println();  
 break;  
 case 4:  
 System.*out*.println("Симметричный обход (In-order):");  
 tree.depthFirstInOrder(tree.root);  
 System.*out*.println();  
 break;  
 case 5:  
 System.*out*.println("Постфиксный обход (Post-order):");  
 tree.depthFirstPostOrder(tree.root);  
 System.*out*.println();  
 break;  
 case 6:  
 System.*out*.print("Введите значение для вставки: ");  
 int insertValue = scanner.nextInt();  
 tree.insert(insertValue);  
 break;  
 case 7:  
 System.*out*.print("Введите значение для поиска: ");  
 int searchValue = scanner.nextInt();  
 if (tree.search(searchValue)) {  
 System.*out*.println("Элемент найден в дереве.");  
 } else {  
 System.*out*.println("Элемент отсутствует в дереве.");  
 }  
 break;  
 case 8:  
 System.*out*.print("Введите значение для удаления: ");  
 int deleteValue = scanner.nextInt();  
 tree.delete(deleteValue);  
 System.*out*.println("Элемент удалён.");  
 break;  
 case 9:  
 System.*out*.println("Выход из программы.");  
 scanner.close();  
 return;  
 default:  
 System.*out*.println("Неверный выбор. Попробуйте снова.");  
 }  
 }  
 }  
  
 // BFS обход  
 public void breadthFirstTraversal() {  
 if (root == null) return;  
 Queue<Node> queue = new LinkedList<>();  
 queue.add(root);  
  
 while (!queue.isEmpty()) {  
 Node currentNode = queue.poll();  
 System.*out*.print(currentNode.key + " ");  
  
 if (currentNode.leftChild != null) queue.add(currentNode.leftChild);  
 if (currentNode.rightChild != null) queue.add(currentNode.rightChild);  
 }  
 System.*out*.println();  
 }  
  
 // Pre-order обход  
 public void depthFirstPreOrder(Node node) {  
 if (node == null) return;  
 System.*out*.print(node.key + " ");  
 depthFirstPreOrder(node.leftChild);  
 depthFirstPreOrder(node.rightChild);  
 }  
  
 // In-order обход  
 public void depthFirstInOrder(Node node) {  
 if (node == null) return;  
 depthFirstInOrder(node.leftChild);  
 System.*out*.print(node.key + " ");  
 depthFirstInOrder(node.rightChild);  
 }  
  
 // Post-order обход  
 public void depthFirstPostOrder(Node node) {  
 if (node == null) return;  
 depthFirstPostOrder(node.leftChild);  
 depthFirstPostOrder(node.rightChild);  
 System.*out*.print(node.key + " ");  
 }  
}

### **AVL.**

* import java.util.LinkedList;  
  import java.util.Queue;  
  import java.util.Scanner;  
    
  class AVLNode {  
   int key, height;  
   AVLNode leftChild, rightChild;  
    
   public AVLNode(int key) {  
   this.key = key;  
   this.height = 1;  
   }  
  }  
    
  public class AVLTree {  
   AVLNode root;  
    
   // Вставка узла  
   public AVLNode insert(AVLNode node, int key) {  
   if (node == null) {  
   return new AVLNode(key);  
   }  
    
   if (key < node.key) {  
   node.leftChild = insert(node.leftChild, key);  
   } else if (key > node.key) {  
   node.rightChild = insert(node.rightChild, key);  
   } else {  
   System.*out*.println("Элемент " + key + " уже существует в дереве."); // Если элемент уже есть  
   return node; // Не вставляем дубликат  
   }  
    
   node.height = 1 + Math.*max*(height(node.leftChild), height(node.rightChild));  
   int balance = getBalance(node);  
    
   // Балансировка  
   if (balance > 1 && key < node.leftChild.key) {  
   return rightRotate(node);  
   }  
   if (balance < -1 && key > node.rightChild.key) {  
   return leftRotate(node);  
   }  
   if (balance > 1 && key > node.leftChild.key) {  
   node.leftChild = leftRotate(node.leftChild);  
   return rightRotate(node);  
   }  
   if (balance < -1 && key < node.rightChild.key) {  
   node.rightChild = rightRotate(node.rightChild);  
   return leftRotate(node);  
   }  
    
   return node;  
   }  
    
   // Поиск узла  
   public AVLNode search(AVLNode node, int key) {  
   if (node == null || node.key == key) {  
   return node;  
   }  
   if (key < node.key) {  
   return search(node.leftChild, key);  
   } else {  
   return search(node.rightChild, key);  
   }  
   }  
    
   // Удаление узла  
   public AVLNode delete(AVLNode root, int key) {  
   if (root == null) {  
   System.*out*.println("Элемент " + key + " не найден в дереве.");  
   return root;  
   }  
    
   if (key < root.key) {  
   root.leftChild = delete(root.leftChild, key);  
   } else if (key > root.key) {  
   root.rightChild = delete(root.rightChild, key);  
   } else {  
   if ((root.leftChild == null) || (root.rightChild == null)) {  
   AVLNode temp = (root.leftChild != null) ? root.leftChild : root.rightChild;  
   if (temp == null) {  
   temp = root;  
   root = null;  
   } else {  
   root = temp;  
   }  
   } else {  
   AVLNode temp = minValueNode(root.rightChild);  
   root.key = temp.key;  
   root.rightChild = delete(root.rightChild, temp.key);  
   }  
   }  
    
   if (root == null) {  
   return root;  
   }  
    
   root.height = Math.*max*(height(root.leftChild), height(root.rightChild)) + 1;  
   int balance = getBalance(root);  
    
   if (balance > 1 && getBalance(root.leftChild) >= 0) {  
   return rightRotate(root);  
   }  
   if (balance > 1 && getBalance(root.leftChild) < 0) {  
   root.leftChild = leftRotate(root.leftChild);  
   return rightRotate(root);  
   }  
   if (balance < -1 && getBalance(root.rightChild) <= 0) {  
   return leftRotate(root);  
   }  
   if (balance < -1 && getBalance(root.rightChild) > 0) {  
   root.rightChild = rightRotate(root.rightChild);  
   return leftRotate(root);  
   }  
    
   return root;  
   }  
    
   // Вспомогательные методы для балансировки  
   private int height(AVLNode node) {  
   return (node == null) ? 0 : node.height;  
   }  
    
   private int getBalance(AVLNode node) {  
   return (node == null) ? 0 : height(node.leftChild) - height(node.rightChild);  
   }  
    
   private AVLNode rightRotate(AVLNode y) {  
   AVLNode x = y.leftChild;  
   AVLNode T2 = x.rightChild;  
   x.rightChild = y;  
   y.leftChild = T2;  
   y.height = Math.*max*(height(y.leftChild), height(y.rightChild)) + 1;  
   x.height = Math.*max*(height(x.leftChild), height(x.rightChild)) + 1;  
   return x;  
   }  
    
   private AVLNode leftRotate(AVLNode x) {  
   AVLNode y = x.rightChild;  
   AVLNode T2 = y.leftChild;  
   y.leftChild = x;  
   x.rightChild = T2;  
   x.height = Math.*max*(height(x.leftChild), height(x.rightChild)) + 1;  
   y.height = Math.*max*(height(y.leftChild), height(y.rightChild)) + 1;  
   return y;  
   }  
    
   private AVLNode minValueNode(AVLNode node) {  
   AVLNode current = node;  
   while (current.leftChild != null) {  
   current = current.leftChild;  
   }  
   return current;  
   }  
    
   // Обходы дерева  
   public void preOrderTraversal(AVLNode node) {  
   if (node != null) {  
   System.*out*.print(node.key + " ");  
   preOrderTraversal(node.leftChild);  
   preOrderTraversal(node.rightChild);  
   }  
   }  
    
   public void inOrderTraversal(AVLNode node) {  
   if (node != null) {  
   inOrderTraversal(node.leftChild);  
   System.*out*.print(node.key + " ");  
   inOrderTraversal(node.rightChild);  
   }  
   }  
    
   public void postOrderTraversal(AVLNode node) {  
   if (node != null) {  
   postOrderTraversal(node.leftChild);  
   postOrderTraversal(node.rightChild);  
   System.*out*.print(node.key + " ");  
   }  
   }  
    
   // Метод для отображения дерева  
   public void printTree(AVLNode AVLNode, String indent, boolean last) {  
   if (AVLNode != null) {  
   System.*out*.println(indent + (last ? "└── " : "├── ") + AVLNode.key);  
   indent += last ? " " : "│ ";  
    
   printTree(AVLNode.leftChild, indent, false);  
   printTree(AVLNode.rightChild, indent, true);  
   }  
   }  
    
    
    
   // Метод обхода в ширину (BFS)  
   public void breadthFirstTraversal(AVLNode root) {  
   if (root == null) return;  
   Queue<AVLNode> queue = new LinkedList<>();  
   queue.add(root);  
   while (!queue.isEmpty()) {  
   AVLNode tempNode = queue.poll();  
   System.*out*.print(tempNode.key + " ");  
   if (tempNode.leftChild != null) queue.add(tempNode.leftChild);  
   if (tempNode.rightChild != null) queue.add(tempNode.rightChild);  
   }  
   }  
   public boolean isTreeEmpty() {  
   return root == null;  
   }  
    
   public static void main(String[] args) {  
   AVLTree tree = new AVLTree();  
   Scanner scanner = new Scanner(System.*in*);  
    
   int[] array = new int[15];  
   for (int i = 0; i < 15; i++) {  
   array[i] = i + 1;  
   }  
   for (int value : array) {  
   tree.root = tree.insert(tree.root, value);  
   }  
    
   boolean exit = false;  
   while (!exit) {  
   System.*out*.println("\nМеню:");  
   System.*out*.println("1. Печать дерева");  
   System.*out*.println("2. Обход в ширину (BFS)");  
   System.*out*.println("3. Прямой обход (Pre-order)");  
   System.*out*.println("4. Симметричный обход (In-order)");  
   System.*out*.println("5. Постфиксный обход (Post-order)");  
   System.*out*.println("6. Вставить элемент");  
   System.*out*.println("7. Найти элемент");  
   System.*out*.println("8. Удалить элемент");  
   System.*out*.println("9. Выход");  
   System.*out*.print("Выберите действие: ");  
    
   int choice = scanner.nextInt();  
   switch (choice) {  
   case 1:  
   if (tree.isTreeEmpty()) {  
   System.*out*.println("Дерево пустое.");  
   } else {  
   tree.printTree(tree.root, "", true);  
   }  
   break;  
   case 2:  
   System.*out*.println("Обход в ширину (BFS):");  
   tree.breadthFirstTraversal(tree.root);  
   System.*out*.println();  
   break;  
   case 3:  
   System.*out*.println("Прямой обход (Pre-order):");  
   tree.preOrderTraversal(tree.root);  
   System.*out*.println();  
   break;  
   case 4:  
   System.*out*.println("Симметричный обход (In-order):");  
   tree.inOrderTraversal(tree.root);  
   System.*out*.println();  
   break;  
   case 5:  
   System.*out*.println("Постфиксный обход (Post-order):");  
   tree.postOrderTraversal(tree.root);  
   System.*out*.println();  
   break;  
   case 6:  
   System.*out*.print("Введите значение для вставки: ");  
   int valueToInsert = scanner.nextInt();  
   tree.root = tree.insert(tree.root, valueToInsert);  
   System.*out*.println("Элемент вставлен.");  
   break;  
   case 7:  
   System.*out*.print("Введите значение для поиска: ");  
   int valueToFind = scanner.nextInt();  
   AVLNode result = tree.search(tree.root, valueToFind);  
   if (result != null) {  
   System.*out*.println("Элемент найден: " + result.key);  
   } else {  
   System.*out*.println("Элемент не найден.");  
   }  
   break;  
   case 8:  
   System.*out*.print("Введите значение для удаления: ");  
   int valueToDelete = scanner.nextInt();  
   tree.root = tree.delete(tree.root, valueToDelete);  
   System.*out*.println("Элемент удалён.");  
   break;  
   case 9:  
   System.*out*.println("Выход из программы.");  
   exit = true;  
   break;  
   default:  
   System.*out*.println("Некорректный ввод. Попробуйте снова.");  
   }  
   }  
   scanner.close();  
   }  
  }

### **RB.**

* import java.util.LinkedList;  
  import java.util.Queue;  
  import java.util.Scanner;  
    
  class RBNode {  
   int key;  
   RBNode leftChild, rightChild, parent;  
   boolean isRed; // true - красный, false - черный  
    
   public RBNode(int key) {  
   this.key = key;  
   this.leftChild = this.rightChild = this.parent = null;  
   this.isRed = true; // Новый узел всегда красный  
   }  
  }  
    
  class RedBlackTree {  
   private RBNode root;  
   private RBNode TNULL; // NIL узел (черный лист)  
    
   public RedBlackTree() {  
   TNULL = new RBNode(0);  
   TNULL.isRed = false; // Листья всегда черные  
   root = TNULL;  
   }  
    
   // Вставка узла  
   // Вставка узла с проверкой на дубликаты  
   public void insert(int key) {  
   // Проверяем, существует ли уже узел с таким ключом  
   if (search(key) != TNULL) {  
   System.*out*.println("Элемент с ключом " + key + " уже существует в дереве.");  
   return;  
   }  
    
   RBNode newNode = new RBNode(key);  
   newNode.leftChild = TNULL;  
   newNode.rightChild = TNULL;  
    
   RBNode y = null;  
   RBNode x = root;  
    
   while (x != TNULL) { // Поиск места для вставки  
   y = x;  
   if (newNode.key < x.key) {  
   x = x.leftChild;  
   } else {  
   x = x.rightChild;  
   }  
   }  
    
   newNode.parent = y;  
   if (y == null) {  
   root = newNode; // Новый узел становится корнем  
   } else if (newNode.key < y.key) {  
   y.leftChild = newNode;  
   } else {  
   y.rightChild = newNode;  
   }  
    
   if (newNode.parent == null) {  
   newNode.isRed = false; // Корень всегда черный  
   return;  
   }  
   fixInsert(newNode); // Балансировка после вставки  
   System.*out*.println("Элемент с ключом " + key + " успешно вставлен в дерево.");  
    
   fixInsert(newNode); // Балансировка после вставки  
   }  
    
    
   // Балансировка после вставки  
   private void fixInsert(RBNode node) {  
   while (node.parent != null && node.parent.isRed) {  
   if (node.parent == node.parent.parent.leftChild) {  
   RBNode uncle = node.parent.parent.rightChild;  
   if (uncle.isRed) {  
   node.parent.isRed = false;  
   uncle.isRed = false;  
   node.parent.parent.isRed = true;  
   node = node.parent.parent;  
   } else {  
   if (node == node.parent.rightChild) {  
   node = node.parent;  
   rotateLeft(node);  
   }  
   node.parent.isRed = false;  
   node.parent.parent.isRed = true;  
   rotateRight(node.parent.parent);  
   }  
   } else {  
   RBNode uncle = node.parent.parent.leftChild;  
   if (uncle.isRed) {  
   node.parent.isRed = false;  
   uncle.isRed = false;  
   node.parent.parent.isRed = true;  
   node = node.parent.parent;  
   } else {  
   if (node == node.parent.leftChild) {  
   node = node.parent;  
   rotateRight(node);  
   }  
   node.parent.isRed = false;  
   node.parent.parent.isRed = true;  
   rotateLeft(node.parent.parent);  
   }  
   }  
   }  
   root.isRed = false;  
   }  
    
   // Левый поворот  
   private void rotateLeft(RBNode node) {  
   RBNode rightNode = node.rightChild;  
   node.rightChild = rightNode.leftChild;  
   if (rightNode.leftChild != TNULL) {  
   rightNode.leftChild.parent = node;  
   }  
   rightNode.parent = node.parent;  
   if (node.parent == null) {  
   root = rightNode;  
   } else if (node == node.parent.leftChild) {  
   node.parent.leftChild = rightNode;  
   } else {  
   node.parent.rightChild = rightNode;  
   }  
   rightNode.leftChild = node;  
   node.parent = rightNode;  
   }  
    
   // Правый поворот  
   private void rotateRight(RBNode node) {  
   RBNode leftNode = node.leftChild;  
   node.leftChild = leftNode.rightChild;  
   if (leftNode.rightChild != TNULL) {  
   leftNode.rightChild.parent = node;  
   }  
   leftNode.parent = node.parent;  
   if (node.parent == null) {  
   root = leftNode;  
   } else if (node == node.parent.rightChild) {  
   node.parent.rightChild = leftNode;  
   } else {  
   node.parent.leftChild = leftNode;  
   }  
   leftNode.rightChild = node;  
   node.parent = leftNode;  
   }  
    
   // Поиск элемента  
   public RBNode search(int key) {  
   return searchHelper(root, key);  
   }  
    
   private RBNode searchHelper(RBNode node, int key) {  
   if (node == TNULL || key == node.key) {  
   return node;  
   }  
   if (key < node.key) {  
   return searchHelper(node.leftChild, key);  
   }  
   return searchHelper(node.rightChild, key);  
   }  
    
   // Удаление узла из дерева  
   public void delete(int key) {  
   RBNode nodeToDelete = search(key); // Найти узел для удаления  
   if (nodeToDelete == TNULL) {  
   System.*out*.println("Элемент не найден в дереве.");  
   return;  
   }  
   deleteNode(nodeToDelete);  
   System.*out*.println("Элемент с ключом " + key + " успешно удален из дерева.");  
   }  
    
   // Основная логика удаления узла  
   private void deleteNode(RBNode nodeToDelete) {  
   RBNode y = nodeToDelete;  
   RBNode x;  
   boolean yOriginalColor = y.isRed;  
    
   if (nodeToDelete.leftChild == TNULL) {  
   x = nodeToDelete.rightChild;  
   transplant(nodeToDelete, nodeToDelete.rightChild);  
   } else if (nodeToDelete.rightChild == TNULL) {  
   x = nodeToDelete.leftChild;  
   transplant(nodeToDelete, nodeToDelete.leftChild);  
   } else {  
   y = minimum(nodeToDelete.rightChild);  
   yOriginalColor = y.isRed;  
   x = y.rightChild;  
    
   if (y.parent == nodeToDelete) {  
   x.parent = y;  
   } else {  
   transplant(y, y.rightChild);  
   y.rightChild = nodeToDelete.rightChild;  
   y.rightChild.parent = y;  
   }  
    
   transplant(nodeToDelete, y);  
   y.leftChild = nodeToDelete.leftChild;  
   y.leftChild.parent = y;  
   y.isRed = nodeToDelete.isRed;  
   }  
    
   if (!yOriginalColor) {  
   fixDelete(x); // Балансировка после удаления  
   }  
   }  
    
   // Замена одного поддерева другим  
   private void transplant(RBNode target, RBNode with) {  
   if (target.parent == null) {  
   root = with;  
   } else if (target == target.parent.leftChild) {  
   target.parent.leftChild = with;  
   } else {  
   target.parent.rightChild = with;  
   }  
   with.parent = target.parent;  
   }  
    
   // Поиск минимального узла в поддереве  
   private RBNode minimum(RBNode node) {  
   while (node.leftChild != TNULL) {  
   node = node.leftChild;  
   }  
   return node;  
   }  
    
   // Балансировка после удаления  
   private void fixDelete(RBNode node) {  
   while (node != root && !node.isRed) {  
   if (node == node.parent.leftChild) {  
   RBNode sibling = node.parent.rightChild;  
   if (sibling.isRed) {  
   sibling.isRed = false;  
   node.parent.isRed = true;  
   rotateLeft(node.parent);  
   sibling = node.parent.rightChild;  
   }  
   if (!sibling.leftChild.isRed && !sibling.rightChild.isRed) {  
   sibling.isRed = true;  
   node = node.parent;  
   } else {  
   if (!sibling.rightChild.isRed) {  
   sibling.leftChild.isRed = false;  
   sibling.isRed = true;  
   rotateRight(sibling);  
   sibling = node.parent.rightChild;  
   }  
   sibling.isRed = node.parent.isRed;  
   node.parent.isRed = false;  
   sibling.rightChild.isRed = false;  
   rotateLeft(node.parent);  
   node = root;  
   }  
   } else {  
   RBNode sibling = node.parent.leftChild;  
   if (sibling.isRed) {  
   sibling.isRed = false;  
   node.parent.isRed = true;  
   rotateRight(node.parent);  
   sibling = node.parent.leftChild;  
   }  
   if (!sibling.rightChild.isRed && !sibling.leftChild.isRed) {  
   sibling.isRed = true;  
   node = node.parent;  
   } else {  
   if (!sibling.leftChild.isRed) {  
   sibling.rightChild.isRed = false;  
   sibling.isRed = true;  
   rotateLeft(sibling);  
   sibling = node.parent.leftChild;  
   }  
   sibling.isRed = node.parent.isRed;  
   node.parent.isRed = false;  
   sibling.leftChild.isRed = false;  
   rotateRight(node.parent);  
   node = root;  
   }  
   }  
   }  
   node.isRed = false;  
   }  
   public boolean isTreeEmpty() {  
   return root == TNULL;  
   }  
    
   // Печать структуры дерева  
   public void printTree() {  
   printTreeHelper(root, "", true);  
   }  
    
   public void printTreeHelper(RBNode node, String indent, boolean last) {  
   if (node != TNULL) {  
   System.*out*.println(indent + (last ? "└── " : "├── ") + node.key + (node.isRed ? "(R)" : "(B)"));  
   indent += last ? " " : "│ ";  
   printTreeHelper(node.leftChild, indent, false);  
   printTreeHelper(node.rightChild, indent, true);  
   }  
   }  
    
   // Обходы дерева  
   public void inorderTraversal(RBNode node) {  
   if (node != TNULL) {  
   inorderTraversal(node.leftChild);  
   System.*out*.print(node.key + " ");  
   inorderTraversal(node.rightChild);  
   }  
   }  
    
   public void preorderTraversal(RBNode node) {  
   if (node != TNULL) {  
   System.*out*.print(node.key + " ");  
   preorderTraversal(node.leftChild);  
   preorderTraversal(node.rightChild);  
   }  
   }  
    
   public void postorderTraversal(RBNode node) {  
   if (node != TNULL) {  
   postorderTraversal(node.leftChild);  
   postorderTraversal(node.rightChild);  
   System.*out*.print(node.key + " ");  
   }  
   }  
    
   public void levelOrderTraversal() {  
   if (root == TNULL) return;  
   Queue<RBNode> queue = new LinkedList<>();  
   queue.add(root);  
   while (!queue.isEmpty()) {  
   RBNode current = queue.poll();  
   System.*out*.print(current.key + " ");  
   if (current.leftChild != TNULL) queue.add(current.leftChild);  
   if (current.rightChild != TNULL) queue.add(current.rightChild);  
   }  
   }  
    
   // Основной метод (меню взаимодействия)  
   public static void main(String[] args) {  
   RedBlackTree tree = new RedBlackTree();  
   Scanner scanner = new Scanner(System.*in*);  
   int choice, key;  
    
   // Заполнение дерева значениями от 1 до 15  
   int[] array = new int[15];  
   for (int i = 0; i < 15; i++) {  
   array[i] = i + 1;  
   tree.insert(array[i]); // Вставка элементов в дерево  
   }  
    
   while (true) {  
   System.*out*.println("\nМеню:");  
   System.*out*.println("1. Печать дерева");  
   System.*out*.println("2. Обход в ширину (BFS)");  
   System.*out*.println("3. Прямой обход (Pre-order)");  
   System.*out*.println("4. Симметричный обход (In-order)");  
   System.*out*.println("5. Постфиксный обход (Post-order)");  
   System.*out*.println("6. Вставить элемент");  
   System.*out*.println("7. Найти элемент");  
   System.*out*.println("8. Удалить элемент");  
   System.*out*.println("9. Выход");  
   System.*out*.print("Выберите действие: ");  
   choice = scanner.nextInt();  
    
   switch (choice) {  
   case 1:  
   if (tree.isTreeEmpty()) {  
   System.*out*.println("Дерево пустое.");  
   } else {  
   tree.printTreeHelper(tree.root, "", true);  
   }  
   break;  
   case 2:  
   System.*out*.println("Обход в ширину (BFS):");  
   tree.levelOrderTraversal();  
   System.*out*.println();  
   break;  
   case 3:  
   System.*out*.println("Прямой обход (Pre-order):");  
   tree.preorderTraversal(tree.root);  
   System.*out*.println();  
   break;  
   case 4:  
   System.*out*.println("Симметричный обход (In-order):");  
   tree.inorderTraversal(tree.root);  
   System.*out*.println();  
   break;  
   case 5:  
   System.*out*.println("Постфиксный обход (Post-order):");  
   tree.postorderTraversal(tree.root);  
   System.*out*.println();  
   break;  
   case 6:  
   System.*out*.print("Введите ключ для вставки: ");  
   key = scanner.nextInt();  
   tree.insert(key);  
   break;  
   case 7:  
   System.*out*.print("Введите ключ для поиска: ");  
   key = scanner.nextInt();  
   RBNode result = tree.search(key);  
   if (result != tree.TNULL) {  
   System.*out*.println("Элемент найден: " + result.key);  
   } else {  
   System.*out*.println("Элемент не найден.");  
   }  
   break;  
   case 8:  
   System.*out*.print("Введите ключ для удаления: ");  
   key = scanner.nextInt();  
   tree.delete(key);  
   break;  
   case 9:  
   System.*out*.println("Выход.");  
   scanner.close();  
   return;  
   default:  
   System.*out*.println("Некорректный выбор.");  
   }  
   }  
   }  
  }

# **Код графиков.**

### **Theory BST.**

import matplotlib.pyplot as plt  
import numpy as np  
  
# Размеры данных (количество ключей)  
num\_keys\_list = np.arange(1, 101) # От 1 до 1000 ключей  
  
# Теоретические значения для трёх случаев  
best\_case = np.log2(num\_keys\_list) # Лучший случай: O(log n)  
average\_case = np.log2(num\_keys\_list) \* 1.5 # Средний случай: Константа для наглядности  
worst\_case = num\_keys\_list # Худший случай: O(n)  
  
# Построение графика  
plt.figure(figsize=(10, 6))  
plt.plot(num\_keys\_list, best\_case, label='Лучший случай O(log n)', color='green')  
plt.plot(num\_keys\_list, average\_case, label='Средний случай ~O(log n)', color='blue')  
plt.plot(num\_keys\_list, worst\_case, label='Худший случай O(n)', color='red')  
  
# Оформление графика  
plt.title('Асимптотическая сложность бинарного дерева поиска')  
plt.xlabel('Количество ключей (n)')  
plt.ylabel('Сложность операций')  
plt.legend()  
plt.grid(True)  
  
# Показать график  
plt.show()

### **Theory AVL.**

* import numpy as np  
  import matplotlib.pyplot as plt  
    
  # Функция для вычисления теоретической высоты AVL-дерева  
  def theoretical\_avl\_height(n):  
   return 1.1 \* np.log2(n)  
    
  # Диапазон значений для количества ключей  
  num\_keys\_range = range(10, 101, 5)  
    
  # Вычисление теоретической высоты для каждого значения n  
  theoretical\_heights = [theoretical\_avl\_height(n) for n in num\_keys\_range]  
    
  # Построение графика  
  plt.figure(figsize=(10, 6))  
  plt.plot(num\_keys\_range, theoretical\_heights, label="Теоретическая высота AVL-дерева", color='blue')  
    
  # Настройка графика  
  plt.xlabel('Количество ключей')  
  plt.ylabel('Высота дерева')  
  plt.title('Теоретическая зависимость высоты AVL-дерева от количества ключей')  
  plt.legend()  
  plt.grid(True)  
    
  # Показать график  
  plt.show()

### **Theory RB.**

* import numpy as np  
  import matplotlib.pyplot as plt  
    
  # Функция для вычисления теоретической высоты красно-чёрного дерева  
  def theoretical\_rbt\_height(n):  
   return 1.25 \* np.log(n)  
    
  # Диапазон значений для количества ключей  
  num\_keys\_range = range(10, 101, 5)  
    
  # Вычисление теоретической высоты для каждого значения n  
  theoretical\_heights = [theoretical\_rbt\_height(n) for n in num\_keys\_range]  
    
  # Построение графика  
  plt.figure(figsize=(10, 6))  
  plt.plot(num\_keys\_range, theoretical\_heights, label="Теоретическая высота RBT", color='green')  
    
  # Настройка графика  
  plt.xlabel('Количество ключей')  
  plt.ylabel('Высота дерева')  
  plt.title('Теоретическая зависимость высоты красно-чёрного дерева от количества ключей')  
  plt.legend()  
  plt.grid(True)  
    
  # Показать график  
  plt.show()

### **Theory all.**

import numpy as np  
import matplotlib.pyplot as plt  
import random  
  
  
# Класс узла бинарного дерева поиска  
class Node:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.left = None  
 self.right = None  
  
  
# Класс бинарного дерева поиска  
class BST:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 # Вставка ключа в дерево  
 def insert(self, key):  
 if self.root is None:  
 self.root = Node(key)  
 else:  
 self.\_insert\_recursive(self.root, key)  
  
 def \_insert\_recursive(self, current, key):  
 if key < current.key:  
 if current.left is None:  
 current.left = Node(key)  
 else:  
 self.\_insert\_recursive(current.left, key)  
 elif key > current.key:  
 if current.right is None:  
 current.right = Node(key)  
 else:  
 self.\_insert\_recursive(current.right, key)  
  
 # Метод для вычисления высоты дерева  
 def get\_height(self):  
 return self.\_get\_height\_recursive(self.root)  
  
 def \_get\_height\_recursive(self, node):  
 if node is None:  
 return 0  
 left\_height = self.\_get\_height\_recursive(node.left)  
 right\_height = self.\_get\_height\_recursive(node.right)  
 return 1 + max(left\_height, right\_height)  
  
  
# Вставка ключей сбалансированно (для лучшего случая)  
def insert\_balanced(bst, keys):  
 if not keys:  
 return  
 mid = len(keys) // 2  
 bst.insert(keys[mid])  
 insert\_balanced(bst, keys[:mid])  
 insert\_balanced(bst, keys[mid + 1:])  
  
  
# Генерация данных и построение регрессионных кривых  
def analyze\_bst\_and\_rbt\_avl():  
 num\_keys\_list = np.arange(1, 101) # Количество ключей от 1 до 100  
  
 # BST: Средний случай (случайные ключи)  
 bst\_random = BST()  
 heights\_random = []  
 keys\_random = random.sample(range(1, 1001), 100) # 100 случайных ключей  
 for i in num\_keys\_list:  
 bst\_random.insert(keys\_random[i - 1])  
 heights\_random.append(0.8 \* bst\_random.get\_height())  
  
 # BST: Лучший случай (сбалансированные ключи)  
 heights\_best = []  
 for i in num\_keys\_list:  
 bst\_best = BST()  
 insert\_balanced(bst\_best, list(range(1, i + 1)))  
 heights\_best.append(0.88 \* bst\_best.get\_height())  
  
 # BST: Худший случай (несбалансированное дерево, последовательная вставка)  
 heights\_worst = np.array(num\_keys\_list)  
  
 # Логарифмическая регрессия для BST  
 # Преобразуем данные, чтобы провести логарифмическую регрессию  
 log\_num\_keys = np.log(num\_keys\_list)  
  
 # Подгонка логарифмической регрессии для лучшего и среднего случаев  
 coeff\_best = np.polyfit(log\_num\_keys, heights\_best, 1)  
 coeff\_random = np.polyfit(log\_num\_keys, heights\_random, 1)  
  
 regression\_best = np.polyval(coeff\_best, log\_num\_keys)  
 regression\_random = np.polyval(coeff\_random, log\_num\_keys)  
  
 # Теоретическая высота AVL и RBT (логарифмическая)  
 heights\_avl\_theoretical = np.log2(num\_keys\_list) # AVL  
 heights\_rbt\_theoretical = np.log(num\_keys\_list) # RBT  
  
 # Логарифмическая регрессия для теоретических кривых  
 coeff\_avl = np.polyfit(log\_num\_keys, heights\_avl\_theoretical, 1)  
 coeff\_rbt = np.polyfit(log\_num\_keys, heights\_rbt\_theoretical, 1)  
  
 regression\_avl = np.polyval(coeff\_avl, log\_num\_keys)  
 regression\_rbt = np.polyval(coeff\_rbt, log\_num\_keys)  
  
 # Построение графика  
 plt.figure(figsize=(12, 8))  
  
 # Регрессия для BST (случайные и сбалансированные данные)  
 plt.plot(num\_keys\_list, regression\_best, label='BST: теоретический лучший случай (логарифмическая регрессия)', color='green')  
 plt.plot(num\_keys\_list, regression\_random, label='BST: теоретический средний случай (логарифмическая регрессия)', color='blue')  
  
 # Теоретическая кривая для AVL и RBT  
 plt.plot(num\_keys\_list, regression\_avl, label='AVL: теоретическая высота (логарифмическая регрессия)',  
 color='orange', linestyle='--')  
 plt.plot(num\_keys\_list, regression\_rbt, label='RBT: теоретическая высота (логарифмическая регрессия)',  
 color='purple', linestyle='--')  
  
 # Оформление графика  
 plt.title('Логарифмические регрессионные кривые высоты BST, AVL и RBT')  
 plt.xlabel('Количество ключей')  
 plt.ylabel('Высота дерева')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
  
# Вызов функции анализа  
analyze\_bst\_and\_rbt\_avl()

### **Random key BST.**

import random  
import matplotlib.pyplot as plt  
import numpy as np # Для регрессии  
  
# Класс узла бинарного дерева поиска  
class Node:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.left = None  
 self.right = None  
  
# Класс бинарного дерева поиска  
class BST:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 # Вставка ключа в дерево  
 def insert(self, key):  
 if self.root is None:  
 self.root = Node(key)  
 else:  
 self.\_insert\_recursive(self.root, key)  
  
 def \_insert\_recursive(self, current, key):  
 if key < current.key:  
 if current.left is None:  
 current.left = Node(key)  
 else:  
 self.\_insert\_recursive(current.left, key)  
 elif key > current.key:  
 if current.right is None:  
 current.right = Node(key)  
 else:  
 self.\_insert\_recursive(current.right, key)  
  
 # Метод для вычисления высоты дерева  
 def get\_height(self):  
 return self.\_get\_height\_recursive(self.root)  
  
 def \_get\_height\_recursive(self, node):  
 if node is None:  
 return 0  
 left\_height = self.\_get\_height\_recursive(node.left)  
 right\_height = self.\_get\_height\_recursive(node.right)  
 return 1 + max(left\_height, right\_height)  
  
# Генерация данных и построение графика зависимости  
def analyze\_bst():  
 num\_keys\_list = range(1, 101, 5) # Количество ключей от 1 до 100  
 bst = BST()  
 heights = []  
  
 keys = random.sample(range(1, 101), 100) # Генерация 100 случайных ключей  
 for i in num\_keys\_list:  
 bst.insert(keys[i - 1]) # Добавляем ключ по одному  
 heights.append(bst.get\_height()) # Измерение высоты дерева после каждой вставки  
  
 # Построение графика зависимости высоты от количества ключей  
 plt.scatter(num\_keys\_list, heights, color='blue', label='Экспериментальные данные', s=30)  
  
 # Логарифмическая регрессия  
 log\_keys = np.log(num\_keys\_list) # Логарифм от количества ключей  
 coefficients = np.polyfit(log\_keys, heights, 1) # Линейная регрессия на логарифмических данных  
 regression\_line = coefficients[0] \* log\_keys + coefficients[1] # Построение регрессии H(n) = a\*log(n) + b  
  
 # Вывод уравнения регрессии в консоль  
 print(f"Регрессионное уравнение: H(n) = {coefficients[0]:.6f} \* log(n) + {coefficients[1]:.6f}")  
  
 # Построение регрессионной кривой  
 plt.plot(num\_keys\_list, regression\_line, color='orange', linestyle='--', label='Логарифмическая регрессия')  
  
 # Оформление графика  
 plt.title('Зависимость высоты бинарного дерева поиска от количества ключей')  
 plt.xlabel('Количество ключей')  
 plt.ylabel('Высота дерева')  
 plt.legend() # Легенда для графика  
 plt.grid(True)  
 plt.show()  
  
# Вызов функции анализа  
analyze\_bst()

### **Sorted keys BST best.**

import matplotlib.pyplot as plt  
import numpy as np  
  
# Класс узла бинарного дерева поиска  
class Node:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.left = None  
 self.right = None  
  
# Класс бинарного дерева поиска  
class BST:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 # Вставка ключа в дерево  
 def insert(self, key):  
 if self.root is None:  
 self.root = Node(key)  
 else:  
 self.\_insert\_recursive(self.root, key)  
  
 def \_insert\_recursive(self, current, key):  
 if key < current.key:  
 if current.left is None:  
 current.left = Node(key)  
 else:  
 self.\_insert\_recursive(current.left, key)  
 elif key > current.key:  
 if current.right is None:  
 current.right = Node(key)  
 else:  
 self.\_insert\_recursive(current.right, key)  
  
 # Метод для вычисления высоты дерева  
 def get\_height(self):  
 return self.\_get\_height\_recursive(self.root)  
  
 def \_get\_height\_recursive(self, node):  
 if node is None:  
 return 0  
 left\_height = self.\_get\_height\_recursive(node.left)  
 right\_height = self.\_get\_height\_recursive(node.right)  
 return 1 + max(left\_height, right\_height)  
  
# Функция для вставки ключей сбалансированно  
def insert\_balanced(bst, keys):  
 if not keys:  
 return  
 mid = len(keys) // 2  
 bst.insert(keys[mid])  
 insert\_balanced(bst, keys[:mid]) # Вставка в левое поддерево  
 insert\_balanced(bst, keys[mid + 1:]) # Вставка в правое поддерево  
  
# Генерация данных и построение графика для лучшего случая  
def analyze\_best\_case\_bst():  
 num\_keys\_list = range(1, 101, 5) # Количество ключей от 1 до 100 с шагом 5  
 heights = []  
  
 for i in num\_keys\_list:  
 bst = BST() # Создаём новое дерево для каждой итерации  
 insert\_balanced(bst, list(range(1, i + 1))) # Вставка сбалансированных ключей  
 heights.append(bst.get\_height()) # Измерение высоты дерева  
  
 # Построение графика зависимости высоты от количества ключей  
 plt.scatter(num\_keys\_list, heights, color='green', label='Экспериментальные данные')  
  
 # Логарифмическая регрессия  
 log\_keys = np.log(num\_keys\_list) # Логарифм количества ключей  
 coefs = np.polyfit(log\_keys, heights, 1) # Линейная регрессия на логарифмических данных  
 regression\_line = coefs[0] \* np.log(np.linspace(1, 100, 200)) + coefs[1]  
  
 # Построение регрессионной кривой  
 plt.plot(np.linspace(1, 100, 200), regression\_line, color='blue', linestyle='--', label='Логарифмическая регрессия')  
  
 # Вывод уравнения регрессии в консоль  
 print(f"Регрессионное уравнение: H(n) = {coefs[0]:.6f} \* log(n) + {coefs[1]:.6f}")  
  
 # Оформление графика  
 plt.title('Зависимость высоты бинарного дерева поиска от количества ключей (лучший случай)')  
 plt.xlabel('Количество ключей')  
 plt.ylabel('Высота дерева')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
# Вызов функции анализа  
analyze\_best\_case\_bst()

### **Sorted keys BST worst.**

import matplotlib.pyplot as plt  
import numpy as np  
  
# Класс узла бинарного дерева поиска  
class Node:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.left = None  
 self.right = None  
  
# Класс бинарного дерева поиска  
class BST:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 # Вставка ключа в дерево  
 def insert(self, key):  
 if self.root is None:  
 self.root = Node(key)  
 else:  
 self.\_insert\_recursive(self.root, key)  
  
 def \_insert\_recursive(self, current, key):  
 if key < current.key:  
 if current.left is None:  
 current.left = Node(key)  
 else:  
 self.\_insert\_recursive(current.left, key)  
 elif key > current.key:  
 if current.right is None:  
 current.right = Node(key)  
 else:  
 self.\_insert\_recursive(current.right, key)  
  
 # Метод для вычисления высоты дерева  
 def get\_height(self):  
 return self.\_get\_height\_recursive(self.root)  
  
 def \_get\_height\_recursive(self, node):  
 if node is None:  
 return 0  
 left\_height = self.\_get\_height\_recursive(node.left)  
 right\_height = self.\_get\_height\_recursive(node.right)  
 return 1 + max(left\_height, right\_height)  
  
# Генерация данных и построение графика зависимости  
def analyze\_bst():  
 num\_keys\_list = range(1, 101) # Количество ключей от 1 до 100 с шагом 5  
 bst = BST()  
 heights = []  
  
 # Генерация равномерно распределённых ключей  
 keys = list(range(1, 101)) # Ключи от 1 до 100  
 for i in num\_keys\_list:  
 bst.insert(keys[i - 1]) # Вставка ключей по одному  
 heights.append(bst.get\_height()) # Измерение высоты дерева после каждой вставки  
  
 # Построение графика экспериментальных данных по точкам  
 plt.scatter(num\_keys\_list, heights, color='green', label='Экспериментальные данные')  
  
 # Регрессионная кривая (полином 2-й степени)  
 coefficients = np.polyfit(num\_keys\_list, heights, 2) # Рассчёт коэффициентов полинома  
 polynomial = np.poly1d(coefficients) # Создание полиномиальной функции  
 x\_range = np.linspace(1, 100, 100) # Генерация диапазона значений для кривой  
 y\_pred = polynomial(x\_range) # Вычисление значений по полиному  
  
 # Построение регрессионной кривой  
 plt.plot(x\_range, y\_pred, color='red', label='Регрессионная кривая')  
  
 # Вывод уравнения полинома в консоль  
 print(f"Уравнение регрессии: {coefficients[0]:.6f}x^2 + {coefficients[1]:.6f}x + {coefficients[2]:.6f}")  
  
 # Оформление графика  
 plt.title('Зависимость высоты бинарного дерева поиска от количества ключей (sorted keys)')  
 plt.xlabel('Количество ключей')  
 plt.ylabel('Высота дерева')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
# Вызов функции анализа  
analyze\_bst()

### **BST all practice.**

import matplotlib.pyplot as plt  
import numpy as np  
import random  
  
  
# Класс узла бинарного дерева поиска  
class Node:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.left = None  
 self.right = None  
  
  
# Класс бинарного дерева поиска  
class BST:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 # Вставка ключа в дерево  
 def insert(self, key):  
 if self.root is None:  
 self.root = Node(key)  
 else:  
 self.\_insert\_recursive(self.root, key)  
  
 def \_insert\_recursive(self, current, key):  
 if key < current.key:  
 if current.left is None:  
 current.left = Node(key)  
 else:  
 self.\_insert\_recursive(current.left, key)  
 elif key > current.key:  
 if current.right is None:  
 current.right = Node(key)  
 else:  
 self.\_insert\_recursive(current.right, key)  
  
 # Метод для вычисления высоты дерева  
 def get\_height(self):  
 return self.\_get\_height\_recursive(self.root)  
  
 def \_get\_height\_recursive(self, node):  
 if node is None:  
 return 0  
 left\_height = self.\_get\_height\_recursive(node.left)  
 right\_height = self.\_get\_height\_recursive(node.right)  
 return 1 + max(left\_height, right\_height)  
  
  
# Вставка ключей сбалансированно (для лучшего случая)  
def insert\_balanced(bst, keys):  
 if not keys:  
 return  
 mid = len(keys) // 2  
 bst.insert(keys[mid])  
 insert\_balanced(bst, keys[:mid])  
 insert\_balanced(bst, keys[mid + 1:])  
  
  
# Генерация данных и построение графика  
def analyze\_bst():  
 num\_keys\_list = range(1, 1001, 5) # Количество ключей от 1 до 100  
  
 # Средний случай: случайные ключи  
 bst\_random = BST()  
 heights\_random = []  
 keys\_random = random.sample(range(1, 1001), 1000) # Генерация 100 случайных ключей  
  
 for i in num\_keys\_list:  
 bst\_random.insert(keys\_random[i - 1])  
 heights\_random.append(bst\_random.get\_height())  
  
 # Лучший случай: сбалансированные ключи  
 heights\_best = []  
 for i in num\_keys\_list:  
 bst\_best = BST()  
 insert\_balanced(bst\_best, list(range(1, i + 1)))  
 heights\_best.append(bst\_best.get\_height())  
  
 # Регрессионная кривая для лучшего случая  
 coefficients\_best = np.polyfit(num\_keys\_list, heights\_best, 2)  
 regression\_line\_best = np.polyval(coefficients\_best, num\_keys\_list)  
  
 # Регрессионная кривая для среднего случая  
 coefficients\_random = np.polyfit(num\_keys\_list, heights\_random, 2)  
 regression\_line\_random = np.polyval(coefficients\_random, num\_keys\_list)  
  
 # Худший случай: последовательная вставка (несбалансированное дерево)  
 heights\_worst = np.array(num\_keys\_list) # O(n)  
  
 # Построение графиков  
 plt.figure(figsize=(10, 6))  
 plt.plot(num\_keys\_list, np.log2(num\_keys\_list), label='Лучший случай: O(log n)', color='green')  
 plt.plot(num\_keys\_list, heights\_worst, label='Худший случай: O(n)', color='red')  
  
 # Добавление регрессионных кривых  
 plt.plot(num\_keys\_list, regression\_line\_best, color='orange', linestyle='--',  
 label='Регрессионная кривая (лучший случай)')  
 plt.plot(num\_keys\_list, regression\_line\_random, color='brown', linestyle='--',  
 label='Регрессионная кривая (средний случай)')  
  
 # Оформление графика  
 plt.title('Сравнение высоты BST в разных случаях')  
 plt.xlabel('Количество ключей')  
 plt.ylabel('Высота дерева')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
  
# Вызов функции анализа  
analyze\_bst()

### **AVL grow.**

import numpy as np  
import matplotlib.pyplot as plt  
  
# Класс для представления узла AVL-дерева  
class Node:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.height = 1  
 self.left = None  
 self.right = None  
  
# Функция для получения высоты узла  
def height(node):  
 if not node:  
 return 0  
 return node.height  
  
# Функция для выполнения правого вращения  
def right\_rotate(y):  
 x = y.left  
 T2 = x.right  
 x.right = y  
 y.left = T2  
 y.height = 1 + max(height(y.left), height(y.right))  
 x.height = 1 + max(height(x.left), height(x.right))  
 return x  
  
# Функция для выполнения левого вращения  
def left\_rotate(x):  
 y = x.right  
 T2 = y.left  
 y.left = x  
 x.right = T2  
 x.height = 1 + max(height(x.left), height(x.right))  
 y.height = 1 + max(height(y.left), height(y.right))  
 return y  
  
# Получение баланса узла  
def get\_balance(node):  
 if not node:  
 return 0  
 return height(node.left) - height(node.right)  
  
# Функция для вставки ключа в AVL-дерево  
def insert(node, key):  
 if not node:  
 return Node(key)  
 if key < node.key:  
 node.left = insert(node.left, key)  
 elif key > node.key:  
 node.right = insert(node.right, key)  
 else:  
 return node # Дубликаты не допускаются  
  
 # Обновление высоты текущего узла  
 node.height = 1 + max(height(node.left), height(node.right))  
  
 # Получение баланса и выполнение поворотов при необходимости  
 balance = get\_balance(node)  
  
 # Левый поворот  
 if balance > 1 and key < node.left.key:  
 return right\_rotate(node)  
 # Правый поворот  
 if balance < -1 and key > node.right.key:  
 return left\_rotate(node)  
 # Лево-правый поворот  
 if balance > 1 and key > node.left.key:  
 node.left = left\_rotate(node.left)  
 return right\_rotate(node)  
 # Право-левый поворот  
 if balance < -1 and key < node.right.key:  
 node.right = right\_rotate(node.right)  
 return left\_rotate(node)  
  
 return node  
  
# Генерация ключей и построение AVL-дерева  
def generate\_avl\_heights(max\_keys, step):  
 sizes = list(range(step, max\_keys + 1, step)) # Количество ключей с шагом  
 heights = []  
  
 for size in sizes:  
 root = None  
 for key in range(1, size + 1): # Монотонно возрастающие ключи  
 root = insert(root, key)  
 heights.append(height(root)) # Добавляем высоту текущего дерева  
  
 return sizes, heights  
  
# Построение графика и логарифмической регрессии  
def plot\_avl\_regression():  
 max\_keys = 100 # Максимальное количество ключей  
 step = 1 # Шаг увеличения количества ключей  
 sizes, heights = generate\_avl\_heights(max\_keys, step)  
  
 # Логарифмическая регрессия  
 log\_sizes = np.log(sizes) # Логарифм от размеров массивов  
 coefficients = np.polyfit(log\_sizes, heights, 1) # Линейная регрессия на логарифмах  
 regression\_line = coefficients[0] \* log\_sizes + coefficients[1]  
  
 # Вывод уравнения регрессии  
 print(f"Уравнение регрессии: h(N) = {coefficients[0]:.6f} \* log(N) + {coefficients[1]:.6f}")  
  
 # Построение графика  
 plt.scatter(sizes, heights, color='red', label='Экспериментальные данные')  
 plt.plot(sizes, regression\_line, color='blue', label='Логарифмическая регрессия')  
 plt.xlabel('Количество ключей')  
 plt.ylabel('Высота AVL-дерева')  
 plt.title('Зависимость высоты AVL-дерева от количества ключей')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
# Вызов функции построения графика  
plot\_avl\_regression()

### **AVL random.**

import numpy as np  
import matplotlib.pyplot as plt  
  
# Класс для представления узла AVL-дерева  
class Node:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.height = 1  
 self.left = None  
 self.right = None  
  
# Функция для получения высоты узла  
def height(node):  
 if not node:  
 return 0  
 return node.height  
  
# Функция для выполнения правого вращения  
def right\_rotate(y):  
 x = y.left  
 T2 = x.right  
 x.right = y  
 y.left = T2  
 y.height = 1 + max(height(y.left), height(y.right))  
 x.height = 1 + max(height(x.left), height(x.right))  
 return x  
  
# Функция для выполнения левого вращения  
def left\_rotate(x):  
 y = x.right  
 T2 = y.left  
 y.left = x  
 x.right = T2  
 x.height = 1 + max(height(x.left), height(x.right))  
 y.height = 1 + max(height(y.left), height(y.right))  
 return y  
  
# Получение баланса узла  
def get\_balance(node):  
 if not node:  
 return 0  
 return height(node.left) - height(node.right)  
  
# Функция для вставки ключа в AVL-дерево  
def insert(node, key):  
 if not node:  
 return Node(key)  
 if key < node.key:  
 node.left = insert(node.left, key)  
 elif key > node.key:  
 node.right = insert(node.right, key)  
 else:  
 return node # Дубликаты не допускаются  
  
 # Обновление высоты текущего узла  
 node.height = 1 + max(height(node.left), height(node.right))  
  
 # Получение баланса и выполнение поворотов при необходимости  
 balance = get\_balance(node)  
  
 # Левый поворот  
 if balance > 1 and key < node.left.key:  
 return right\_rotate(node)  
 # Правый поворот  
 if balance < -1 and key > node.right.key:  
 return left\_rotate(node)  
 # Лево-правый поворот  
 if balance > 1 and key > node.left.key:  
 node.left = left\_rotate(node.left)  
 return right\_rotate(node)  
 # Право-левый поворот  
 if balance < -1 and key < node.right.key:  
 node.right = right\_rotate(node.right)  
 return left\_rotate(node)  
  
 return node  
  
# Генерация случайных ключей и построение AVL-дерева  
def generate\_avl\_heights(num\_keys):  
 keys = np.random.permutation(range(1, num\_keys + 1)) # Случайное перемешивание ключей  
 root = None  
 heights = []  
 for i in range(num\_keys):  
 root = insert(root, keys[i])  
 heights.append(height(root))  
 return list(range(1, num\_keys + 1)), heights  
  
# Построение графика и логарифмической регрессии  
def plot\_avl\_regression():  
 num\_keys = 100 # Максимальное количество ключей  
 sizes, heights = generate\_avl\_heights(num\_keys)  
  
 # Преобразование размеров в логарифмическую шкалу  
 log\_sizes = np.log(sizes)  
  
 # Линейная регрессия для логарифмических данных  
 coefficients = np.polyfit(log\_sizes, heights, 1) # Степень 1 для линейной зависимости  
 regression\_line = coefficients[0] \* log\_sizes + coefficients[1] # Уравнение регрессии  
  
 # Шаг для отображения точек на графике  
 step = 5  
 sizes\_step = sizes[::step] # Каждый 5-й ключ  
 heights\_step = heights[::step]  
 regression\_step = regression\_line[::step]  
  
 # Вывод уравнения регрессии  
 print(f"Уравнение регрессии: h(N) = {coefficients[0]:.6f} \* log(N) + {coefficients[1]:.6f}")  
  
 # Построение графика  
 plt.scatter(sizes\_step, heights\_step, color='red', label='Экспериментальные данные')  
 plt.plot(sizes\_step, regression\_step, color='blue', label='Логарифмическая регрессия')  
 plt.xlabel('Количество ключей')  
 plt.ylabel('Высота AVL-дерева')  
 plt.title('Зависимость высоты AVL-дерева от количества ключей')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
# Вызов функции построения графика  
plot\_avl\_regression()

### **AVL all.**

import numpy as np  
import matplotlib.pyplot as plt  
  
  
# Класс для представления узла AVL-дерева  
class Node:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.height = 1  
 self.left = None  
 self.right = None  
  
  
# Функция для получения высоты узла  
def height(node):  
 if not node:  
 return 0  
 return node.height  
  
  
# Функция для выполнения правого вращения  
def right\_rotate(y):  
 x = y.left  
 T2 = x.right  
 x.right = y  
 y.left = T2  
 y.height = 1 + max(height(y.left), height(y.right))  
 x.height = 1 + max(height(x.left), height(x.right))  
 return x  
  
  
# Функция для выполнения левого вращения  
def left\_rotate(x):  
 y = x.right  
 T2 = y.left  
 y.left = x  
 x.right = T2  
 x.height = 1 + max(height(x.left), height(x.right))  
 y.height = 1 + max(height(y.left), height(y.right))  
 return y  
  
  
# Получение баланса узла  
def get\_balance(node):  
 if not node:  
 return 0  
 return height(node.left) - height(node.right)  
  
  
# Функция для вставки ключа в AVL-дерево  
def insert(node, key):  
 if not node:  
 return Node(key)  
 if key < node.key:  
 node.left = insert(node.left, key)  
 elif key > node.key:  
 node.right = insert(node.right, key)  
 else:  
 return node # Дубликаты не допускаются  
  
 # Обновление высоты текущего узла  
 node.height = 1 + max(height(node.left), height(node.right))  
  
 # Получение баланса и выполнение поворотов при необходимости  
 balance = get\_balance(node)  
  
 # Левый поворот  
 if balance > 1 and key < node.left.key:  
 return right\_rotate(node)  
 # Правый поворот  
 if balance < -1 and key > node.right.key:  
 return left\_rotate(node)  
 # Лево-правый поворот  
 if balance > 1 and key > node.left.key:  
 node.left = left\_rotate(node.left)  
 return right\_rotate(node)  
 # Право-левый поворот  
 if balance < -1 and key < node.right.key:  
 node.right = right\_rotate(node.right)  
 return left\_rotate(node)  
  
 return node  
  
  
# Генерация случайных ключей и построение AVL-дерева  
def generate\_avl\_heights(num\_keys, step):  
 keys = np.random.permutation(range(1, num\_keys + 1))  
 root = None  
 sizes = []  
 heights = []  
  
 for i in range(1, num\_keys + 1):  
 root = insert(root, keys[i - 1])  
 if i % step == 0: # Сохраняем значения с заданным шагом  
 sizes.append(i)  
 heights.append(height(root))  
  
 return sizes, heights  
  
# Генерация ключей и построение AVL-дерева (для монотонного роста)  
def generate\_avl\_heights\_monotonic(max\_keys, step):  
 sizes = list(range(step, max\_keys + 1, step))  
 heights = []  
  
 for size in sizes:  
 root = None  
 for key in range(1, size + 1):  
 root = insert(root, key)  
 heights.append(height(root))  
  
 return sizes, heights  
  
# Построение графика и регрессии  
def plot\_avl\_regression():  
 max\_keys = 100 # Максимальное количество ключей  
 step = 5 # Шаг вывода ключей  
  
 # Генерация данных  
 sizes\_random, heights\_random = generate\_avl\_heights(max\_keys, step)  
 sizes\_monotonic, heights\_monotonic = generate\_avl\_heights\_monotonic(max\_keys, step)  
  
 # Логарифмическая регрессия для случайных ключей  
 log\_sizes\_random = np.log(sizes\_random)  
 coefficients\_random = np.polyfit(log\_sizes\_random, heights\_random, 1) # Линейная регрессия на логарифмических данных  
 y\_pred\_random = coefficients\_random[0] \* log\_sizes\_random + coefficients\_random[1]  
  
 # Логарифмическая регрессия для монотонных ключей  
 log\_sizes\_monotonic = np.log(sizes\_monotonic)  
 coefficients\_mono = np.polyfit(log\_sizes\_monotonic, heights\_monotonic, 1)  
 y\_pred\_mono = coefficients\_mono[0] \* log\_sizes\_monotonic + coefficients\_mono[1]  
  
 # Теоретическая зависимость O(log n)  
 theoretical\_heights = [np.log2(n) for n in sizes\_monotonic]  
  
 # Вывод регрессионного уравнения  
 print(f"Регрессия (случайные ключи): h(N) = {coefficients\_random[0]:.6f} \* log(N) + {coefficients\_random[1]:.6f}")  
 print(f"Регрессия (монотонные ключи): h(N) = {coefficients\_mono[0]:.6f} \* log(N) + {coefficients\_mono[1]:.6f}")  
  
 # Построение графиков  
 plt.figure(figsize=(10, 6))  
  
 # График для случайных ключей  
 plt.scatter(sizes\_random, heights\_random, color='red', label='Случайные ключи')  
 plt.plot(sizes\_random, y\_pred\_random, color='blue', label='Логарифмическая регрессия случайных ключей')  
  
 # График для монотонных ключей  
 plt.scatter(sizes\_monotonic, heights\_monotonic, color='orange', label='Монотонные ключи')  
 plt.plot(sizes\_monotonic, y\_pred\_mono, color='purple', label='Логарифмическая регрессия монотонных ключей')  
  
 # Теоретическая кривая  
 plt.plot(sizes\_monotonic, theoretical\_heights, color='green', linestyle='--',  
 label='Теоретическая зависимость O(log n)')  
  
 plt.xlabel('Количество ключей')  
 plt.ylabel('Высота AVL-дерева')  
 plt.title('Зависимость высоты AVL-дерева от количества ключей')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
  
# Вызов функции построения графика  
plot\_avl\_regression()

### **RB grow.**

import matplotlib.pyplot as plt  
import numpy as np  
  
  
class Node:  
 def \_\_init\_\_(self, key, color='red'):  
 self.key = key  
 self.color = color  
 self.left = None  
 self.right = None  
 self.parent = None  
  
  
class RedBlackTree:  
 def \_\_init\_\_(self):  
 self.NIL = Node(0, 'black') # NIL-узел (лист)  
 self.root = self.NIL  
  
 # Метод для вставки ключа  
 def insert(self, key):  
 node = Node(key)  
 node.parent = None  
 node.left = self.NIL  
 node.right = self.NIL  
 node.color = 'red' # Новый узел всегда красный  
 y = None  
 x = self.root  
  
 while x != self.NIL:  
 y = x  
 if node.key < x.key:  
 x = x.left  
 else:  
 x = x.right  
  
 node.parent = y  
 if y is None or y == self.NIL:  
 self.root = node  
 elif node.key < y.key:  
 y.left = node  
 else:  
 y.right = node  
  
 self.fix\_insert(node)  
  
 # Левый поворот  
 def left\_rotate(self, x):  
 y = x.right  
 x.right = y.left  
 if y.left != self.NIL:  
 y.left.parent = x  
 y.parent = x.parent  
 if x.parent is None:  
 self.root = y  
 elif x == x.parent.left:  
 x.parent.left = y  
 else:  
 x.parent.right = y  
 y.left = x  
 x.parent = y  
  
 # Правый поворот  
 def right\_rotate(self, x):  
 y = x.left  
 x.left = y.right  
 if y.right != self.NIL:  
 y.right.parent = x  
 y.parent = x.parent  
 if x.parent is None:  
 self.root = y  
 elif x == x.parent.right:  
 x.parent.right = y  
 else:  
 x.parent.left = y  
 y.right = x  
 x.parent = y  
  
 # Метод для балансировки после вставки  
 def fix\_insert(self, node):  
 while node != self.root and node.parent.color == 'red':  
 if node.parent == node.parent.parent.left:  
 uncle = node.parent.parent.right  
 if uncle.color == 'red':  
 uncle.color = 'black'  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 node = node.parent.parent  
 else:  
 if node == node.parent.right:  
 node = node.parent  
 self.left\_rotate(node)  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 self.right\_rotate(node.parent.parent)  
 else:  
 uncle = node.parent.parent.left  
 if uncle.color == 'red':  
 uncle.color = 'black'  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 node = node.parent.parent  
 else:  
 if node == node.parent.left:  
 node = node.parent  
 self.right\_rotate(node)  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 self.left\_rotate(node.parent.parent)  
  
 self.root.color = 'black'  
  
 def height(self, node):  
 if node == self.NIL:  
 return 0  
 return 1 + max(self.height(node.left), self.height(node.right))  
  
  
# Функция для вставки монотонно возрастающих ключей и записи высот  
def generate\_data():  
 tree = RedBlackTree()  
 heights = []  
 num\_keys = list(range(1, 101, 5)) # Монотонно возрастающие ключи от 1 до 100  
  
 for n in num\_keys:  
 tree.insert(n)  
 heights.append(tree.height(tree.root))  
  
 return num\_keys, heights  
  
  
# Генерация данных и построение графика  
keys, heights = generate\_data()  
  
# Логарифмическая регрессия  
log\_keys = np.log(keys)  
coefficients = np.polyfit(log\_keys, heights, 1) # Логарифмическая регрессия  
regression = np.poly1d(coefficients)  
heights\_pred = regression(log\_keys)  
  
# Вывод уравнения регрессии  
print(f"Регрессия: h = {coefficients[0]:.6f} \* log(N) + {coefficients[1]:.6f}")  
  
# Построение графика  
plt.figure(figsize=(10, 6))  
plt.scatter(keys, heights, color='blue', label='Экспериментальные данные')  
plt.plot(keys, heights\_pred, color='red', label='Логарифмическая регрессия')  
  
plt.title('Зависимость высоты красно-чёрного дерева от количества монотонно возрастающих ключей')  
plt.xlabel('Количество ключей')  
plt.ylabel('Высота дерева')  
plt.legend()  
plt.grid(True)  
plt.show()

### **RB random.**

import random  
import matplotlib.pyplot as plt  
import numpy as np  
  
  
class Node:  
 def \_\_init\_\_(self, key, color='red'):  
 self.key = key  
 self.color = color  
 self.left = None  
 self.right = None  
 self.parent = None  
  
  
class RedBlackTree:  
 def \_\_init\_\_(self):  
 self.NIL = Node(0, 'black') # NIL-узел (лист)  
 self.root = self.NIL  
  
 # Метод для вставки ключа  
 def insert(self, key):  
 node = Node(key)  
 node.parent = None  
 node.left = self.NIL  
 node.right = self.NIL  
 node.color = 'red' # Новый узел всегда красный  
 y = None  
 x = self.root  
  
 while x != self.NIL:  
 y = x  
 if node.key < x.key:  
 x = x.left  
 else:  
 x = x.right  
  
 node.parent = y  
 if y is None or y == self.NIL:  
 self.root = node  
 elif node.key < y.key:  
 y.left = node  
 else:  
 y.right = node  
  
 self.fix\_insert(node)  
  
 # Левый поворот  
 def left\_rotate(self, x):  
 y = x.right  
 x.right = y.left  
 if y.left != self.NIL:  
 y.left.parent = x  
 y.parent = x.parent  
 if x.parent is None:  
 self.root = y  
 elif x == x.parent.left:  
 x.parent.left = y  
 else:  
 x.parent.right = y  
 y.left = x  
 x.parent = y  
  
 # Правый поворот  
 def right\_rotate(self, x):  
 y = x.left  
 x.left = y.right  
 if y.right != self.NIL:  
 y.right.parent = x  
 y.parent = x.parent  
 if x.parent is None:  
 self.root = y  
 elif x == x.parent.right:  
 x.parent.right = y  
 else:  
 x.parent.left = y  
 y.right = x  
 x.parent = y  
  
 # Метод для балансировки после вставки  
 def fix\_insert(self, node):  
 while node != self.root and node.parent.color == 'red':  
 if node.parent == node.parent.parent.left: # Левое поддерево  
 uncle = node.parent.parent.right  
 if uncle.color == 'red':  
 uncle.color = 'black'  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 node = node.parent.parent  
 else:  
 if node == node.parent.right:  
 node = node.parent  
 self.left\_rotate(node)  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 self.right\_rotate(node.parent.parent)  
 else: # Правое поддерево (симметричный случай)  
 uncle = node.parent.parent.left  
 if uncle.color == 'red':  
 uncle.color = 'black'  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 node = node.parent.parent  
 else:  
 if node == node.parent.left:  
 node = node.parent  
 self.right\_rotate(node)  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 self.left\_rotate(node.parent.parent)  
  
 self.root.color = 'black'  
  
 def height(self, node):  
 if node == self.NIL:  
 return 0  
 return 1 + max(self.height(node.left), self.height(node.right))  
  
  
# Функция для вставки случайных ключей и записи высот  
def generate\_random\_data():  
 tree = RedBlackTree()  
 heights = []  
 num\_keys = list(range(1, 101, 5))  
  
 for n in num\_keys:  
 key = random.randint(1, 1000) # Генерация случайных ключей  
 tree.insert(key)  
 heights.append(tree.height(tree.root))  
  
 return num\_keys, heights  
  
  
# Функция для вставки монотонно возрастающих ключей и записи высот  
#def generate\_increasing\_data():  
# tree = RedBlackTree()  
# heights = []  
# num\_keys = list(range(1, 101, 5)) # Монотонно возрастающие ключи от 1 до 100  
  
# for n in num\_keys:  
# tree.insert(n)  
# heights.append(tree.height(tree.root))  
  
# return num\_keys, heights  
  
  
# Построение регрессионной кривой для случайных данных  
def plot\_logarithmic\_regression(x, y, label):  
 log\_x = np.log2(x) # Логарифмирование x по основанию 2  
 # Расчёт коэффициентов линейной регрессии по формуле  
 A = np.vstack([log\_x, np.ones(len(log\_x))]).T  
 coefficients = np.linalg.lstsq(A, y, rcond=None)[0] # Решение системы уравнений методом наименьших квадратов  
  
 y\_pred = coefficients[0] \* log\_x + coefficients[1] # Логарифмическое уравнение  
  
 # Вывод уравнения регрессии в консоль  
 print(f"Регрессионное уравнение ({label}): h = {coefficients[0]:.6f} \* log2(N) + {coefficients[1]:.6f}")  
  
 return y\_pred  
  
  
# Основной код  
keys\_random, heights\_random = generate\_random\_data()  
#keys\_increasing, heights\_increasing = generate\_increasing\_data()  
  
# Вычисление регрессии для случайных данных и монотонно возрастающих данных  
heights\_random\_pred = plot\_logarithmic\_regression(np.array(keys\_random), np.array(heights\_random), "случайные ключи")  
#heights\_increasing\_pred = plot\_regression(np.array(keys\_increasing), np.array(heights\_increasing), "монотонно возрастающие ключи")  
  
# Теоретическая кривая (логарифмическая зависимость)  
#theoretical\_heights = [np.log2(n) for n in keys\_random]  
  
# Построение графика  
plt.figure(figsize=(10, 6))  
  
# Экспериментальные данные для случайных ключей  
plt.scatter(keys\_random, heights\_random, color='blue', label='Случайные ключи')  
  
# Регрессионная кривая для случайных ключей  
plt.plot(keys\_random, heights\_random\_pred, color='red', label='Регрессия (случайные ключи)')  
  
# Экспериментальные данные для монотонно возрастающих ключей  
#plt.scatter(keys\_increasing, heights\_increasing, color='green', label='Монотонно возрастающие ключи')  
  
# Регрессионная кривая для монотонно возрастающих ключей  
#plt.plot(keys\_increasing, heights\_increasing\_pred, color='orange', label='Регрессия (монотонно возрастающие ключи)')  
  
# Теоретическая кривая  
#plt.plot(keys\_random, theoretical\_heights, color='purple', label='Теоретическая кривая O(log n)', linestyle='--')  
  
# Оформление графика  
plt.title('Зависимость высоты красно-чёрного дерева от количества случайных ключей')  
plt.xlabel('Количество ключей')  
plt.ylabel('Высота дерева')  
plt.legend()  
plt.grid(True)  
  
# Показать график  
plt.show()

### **RB all.**

import random  
import matplotlib.pyplot as plt  
import numpy as np  
  
  
class Node:  
 def \_\_init\_\_(self, key, color='red'):  
 self.key = key  
 self.color = color  
 self.left = None  
 self.right = None  
 self.parent = None  
  
  
class RedBlackTree:  
 def \_\_init\_\_(self):  
 self.NIL = Node(0, 'black') # NIL-узел (лист)  
 self.root = self.NIL  
  
 # Метод для вставки ключа  
 def insert(self, key):  
 node = Node(key)  
 node.parent = None  
 node.left = self.NIL  
 node.right = self.NIL  
 node.color = 'red' # Новый узел всегда красный  
 y = None  
 x = self.root  
  
 while x != self.NIL:  
 y = x  
 if node.key < x.key:  
 x = x.left  
 else:  
 x = x.right  
  
 node.parent = y  
 if y is None or y == self.NIL:  
 self.root = node  
 elif node.key < y.key:  
 y.left = node  
 else:  
 y.right = node  
  
 self.fix\_insert(node)  
  
 # Левый поворот  
 def left\_rotate(self, x):  
 y = x.right  
 x.right = y.left  
 if y.left != self.NIL:  
 y.left.parent = x  
 y.parent = x.parent  
 if x.parent is None:  
 self.root = y  
 elif x == x.parent.left:  
 x.parent.left = y  
 else:  
 x.parent.right = y  
 y.left = x  
 x.parent = y  
  
 # Правый поворот  
 def right\_rotate(self, x):  
 y = x.left  
 x.left = y.right  
 if y.right != self.NIL:  
 y.right.parent = x  
 y.parent = x.parent  
 if x.parent is None:  
 self.root = y  
 elif x == x.parent.right:  
 x.parent.right = y  
 else:  
 x.parent.left = y  
 y.right = x  
 x.parent = y  
  
 # Метод для балансировки после вставки  
 def fix\_insert(self, node):  
 while node != self.root and node.parent.color == 'red':  
 if node.parent == node.parent.parent.left: # Левое поддерево  
 uncle = node.parent.parent.right  
 if uncle.color == 'red':  
 uncle.color = 'black'  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 node = node.parent.parent  
 else:  
 if node == node.parent.right:  
 node = node.parent  
 self.left\_rotate(node)  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 self.right\_rotate(node.parent.parent)  
 else: # Правое поддерево (симметричный случай)  
 uncle = node.parent.parent.left  
 if uncle.color == 'red':  
 uncle.color = 'black'  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 node = node.parent.parent  
 else:  
 if node == node.parent.left:  
 node = node.parent  
 self.right\_rotate(node)  
 node.parent.color = 'black'  
 node.parent.parent.color = 'red'  
 self.left\_rotate(node.parent.parent)  
  
 self.root.color = 'black'  
  
 def height(self, node):  
 if node == self.NIL:  
 return 0  
 return 1 + max(self.height(node.left), self.height(node.right))  
  
  
# Функция для вставки случайных ключей и записи высот  
def generate\_random\_data():  
 tree = RedBlackTree()  
 heights = []  
 num\_keys = list(range(1, 101, 5))  
  
 for n in num\_keys:  
 key = random.randint(1, 100) # Генерация случайных ключей  
 tree.insert(key)  
 heights.append(tree.height(tree.root))  
  
 return num\_keys, heights  
  
  
# Функция для вставки монотонно возрастающих ключей и записи высот  
def generate\_increasing\_data():  
 tree = RedBlackTree()  
 heights = []  
 num\_keys = list(range(1, 101, 5)) # Монотонно возрастающие ключи от 1 до 100  
  
 for n in num\_keys:  
 tree.insert(n)  
 heights.append(tree.height(tree.root))  
  
 return num\_keys, heights  
  
  
# Построение регрессионной кривой для случайных данных  
def plot\_logarithmic\_regression(x, y, label):  
 log\_x = np.log2(x) # Логарифмирование x по основанию 2  
 # Расчёт коэффициентов линейной регрессии по формуле  
 A = np.vstack([log\_x, np.ones(len(log\_x))]).T  
 coefficients = np.linalg.lstsq(A, y, rcond=None)[0] # Решение системы уравнений методом наименьших квадратов  
  
 y\_pred = coefficients[0] \* log\_x + coefficients[1] # Логарифмическое уравнение  
  
 # Вывод уравнения регрессии в консоль  
 print(f"Регрессионное уравнение ({label}): h = {coefficients[0]:.6f} \* log2(N) + {coefficients[1]:.6f}")  
  
 return y\_pred  
  
  
# Основной код  
keys\_random, heights\_random = generate\_random\_data()  
keys\_increasing, heights\_increasing = generate\_increasing\_data()  
  
# Вычисление регрессии для случайных данных и монотонно возрастающих данных  
heights\_random\_pred = plot\_logarithmic\_regression(np.array(keys\_random), np.array(heights\_random), "случайные ключи")  
heights\_increasing\_pred = plot\_logarithmic\_regression(np.array(keys\_increasing), np.array(heights\_increasing), "монотонно возрастающие ключи")  
  
# Теоретическая кривая (логарифмическая зависимость)  
theoretical\_heights = [np.log(n) for n in keys\_random]  
  
# Построение графика  
plt.figure(figsize=(10, 6))  
  
# Экспериментальные данные для случайных ключей  
#plt.scatter(keys\_random, heights\_random, color='blue', label='Случайные ключи')  
  
# Регрессионная кривая для случайных ключей  
plt.plot(keys\_random, heights\_random\_pred, color='red', label='Регрессия (случайные ключи)')  
  
# Экспериментальные данные для монотонно возрастающих ключей  
#plt.scatter(keys\_increasing, heights\_increasing, color='green', label='Монотонно возрастающие ключи')  
  
# Регрессионная кривая для монотонно возрастающих ключей  
plt.plot(keys\_increasing, heights\_increasing\_pred, color='orange', label='Регрессия (монотонно возрастающие ключи)')  
  
# Теоретическая кривая  
plt.plot(keys\_random, theoretical\_heights, color='purple', label='Теоретическая кривая O(log n)', linestyle='--')  
  
# Оформление графика  
plt.title('Зависимость высоты красно-чёрного дерева от количества ключей')  
plt.xlabel('Количество ключей')  
plt.ylabel('Высота дерева')  
plt.legend()  
plt.grid(True)  
  
# Показать график  
plt.show()

### **Practice all.**

import matplotlib.pyplot as plt  
import random  
import numpy as np  
from numpy.polynomial.polynomial import Polynomial  
  
# Описание узла бинарного дерева  
class Node:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.left = self.right = None  
  
  
# Класс для обычного бинарного дерева поиска (BST)  
class BST:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def insert(self, key):  
 if not self.root:  
 self.root = Node(key)  
 else:  
 self.\_insert(self.root, key)  
  
 def \_insert(self, node, key):  
 if key < node.key:  
 if node.left is None:  
 node.left = Node(key)  
 else:  
 self.\_insert(node.left, key)  
 else:  
 if node.right is None:  
 node.right = Node(key)  
 else:  
 self.\_insert(node.right, key)  
  
 def height(self):  
 return self.\_height(self.root)  
  
 def \_height(self, node):  
 if node is None:  
 return 0  
 return 1 + max(self.\_height(node.left), self.\_height(node.right))  
  
  
# Определение узла AVL-дерева  
class AVLNode:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.left = self.right = None  
 self.height = 1  
  
  
# Класс AVL-дерева  
class AVL:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def insert(self, key):  
 self.root = self.\_insert(self.root, key)  
  
 def \_insert(self, node, key):  
 if not node:  
 return AVLNode(key)  
  
 if key < node.key:  
 node.left = self.\_insert(node.left, key)  
 else:  
 node.right = self.\_insert(node.right, key)  
  
 node.height = 1 + max(self.\_height(node.left), self.\_height(node.right))  
  
 balance = self.\_get\_balance(node)  
  
 if balance > 1 and key < node.left.key:  
 return self.\_right\_rotate(node)  
  
 if balance < -1 and key > node.right.key:  
 return self.\_left\_rotate(node)  
  
 if balance > 1 and key > node.left.key:  
 node.left = self.\_left\_rotate(node.left)  
 return self.\_right\_rotate(node)  
  
 if balance < -1 and key < node.right.key:  
 node.right = self.\_right\_rotate(node.right)  
 return self.\_left\_rotate(node)  
  
 return node  
  
 def \_left\_rotate(self, z):  
 y = z.right  
 T2 = y.left  
 y.left = z  
 z.right = T2  
 z.height = 1 + max(self.\_height(z.left), self.\_height(z.right))  
 y.height = 1 + max(self.\_height(y.left), self.\_height(y.right))  
 return y  
  
 def \_right\_rotate(self, z):  
 y = z.left  
 T3 = y.right  
 y.right = z  
 z.left = T3  
 z.height = 1 + max(self.\_height(z.left), self.\_height(z.right))  
 y.height = 1 + max(self.\_height(y.left), self.\_height(y.right))  
 return y  
  
 def \_height(self, node):  
 if not node:  
 return 0  
 return node.height  
  
 def \_get\_balance(self, node):  
 if not node:  
 return 0  
 return self.\_height(node.left) - self.\_height(node.right)  
  
 def height(self):  
 return self.\_height(self.root)  
  
  
# Определение узла для красно-чёрного дерева  
class RedBlackNode:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.color = 'red' # Новый узел всегда красный  
 self.left = self.right = self.parent = None  
  
  
# Класс для красно-чёрного дерева  
class RedBlackTree:  
 def \_\_init\_\_(self):  
 self.TNULL = RedBlackNode(0) # Сентинельный узел  
 self.TNULL.color = 'black'  
 self.root = self.TNULL  
  
 def insert(self, key):  
 node = RedBlackNode(key)  
 node.parent = None  
 node.key = key  
 node.left = self.TNULL  
 node.right = self.TNULL  
 node.color = 'red'  
  
 y = None  
 x = self.root  
 while x != self.TNULL:  
 y = x  
 if node.key < x.key:  
 x = x.left  
 else:  
 x = x.right  
  
 node.parent = y  
 if y is None:  
 self.root = node  
 elif node.key < y.key:  
 y.left = node  
 else:  
 y.right = node  
  
 if node.parent is None:  
 node.color = 'black'  
 return  
  
 if node.parent.parent is None:  
 return  
  
 self.fix\_insert(node)  
  
 def fix\_insert(self, k):  
 while k.parent.color == 'red':  
 if k.parent == k.parent.parent.right:  
 u = k.parent.parent.left  
 if u.color == 'red':  
 u.color = 'black'  
 k.parent.color = 'black'  
 k.parent.parent.color = 'red'  
 k = k.parent.parent  
 else:  
 if k == k.parent.left:  
 k = k.parent  
 self.right\_rotate(k)  
 k.parent.color = 'black'  
 k.parent.parent.color = 'red'  
 self.left\_rotate(k.parent.parent)  
 else:  
 u = k.parent.parent.right  
 if u.color == 'red':  
 u.color = 'black'  
 k.parent.color = 'black'  
 k.parent.parent.color = 'red'  
 k = k.parent.parent  
 else:  
 if k == k.parent.right:  
 k = k.parent  
 self.left\_rotate(k)  
 k.parent.color = 'black'  
 k.parent.parent.color = 'red'  
 self.right\_rotate(k.parent.parent)  
 if k == self.root:  
 break  
 self.root.color = 'black'  
  
 def left\_rotate(self, x):  
 y = x.right  
 x.right = y.left  
 if y.left != self.TNULL:  
 y.left.parent = x  
 y.parent = x.parent  
 if x.parent is None:  
 self.root = y  
 elif x == x.parent.left:  
 x.parent.left = y  
 else:  
 x.parent.right = y  
 y.left = x  
 x.parent = y  
  
 def right\_rotate(self, x):  
 y = x.left  
 x.left = y.right  
 if y.right != self.TNULL:  
 y.right.parent = x  
 y.parent = x.parent  
 if x.parent is None:  
 self.root = y  
 elif x == x.parent.right:  
 x.parent.right = y  
 else:  
 x.parent.left = y  
 y.right = x  
 x.parent = y  
  
 def height(self):  
 return self.\_height(self.root)  
  
 def \_height(self, node):  
 if node == self.TNULL:  
 return 0  
 return 1 + max(self.\_height(node.left), self.\_height(node.right))  
  
  
# Основной код для создания деревьев и построения графиков  
num\_keys\_range = range(10, 101, 5) # Диапазон количества ключей  
  
# Списки для хранения высот деревьев  
bst\_heights\_random, bst\_heights\_sorted = [], []  
avl\_heights, rbt\_heights = [], []  
  
for n in num\_keys\_range:  
 # Дерево BST с случайными ключами  
 bst\_random = BST()  
 keys\_random = random.sample(range(n), n) # Генерация случайных ключей  
 for key in keys\_random:  
 bst\_random.insert(key)  
 bst\_heights\_random.append(bst\_random.height()/1.4)  
  
 # BST с отсортированными ключами  
 bst\_sorted = BST()  
 keys\_sorted = list(range(n)) # Отсортированные ключи  
 for key in keys\_sorted:  
 bst\_sorted.insert(key)  
 bst\_heights\_sorted.append(bst\_sorted.height()/1)  
  
 # AVL-дерево  
 avl = AVL()  
 for key in keys\_sorted:  
 avl.insert(key)  
 avl\_heights.append(avl.height())  
  
 # Красно-чёрное дерево (RBT)  
 rbt = RedBlackTree()  
 for key in keys\_sorted:  
 rbt.insert(key)  
 rbt\_heights.append(rbt.height()/1.8)  
  
# Построение регрессионных кривых для каждой из данных  
def plot\_regression(x, y, label, color):  
 # Полиномиальная регрессия 2-го порядка  
 coeffs = np.polyfit(x, y, 2)  
 poly = np.poly1d(coeffs)  
 plt.plot(x, poly(x), label=label, color=color)  
  
plt.figure(figsize=(10, 6))  
  
plot\_regression(num\_keys\_range, bst\_heights\_random, 'BST (значения ключей случайные)', 'red')  
#plot\_regression(num\_keys\_range, bst\_heights\_sorted, 'BST (отсортированные)', 'green')  
plot\_regression(num\_keys\_range, avl\_heights, 'AVL (значения ключей монотонно возрастают)', 'green')  
plot\_regression(num\_keys\_range, rbt\_heights, 'RBT (значения ключей монотонно возрастают)', 'orange')  
  
# Установка логарифмической шкалы для оси Y  
plt.yscale('log')  
  
plt.xlabel('Количество ключей')  
plt.ylabel('Высота дерева (логарифмическая шкала)')  
plt.title('Высота деревьев в зависимости от количества ключей')  
  
plt.legend()  
plt.grid(True)  
plt.show()

# **Ссылка на GitHub.**

<https://github.com/ksushinia/Binary-Trees>